Invariant Graph Propagation in Constraint-Based Local Search

FREJ KNUTAR LEWANDER*, PIERRE FLENER, and JUSTIN PEARSON, Uppsala University, Sweden

In constraint-based local search, an assignment to the search variables is improved upon by an iterative procedure that replaces the current assignment with a similar assignment. The latter is selected by a heuristic that assesses the qualities of a subset of all similar assignments, where the quality of such assignments is determined via a process called invariant graph propagation. Since, typically, many similar assignments are considered in every iteration, invariant graph propagation must be as efficient as possible. Since invariant graph propagation is independent of the selection heuristic, any comparison between different invariant graph propagation styles under different selection heuristics can be misleading. In this paper, we describe and compare both theoretically and empirically the throughput of several invariant graph propagation styles, and give criteria when one style or another is to be used.

JAIR Track: Surveys

JAIR Associate Editor: Ken Brown

JAIR Reference Format:
Frej Knutar Lewander, Pierre Flener, and Justin Pearson. 2025. Invariant Graph Propagation in Constraint-Based Local Search. *Journal of Artificial Intelligence Research* 83, Article 10 (July 2025), 42 pages. DOI: 10.1613/jair.1.17482

1 Introduction

Constraint-based local search (CBLS) is a technology where constraint satisfaction problems and constrained optimisation problems are modelled and solved. It typically scales well to larger problem instances, at the expense that it does not provide the guarantee of optimality that systematic-search technologies (such as integer programming and constraint programming) offer. In CBLS, a problem is modelled by a directed graph, called the *invariant graph*, with variables and invariants as nodes, where each *invariant* defines some variables in terms of others. A current assignment to the source variables of the graph is initialised and then iteratively replaced with a selected similar assignment, called a *neighbour*. Much research has been done on selection heuristics, which focus on what neighbours to probe and how to select a neighbour at each iteration, and on how to escape local optima of the objective function. Independent of the selection heuristic is the assessment of the quality of a neighbour and the replacement of the current assignment with a neighbour, both performed by a process called *invariant graph propagation*, where the non-source variables are updated according to the invariants. The efficiency of invariant graph propagation is crucial for the overall performance of a CBLS solver.

After we give the necessary background (Section 2), our contributions are:

- the collection of invariant graph propagation algorithms from the literature and their presentation in a uniform way (Sections 3.1 to 3.4);
- proofs of correctness for the presented invariant graph propagation algorithms;
- the computation of the time complexities of these algorithms (Section 3.5);

*Corresponding Author.

Authors' Contact Information: Frej Knutar Lewander, ORCID: https://orcid.org/0009-0009-6215-4666, frej.knutar.lewander@it.uu.se; Pierre Flener, ORCID: https://orcid.org/0000-0001-8730-4098, pierre.flener@it.uu.se; Justin Pearson, ORCID: https://orcid.org/0000-0002-0084-8891, justin.pearson@it.uu.se, Uppsala University, Department of Information Technology, Uppsala, Sweden.

DOI: 10.1613/jair.1.17482

- the theoretical comparison of these time complexities, to recommend an invariant graph propagation algorithm based on features of a given invariant graph (Section 3.6);
- the measuring of the throughput (number of probes per second) of the algorithms for various invariant graphs, validating our recommendations (Section 4).

Finally, we conclude in Section 5.

Note that we here assume that the value of each sink node variable (such as the objective variable) must be updated. However, a selection heuristic could be extended to choose only some variables (typically a subset of the sink node variables) whose values must be updated [13], possibly improving the throughput of invariant graph propagation as the values of fewer variables must be updated. In the worst case, which happens often in CBLS, the extended selection heuristic would choose all sink node variables, as in our assumption. Note that no such assumption is made in [13], as their use case is not CBLS and does not have the notion of a selection heuristic. Additionally, they have a single algorithm and therefore perform no comparisons of algorithms. Because of our assumption, the computation of the time complexities of invariant graph propagation algorithms in Section 3.5, our recommendation of an invariant graph propagation algorithm in Section 3.6, and the measuring of the throughput in Section 4 are restricted to invariant graphs where the values of all sink node variables are always required to be updated.

2 Background

CBLS was made popular by Comet [23] and its predecessor Localizer [18, 24, 17]. Many other CBLS solvers have been designed, such as iOpt [26], Kangaroo [20], Hexaly (formerly known as LocalSolver) [4], InCELL [21], OscaR.cbls [6], fzn-oscar-cbls [5], Yuck [16], and Athanor [1, 2]. The source code of Localizer, iOpt, and Kangaroo is currently unobtainable to the public; Hexaly is commercial, and so was Comet.

We now present the concepts of CBLS that are relevant to our purpose, using a running example of how a constraint satisfaction problem is modelled in CBLS before giving a detailed comparison with related work in Section 2.10. Note that all concepts are defined mathematically: efficient implementations in terms of data structures and algorithms will be discussed in Section 3.

2.1 Constraint-Based Local Search

In CBLS, a problem is modelled by a directed graph, called the *invariant graph*, with variables and invariants as nodes, where each *invariant* defines some variables in terms of others. A current assignment to the source variables of the graph is initialised and then iteratively replaced with a selected similar assignment, called a *neighbour*. During each iteration, first some neighbours are *probed*, where their quality is determined, and then a neighbour is selected to be *moved to*, where the current assignment is replaced by it. Both probing and moving are performed by a process called *invariant graph propagation*, where the non-source variables are updated according to the invariants.

In order to perform invariant graph propagation efficiently, the current assignment must be initialised so that the following *loop condition* holds for all iterations: the value of each variable is (uniquely) determinable under the current assignment. Before an iteration, the loop condition must hold, and it must also hold for the probed neighbour that is selected by the selection heuristic, as it becomes the new current assignment. This loop condition is present in the CBLS solvers Localizer [24], iOpt [26], Comet [23], Kangaroo [20], Hexaly [4], InCELL [21], OscaR.cbls [6], Yuck [16], and Athanor [1, 2], and by extension also in fzn-oscar-cbls [5], which makes use of OscaR.cbls.

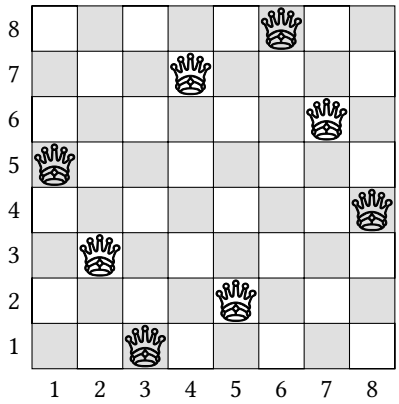

Fig. 1. A chessboard that depicts a solution to the n-queens problem for $n = 8$.

```
1  int: n; % number of queens
2  % Row[c] = the row of the queen in column c;
3  % enforces that all queens are in distinct columns:
4  array[1..n] of var 1..n: Row;
5  % all queens are in distinct rows:
6  constraint all_different( Row                    );
7  % all queens are in distinct upward diagonals:
8  constraint all_different([Row[c]+c | c in 1..n]);
9  % all queens are in distinct downward diagonals:
10 constraint all_different([Row[c]-c | c in 1..n]);
```

Listing 1. A MiniZinc model for the n-queens problem.

2.2 Running Example: The n-Queens Problem

In order to show how a constraint satisfaction problem is modelled in CBLS, we now give a running example that will be used throughout Section 2. The n-queens problem consists of n queens that are to be placed on an n by n chessboard such that no two queens are on the same row, column, or diagonal. Figure 1 depicts a solution to the 8-queens instance.

A model for n-queens in the solver-independent MiniZinc language [19] is in Listing 1. Line 1 declares the parameter `n`. Line 4 declares the array `Row` of `n` decision variables, where `Row[c]` denotes the row of the queen in column `c`, thereby enforcing that no two queens are on the same column. The three constraints on lines 6 to 10 enforce that no two queens are on the same row, upward diagonal, or downward diagonal, respectively.

2.3 Invariants, Input Variables, and Output Variables

An *invariant*, called a *one-way constraint* in [26], functionally defines some variables, called its *output variables*, by a total deterministic function on one or more variables, called its *input variables*, and must invariably hold at all iterations of the local search. A variable may not be an output variable of more than one invariant.

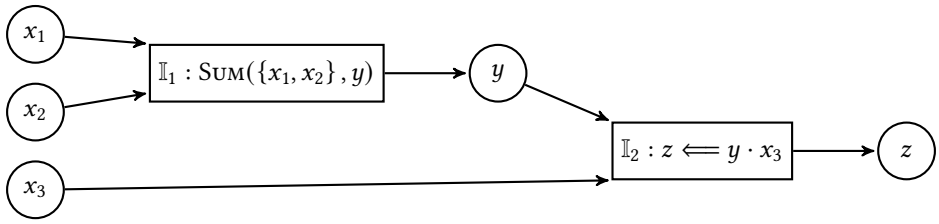

Fig. 2. An invariant graph with five variables, $\{x_1, x_2, x_3, y, z\}$; three search variables, $\{x_1, x_2, x_3\}$; two invariants, $\{\mathbb{I}_1, \mathbb{I}_2\}$; and one probed variable, z. The variables x_1 and x_2 are the inputs to $\mathbb{I}_1$, while x_3 and y are the inputs to $\mathbb{I}_2$. The variable y is the output of $\mathbb{I}_1$ and z is the output of $\mathbb{I}_2$.

For example, given the variable set $\mathcal{X}$ and the variable y, the $\textsc{Sum}(\mathcal{X}, y)$ invariant functionally defines the output variable y to be the sum of the input variables x of $\mathcal{X}$, and is here denoted by $y \Longleftarrow \sum_{x \in \mathcal{X}} x$ instead of the syntax $y = \sum_{x \in \mathcal{X}} x$ of an equality constraint.

2.4 Invariant Graphs, Search Variables, and Probed Variables

The invariants and variables together induce a directed graph, called the *invariant graph*, where the variables and invariants are the nodes. There is an edge from each variable x to each invariant that x is an input to and there is an edge from each invariant to each of its output variables. If there is a path from node u to node v, then v is said to *depend* on u.

Consider an invariant graph $\mathcal{G}$ with nodes $\mathcal{N}$ and edges $\mathcal{E}$. No invariant $\mathbb{I} \in \mathcal{N}$ is a source or sink node of $\mathcal{G}$, as there exists at least one edge in $\mathcal{E}$ to $\mathbb{I}$ from its input variables and there exists at least one edge in $\mathcal{E}$ from $\mathbb{I}$ to its output variables. Thus, all source and sink nodes in $\mathcal{G}$ are variables. Each source node in $\mathcal{G}$ is called a *search variable*, as it is not an output variable of any invariant and is instead assigned a value by search (see Section 2.9). Each sink node in $\mathcal{G}$ is called a *probed variable*, such as the objective variable of an optimisation problem, as its value is assessed by the selection heuristic. If a variable x is neither an input to some invariant nor an output of some invariant, then x is both a search variable and a probed variable. An example of an invariant graph is given in Figure 2, where circled nodes are variables and boxed nodes are invariants.

In Kangaroo [20] each invariant and its output variables are merged into a single node in the invariant graph. In Hexaly [4] and InCELL [21] the invariant graph is required to be acyclic. In fzn-oscar-cbls [5] and Yuck [16] the invariant graph is created from a high-level MiniZinc [19] model, which has no syntax for invariants. In Athanor [1, 2] the invariant graph is instead created from the abstract syntax tree of an Essence [9] model, which also has no syntax for invariants.

2.5 Assignments and Values

In CBLS, each variable has an associated value at any time of the search (see Section 2.9). The value of each search variable is initialised when search starts and potentially updated during search. The value of each non-search variable x is recursively given by the invariant $\mathbb{I}$ with x as output variable and the values of the input variables to $\mathbb{I}$:

Definition 2.1 (assignment and value)**.** Consider an invariant graph with variables $\mathcal{V}$ and the set $\mathcal{S} \subseteq \mathcal{V}$ of search variables. An *assignment* is a total function $\alpha \colon \mathcal{S} \to \mathbb{Z}$ that maps each search variable to an integer value.

The *value* $\mathrm{v}_\alpha(x)$ of variable $x \in \mathcal{V}$ under assignment α is recursively determined as follows:

$$\mathrm{v}_\alpha(x) = \begin{cases} \alpha(x) & \text{if } x \in \mathcal{S} \\ \mathrm{v}_\alpha(e) & \text{otherwise, where } x \Longleftarrow e \text{ is the invariant that defines variable } x, \\ & \text{and } \mathrm{v}_\alpha \text{ is pushed down to the operands of expression } e. \end{cases}$$

For example, consider the variables $\mathcal{X} = \{x_1, \ldots, x_n\}$, the variable y, the $\textsc{Sum}(\mathcal{X}, y)$ invariant $y \Longleftarrow \sum_{x \in \mathcal{X}} x$, and an assignment α. The value of y under α is

$$\mathrm{v}_\alpha(y) = \mathrm{v}_\alpha\left(\sum_{x \in \mathcal{X}} x\right) = \sum_{x \in \mathcal{X}} \mathrm{v}_\alpha(x)$$

2.6 Hard, Softened, Soft, and Implicit Constraints

In a satisfaction or optimisation problem, a *constraint* is a relation over some variables that must be satisfied in each solution. For example, for the variables $\mathcal{X} = \{x_1, \ldots, x_n\}$ and an assignment α, the *AllDifferent*$(\mathcal{X})$ constraint is the n-ary relation over $\mathcal{X}$ that requires the values of the variables in $\mathcal{X}$ to be pairwise distinct, that is $\forall 1 \leq i < j \leq n : \mathrm{v}_\alpha(x_i) \neq \mathrm{v}_\alpha(x_j)$.

A hard equality constraint of a problem can (but does not have to) be modelled by an invariant that defines one of its variables. For example, the constraint $y = \sum_{x \in \mathcal{X}} x$ can be modelled as the $\textsc{Sum}(\mathcal{X}, y)$ invariant, which can even be rewritten by the solver into $\textsc{Sum}(\mathcal{X} \setminus \{x\} \cup \{-y\}, -x)$ for some $x \in \mathcal{X}$, if that is preferable for some reason.

A hard constraint of a problem can be *softened* by the modeller into an invariant, called a *violation invariant*, that functionally defines a variable y, called a *violation variable*, so that $\mathrm{v}_\alpha(y) = 0$ if the constraint is satisfied under the assignment α, else $\mathrm{v}_\alpha(y) > 0$ and $\mathrm{v}_\alpha(y)$ is typically proportional to the amount of violation of the constraint under α (rather than always the value 1). The softened constraints of a problem may thus be violated during search (see Section 2.9), but the aim is to satisfy them all, as they are actually hard. The soft constraints of an optimisation problem are also modelled by violation invariants.

For example, a hard *AllDifferent*$(\mathcal{X})$ constraint can be softened into the violation invariant $\textsc{AllDifferent}(\mathcal{X}, y)$, which defines the violation variable y to be the minimum number of variables of $\mathcal{X}$ whose values need to be updated for the constraint to become satisfied. For instance, consider $n = 5$ and an assignment α with $\mathrm{v}_\alpha(x_1) = \mathrm{v}_\alpha(x_2) = \mathrm{v}_\alpha(x_3) = 1$ and $\mathrm{v}_\alpha(x_4) = \mathrm{v}_\alpha(x_5) = 2$. We have $\mathrm{v}_\alpha(y) = 2 + 1 = 3$, as at least two variables among x_1, x_2, and x_3 as well as at least one variable among x_4 and x_5 must get suitable new values for the constraint to become satisfied.

The sum of the violation variables of the softened hard constraints of a problem is called the *total violation variable* and is defined by a Sum invariant. The value it takes under an assignment α indicates how close α is to denoting a solution, where 0 corresponds to a solution and a positive value corresponds to a non-solution. The invariant graph for a satisfaction problem then corresponds to an optimisation problem where the total violation variable is probed and is the objective variable that is to be minimised, ideally down to 0, but this is impossible for an unsatisfiable problem instance. Conversely, the invariant graph for an optimisation problem can have the total violation variable of the softened hard constraints, the total violation variable of the soft constraints, and the objective variable as probed variables, or make them inputs to an invariant that arithmetically combines them into a unique probed variable.

For example, consider the n-queens problem (see Section 2.2). A naïve invariant graph is in Figure 3 and corresponds to Listing 1. The output violation variables v_1, v_2, and v_3 take the value 0 if there are no queens on the same upward diagonal, downward diagonal, and row respectively, otherwise they take the number of queens that have to be moved away from shared such slices of the chessboard. The probed total violation variable v is the sum of v_1, v_2, and v_3: any assignment where v is 0 is a solution to the problem.

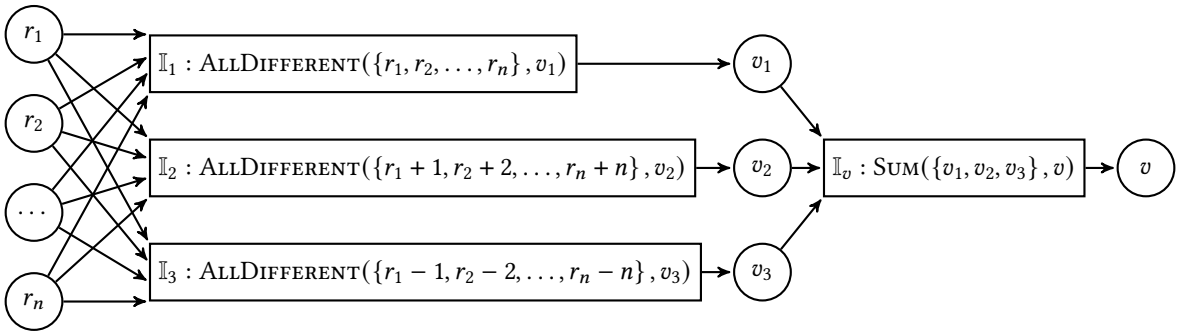

Fig. 3. Naïve invariant graph for the n-queens problem: it has $n + 4$ variables; n search variables, $\{r_1, r_2, \ldots, r_n\}$; 1 probed variable, v; 4 invariants, $\{\mathbb{I}_1, \mathbb{I}_2, \mathbb{I}_3, \mathbb{I}_v\}$; and 3 violation invariants, $\{\mathbb{I}_1, \mathbb{I}_2, \mathbb{I}_3\}$. For simplicity, the additions and subtractions to the input variables of invariants $\mathbb{I}_2$ and $\mathbb{I}_3$ are not depicted as further invariants.

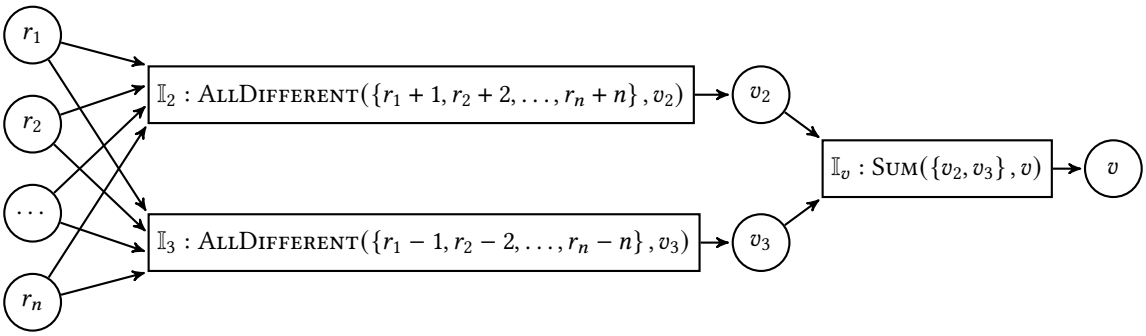

Fig. 4. Good invariant graph for the n-queens problem: it has $n + 3$ variables; n search variables, $\{r_1, r_2, \ldots, r_n\}$; 1 probed variable, v; 3 invariants, $\{\mathbb{I}_2, \mathbb{I}_3, \mathbb{I}_v\}$; and 2 violation invariants, $\{\mathbb{I}_2, \mathbb{I}_3\}$. This is a subgraph of Figure 3 as nodes $\mathbb{I}_1$ and v_1 were removed: the constraint that there is at most one queen per row was made implicit.

A hard constraint of a problem can also be made *implicit* by the modeller: it is not in the invariant graph given to the solver, but made to hold for all assignments by an initialisation heuristic and a selection heuristic (see Section 2.9), both provided by the modeller, that both only consider assignments that satisfy the constraint. For example, the *AllDifferent*($\mathcal{X}$) constraint is made implicit in the initial assignment α when the values of the variables in $\mathcal{X}$ are distinct under α. If no such initial assignment exists, then there exists no solution to the problem instance. During search, only assignments where the values of the variables in $\mathcal{X}$ remain distinct must then be considered. If the variables in $\mathcal{X}$ are search variables, then this is done by swapping the values of two variables or by updating the value of a variable to some value of no other variable.

For example, reconsider the n-queens problem. A good invariant graph is in Figure 4, as the constraint in line 6 of Listing 1 was made implicit.

2.7 Static and Dynamic Input Variables and Invariants

For an invariant, only a subset of the input variables under an assignment α might be required to determine the values of its output variables under α.

For example, consider the array $\mathcal{X} = [x_1, \ldots, x_n]$ of variables as well as the variables i and y. The Element$(\mathcal{X}, i, y)$ invariant defines y to be equal to the variable in $\mathcal{X}$ at index i, denoted by $y \Longleftarrow \mathcal{X}[i]$. Variable i is required under *almost all* assignments for determining the value of y; as an example of an exception, if all variables in $\mathcal{X}$ are the same variable, $[x, x, \ldots, x]$, then i is not actually required for determining the value of y. We say that each such input variable is a *static input variable*. However, only one of the other input variables, namely $\mathcal{X}[\mathrm{v}_\alpha(i)]$, is required under *any* assignment α to determine $\mathrm{v}_\alpha(y)$, but we do not know up-front which one: variable i has a different status than all variables of $\mathcal{X}$. We say that each such input variable is a *dynamic input variable*.

As another example, consider the variables v, x, y, and z, where v takes a non-negative value and typically is a violation variable. The IfThenElse(v, x_1, x_2, y) invariant defines y to be equal to x_1 if v takes value 0, and otherwise defines y to be equal to x_2. Variable v is a static input variable, while variables x_1 and x_2 are dynamic input variables.

Definition 2.2 (static and dynamic invariants)**.** If an invariant has one or more dynamic input variables, then it is called a *dynamic invariant*, else it is called a *static invariant*.

Static and dynamic invariants will be handled differently in the rest of this paper. For a CBLS solver, some invariants can be implemented as dynamic invariants (such as Element and IfThenElse), but some invariant propagation algorithms sometimes require a static counterpart (where all input variables are static inputs), as we will see in Section 3.4.1. A static counterpart of a dynamic invariant can always safely be used, but with a performance loss, as we will see in Section 3.5.

2.8 Static and Dynamic Cycles in an Invariant Graph

Consider an acyclic invariant graph (such as required by Hexaly and InCELL) with the variables $\mathcal{V}$ and an assignment α. For each variable $x \in \mathcal{V}$, its value $\mathrm{v}_\alpha(x)$ is determined because (i) variable x is output of at most one invariant and (ii) variable x cannot depend on itself since the invariant graph is acyclic. Requiring the invariant graph to be acyclic thus trivially satisfies the CBLS loop condition (see Section 2.1). If the value of a variable is not determinable, then it is *undeterminable* as the variable either takes no or more than one value.

Other CBLS solvers allow cyclic invariant graphs, as not all cycles violate the loop condition that the value of each variable is determinable under the current assignment. Cycles actually enable legitimate and competitive models for many problems, such as the travelling salesperson problem with time windows [3] (see Section 4.3). The potential presence of cycles in an invariant graph makes the satisfaction of the CBLS loop condition a subtle issue, as discussed next.

If an invariant graph has a cycle where each edge to an invariant originates from a *static* input variable, then it is not guaranteed that the values of all output variables of all invariants in that cycle are determinable under all assignments. As a minimal but artificial example, consider assignment α and the static invariant $y \Longleftarrow x \cdot y$ with both x and y as static inputs, and y also as output: if $\mathrm{v}_\alpha(x) = 0$, then $\mathrm{v}_\alpha(y)$ is determined to be 0, otherwise the value of y is undeterminable, as y either takes more than one value or no value for $\mathrm{v}_\alpha(x) = 1$ and $\mathrm{v}_\alpha(x) \notin \{0, 1\}$ respectively. We now define two types of cycles in an invariant graph.

Definition 2.3 (static and dynamic cycles)**.** Consider an invariant graph with a cycle c. If there exists an edge in c from a dynamic input variable to an invariant, then c is called a *dynamic cycle*, else c is called a *static cycle*.

For example, the real-world invariant subgraph in Figure 5 (whose grey backgrounds of some nodes will be explained in Section 3.4.1) has the dynamic cycle $\langle y_1, \mathbb{I}_2, y_2, \mathbb{I}_1, y_1 \rangle$ because of, for instance, the edge from the dynamic input variable y_2 to the dynamic invariant $\mathbb{I}_1$.

In theory, the detection if the value of each variable on a static cycle under an assignment is determinable is possible using a complete solver (such as a constraint programming solver). In practice however, this is not done by any CBLS solver as it would be too inefficient. Therefore, the invariant graph must not contain static cycles.

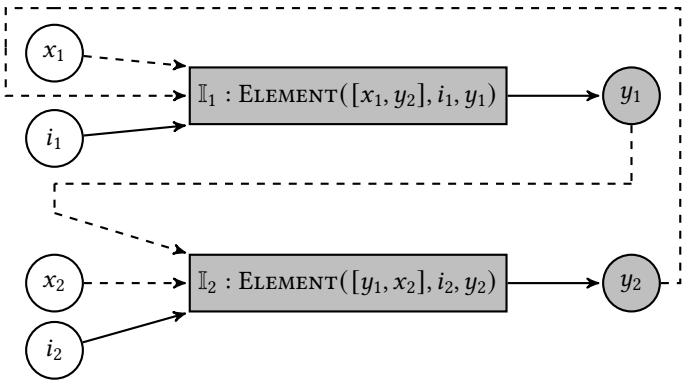

Fig. 5. A cyclic invariant subgraph with six variables, $\{x_1, x_2, i_1, i_2, y_1, y_2\}$; two dynamic invariants, $\{\mathbb{I}_1, \mathbb{I}_2\}$; two static input variables, $\{i_1, i_2\}$, with outgoing solid edges; four dynamic input variables, $\{x_1, x_2, y_1, y_2\}$, with outgoing dashed edges; and two output variables, $\{y_1, y_2\}$. The cycle $\langle y_1, \mathbb{I}_2, y_2, \mathbb{I}_1, y_1 \rangle$ is dynamic. The white nodes are in level 1, and the grey ones in level 2 (see Section 3.4.1). This is a subgraph of an invariant graph for the travelling salesperson problem with time windows [3] (see Section 4.3), where y_1 and y_2 are the arrival times at two different locations.

We will show at the end of this subsection how an invariant graph with static cycles can be transformed into an equivalent invariant graph without static cycles.

If an invariant graph has a cycle where at least one edge to an invariant originates from a *dynamic* input variable, then there may exist an assignment under which the values of all output variables of all invariants in that cycle are determined. As a minimal but artificial example, consider the dynamic invariant Element($[x, y], i, y$), which denotes $y \Longleftarrow [x, y][i]$, with variable i as static input, variables x and y as dynamic inputs, and y also as output: if i is 1 (note that we index from 1 throughout this paper), then y takes the same (and determined) value as x, else the value of y is undeterminable. Some CBLS solvers, such as Localizer, Comet, and OscaR.cbls, make use of algorithms that have preconditions on the dynamic cycles of the invariant graph, hence they only allow specific dynamic cycles (see Section 3.4.1); other CBLS solvers, such as Hexaly and InCELL, allow no dynamic cycles; and the remaining CBLS solvers, such as iOpt and Kangaroo, allow all dynamic cycles

Any cyclic invariant graph $\mathcal{G}$ can be automatically made acyclic [5]. In particular, each invariant graph with *static* cycles must be transformed by the modeller or solver into an invariant graph without static cycles, as there is otherwise no guarantee that the values of all variables are determinable under all assignments. A cycle c is removed from the invariant graph by splitting one output and input variable x on c into an output variable x_1 and an input variable x_2 and softening their equality constraint $x_1 = x_2$ into the violation invariant $v \Longleftarrow |x_1 - x_2|$ or $v \Longleftarrow \min(1, |x_1 - x_2|)$, with the new violation variable v added as another input to the Sum invariant that defines the total violation variable [5]. Note that adding search variables and equality violation invariants typically worsens the performance of a CBLS solver, as additional variables and invariants must be considered in probes and moves. Furthermore, finding a solution then becomes more difficult for a CBLS solver, as the value of each additional violation variable must be zero in every solution. We discuss dynamic cycles further in Section 3.2.

2.9 Search, Initialisation, Neighbours, Selection, Probes, Moves, Heuristics

A CBLS solver conducts search by maintaining a *current assignment*, which is initialised by an *initialisation heuristic* and then iteratively updated by a selection heuristic, both provided by the modeller or solver.

Definition 2.4 (neighbour)**.** A *neighbour* of an assignment α is an assignment α' where the values of one or more search variables are different from those under α. Each variable x with $\mathrm{v}_\alpha(x) \neq \mathrm{v}_{\alpha'}(x)$ is called an *updated variable*, whether it is a search variable or not.

At every iteration of local search, the *selection heuristic* explores a non-empty subset of the neighbours of the current assignment:

Definition 2.5 (probe and probing)**.** Consider an invariant graph and a neighbour α' of the current assignment. The determination of the values of the probed variables under α' is called *probing* α' or making a *probe* of α'.

After probing, the selection heuristic selects one probed neighbour and makes it the current assignment, thereby the current iteration of local search is ended:

Definition 2.6 (move and moving)**.** Consider a probed and selected neighbour α' of the current assignment α. The switch to α' as the current assignment is called *moving* from α to α' or making a *move* to α'.

When probing a neighbour α' of the current assignment, if the value of some variable is undeterminable under α', which both can happen when the variable is on a dynamic cycle, then α' may not be selected and moved to.

In Localizer [17], probing a neighbour is called *simulating a move*. In Comet [23], probing is called *simulating*. In Kangaroo [20], probing and moving are referred to as being in the *simulation* and *execution* phases respectively.

In practice, one also needs a *meta-heuristic*, provided by the modeller or the solver itself, in order to break out from local optima of the probed variables. Two popular meta-heuristics are simulated annealing [14] and tabu search [11].

Consider the n-queens problem (see Section 2.2). The good invariant graph in Figure 4 is generated automatically from the MiniZinc model in Listing 1 by fzn-oscar-cbls [5]: the hard constraints of the model are softened or made implicit automatically, instead of by the modeller. The generated selection heuristic only considers moves that swap the values of two of the variables in $\{r_1, r_2, \ldots, r_n\}$.

2.10 Related Work

Invariant graph propagation algorithms were inspired by algorithms for updating graphical user interfaces as well as for circuit simulations [**Alpern:IncEvalOfCompCircuits**, 7, 22, 12, 13].

There are two main invariant graph propagation styles, here called *output-to-input* and *input-to-output*. The output-to-input propagation style is called the *nullification/reevaluation approach* in [22], the *mark-sweep strategy* in [25, 26], and the *mark-sweep approach* in [20]; it is used in iOpt [26] and Kangaroo [20]; it is a recursive approach that starts with the probed variables and ends with the search variables. Conversely, the input-to-output propagation style is called the *topological ordering strategy* in [22, 25] and the *incremental graph evaluation* in [12]; it is used in Localizer [17], iOpt [26], Comet [23], Hexaly [4], InCELL [21], OscaR.cbls [6], Yuck [16], and Athanor [1, 2]; it is an iterative approach that starts with the search variables and ends with the probed variables.

With risk of oversimplification, we here label each instance of an algorithm as either *inside* or *outside the CBLS domain*, depending on whether the publication venue of the algorithm covers CBLS or not. Many of the algorithms published until 2001 were outside the CBLS domain; therefore our use of the algorithms differs from those in [**Alpern:IncEvalOfCompCircuits**, 7, 22, 12, 13, 25], with the following notable differences:

- inside the CBLS domain, after the invariant graph has been constructed it is typically not modified, while outside the CBLS domain, the counterpart of the invariant graph is typically modified between propagations;
- inside the CBLS domain, neighbouring assignments are probed before moving to one of them, while outside the CBLS domain, there is no notion of probing, as propagation only corresponds to moves; and
- inside the CBLS domain, each variable takes exactly one value, while outside the CBLS domain, there is no limit to the number of values (often called *attributes*) a node corresponding to a variable has.

Analyses of the running time for output-to-input propagation style algorithms were made in [7, 22, 13], but the former two are only high-level informal analyses and the latter assumes that the nodes of the (invariant) graph have bounded in-degree and out-degree. In contrast, we lift the latter assumption in Section 3 and make detailed formal analyses there, of both styles.

Inside the domain of CBLS and for the input-to-output propagation algorithm, considerable work has been given in [18, 24, 17, 23]. We contribute by giving the proof of when dynamic cycles are allowed (see Lemma 3.1 in Section 3.4.1) and by extending the reasoning of only the dynamic ELEMENT invariant in [24] to any dynamic invariant.

Additionally, we contribute by being the first to present the output-to-input and input-to-output styles in a uniform way (see Algorithms 1 and 2 respectively). We then give proofs of correctness for these algorithms (see Theorems 3.1 and 3.3 respectively), before computing their time complexities (see Theorems 3.5 and 3.6 respectively) and making an analysis (see Section 3.6) in order to recommend an invariant graph propagation style based on features of a given invariant graph.

Furthermore, empirical comparison between the performance of invariant graph propagation algorithms has been anecdotal in [25, 26] or has included additional degrees of freedom, such as selection heuristics, meta-heuristics, and initialisation heuristics in [20]. We here contribute by empirically measuring only the throughput (number of probes per second) of the invariant graph propagation algorithms (see Section 4). We assume that during search, the number of probes is typically one order of magnitude greater than the number of moves. Therefore, given this assumption, we measure the throughput in probes per second and not in moves per second (or a combination thereof), as the number of probes per second an invariant graph propagation algorithm can perform is of utmost importance.

3 Invariant Graph Propagation

Invariant graph propagation is a central concept of a CBLS solver, where an efficient data structure stores the current values of all variables. There are two main invariant graph propagation styles, here called *output-to-input* and *input-to-output*. The output-to-input style, used by iOpt [26] and Kangaroo [20], is a recursive approach that starts with the probed variables and ends with the search variables. Conversely, the input-to-output style, used by Localizer [24], Comet [23], Hexaly [4], InCELL [21], OscaR.cbls [6], Yuck [16], and Athanor [1, 2], is an iterative approach that starts with the search variables and ends with the probed variables.

We first describe a data structure (Section 3.1) and revisit dynamic cycles (Section 3.2). After the presentation of the two invariant graph propagation styles and their variations (Sections 3.3 and 3.4), we establish their time complexities (Section 3.5), and use them to recommend a heuristic for the selection of a propagation style and a variation (Section 3.6).

3.1 Valuations and Required Input Variables

During local search, the current value of *each* variable needs to be stored and possibly updated when a neighbour of the current assignment is probed or moved to. Recall that an assignment (and hence a neighbour) is only on the search variables.

Definition 3.1 (valuation)**.** Consider an invariant graph with variables $\mathcal{V}$. A *valuation* is a total function $\gamma\colon \mathcal{V} \to \mathbb{Z}$ that maps each variable to an integer value. We denote the value of variable x under valuation γ by $\gamma[x]$. A local-search solver initialises and iteratively updates a valuation, called the *current valuation*, which gives the *current value* of each variable.

Note that we now use the verb "to update" for a change of a variable within both assignments (see Definitions 2.1 and 2.4) and valuations, but whereas an update in an assignment definitely changes the value, an update in a

valuation might not actually change the value of the variable, as the new value can coincide with the previous one.

Consider an invariant graph with probed variables $\mathcal{P}$, a considered neighbour α' of the current assignment, and the current valuation γ. In order for a selection heuristic to assess each probed variable $p \in \mathcal{P}$ under γ, the value $\gamma[p]$ must become the value $\mathrm{v}_{\alpha'}(p)$. In general:

Definition 3.2 (correct valuation (on a variable set))**.** Consider an invariant graph with variables $\mathcal{V}$ and search variables $\mathcal{S} \subseteq \mathcal{V}$, an assignment α on $\mathcal{S}$, and a valuation γ on $\mathcal{V}$. We say that γ is *correct on* a variable subset $\mathcal{X} \subseteq \mathcal{V}$ with respect to α if $\gamma[x] = \mathrm{v}_\alpha(x)$ for each $x \in \mathcal{X}$. If $\mathcal{X} = \mathcal{V}$, then we just say that γ is *correct* with respect to α.

In [13, 25, 26, 20], if a variable x is correct with respect to the current assignment, then it is said to be *up-to-date*, otherwise x is said to be *out-of-date*. Note that our notion of a valuation γ being correct on variable x with respect to an assignment is more precise than the notion of x being up-to-date, as correctness is defined given a specific assignment (typically the current assignment or a neighbour thereof) while being up-to-date is defined without giving an explicit assignment.

We now extend the CBLS loop condition (Section 2.1) that the value of each variable is determinable under the current assignment. Consider an invariant graph with variables $\mathcal{V}$ and search variables $\mathcal{S} \subseteq \mathcal{V}$. Given the current assignment α on $\mathcal{S}$ and the current valuation γ on $\mathcal{V}$, the following *loop conditions* must hold at the start of each iteration:

LC1 the value of each variable in $\mathcal{V}$ is determined under α; and
LC2 γ is correct with respect to α.

Additionally, consider the probed variables $\mathcal{P} \subseteq \mathcal{V}$. For a CBLS solver to be correct, the following *CBLS solver conditions* are required:

- After the initialisation heuristic has created the initial version of α, the initial version of γ must be computed so that the loop conditions hold for the first iteration.
- Before a neighbour of the current assignment is probed, the loop conditions must hold, hence they are also loop conditions for the loop over neighbours *within* an iteration of the solver.
- After a neighbour α' of the current assignment was probed, the current valuation γ must be correct on $\mathcal{P}$ with respect to α', as the quality of α' is assessed by the selection heuristic based on the possibly new values of the probed variables. We describe at the end of this subsection the data structure for the current valuation in our implementation, which enables the zero-time undoing of a probe so as to re-establish the loop conditions for probing the next neighbour. Not requiring γ to be correct on all of $\mathcal{V}$ with respect to α' can result in faster invariant graph propagation when probing, as we shall see in Section 3.3.
- After a selected neighbour α' of the current assignment α is moved to, hence $\alpha := \alpha'$, the loop conditions must hold for the next iteration of the solver.

Both probing and moving are performed on the current valuation γ by what is called *invariant graph propagation*: for each invariant, the value $\gamma[y]$ of each output y might have to be updated if the value $\gamma[x]$ of some input x was updated. If any static input variable to some invariant $\mathbb{I}$ was updated, then the output variables of $\mathbb{I}$ have to be updated. However, if some dynamic input variable to some invariant $\mathbb{I}$ was updated, then the output variables of $\mathbb{I}$ might not have to be updated. As a motivating example, consider the array $\mathcal{X} = [x_1, \ldots, x_n]$ of variables; the variables i and y; the $\textsc{Element}(\mathcal{X}, i, y)$ invariant $\mathbb{I}$, which denotes $y \Longleftarrow \mathcal{X}[i]$; and the valuation γ. The input variable i is static and is required for determining the value of y under almost any valuation. However, only one dynamic input variable, namely $\mathcal{X}[\gamma[i]]$, is required under γ for determining the value of y under γ. We now define which input variables of an invariant actually need to be watched for detecting the need for such an update:

Definition 3.3 (required input variables (under a valuation))**.** Consider an invariant graph with variables $\mathcal{V}$ and search variables $\mathcal{S} \subseteq \mathcal{V}$; an assignment α on $\mathcal{S}$; a valuation γ on $\mathcal{V}$ that is correct with respect to α; and an invariant $\mathbb{I}$ with input variables $\mathcal{X} \subseteq \mathcal{V}$, static input variables $\mathcal{X}_\mathrm{s} \subseteq \mathcal{X}$, dynamic input variables $\mathcal{X}_\mathrm{d} = \mathcal{X} \setminus \mathcal{X}_\mathrm{s}$, and output variables $\mathcal{Y} \subseteq \mathcal{V}$. A dynamic input variable $d \in \mathcal{X}_\mathrm{d}$ is a *required input variable* to $\mathbb{I}$ *under* γ if there exists a neighbour assignment α' and a valuation γ' that is correct on $\mathcal{X} \cup \mathcal{Y}$ with respect to α' where γ' differs from γ only in variable d and at least one output, but not in the other inputs. Recall (from Section 2.7) that each static input variable is required under almost all assignments α for determining the value $\mathrm{v}_\alpha(y)$ of each output variable y. This property transposes to valuations: each static input variable $s \in \mathcal{X}_\mathrm{s}$ is a *required input variable* to $\mathbb{I}$, irrespective of γ.

The designer of a CBLS solver must implement a required-dynamic-input-variables$(\mathbb{I}, \gamma)$ function that returns the required dynamic variables to $\mathbb{I}$ under γ, with the precondition that γ is already correct on the static input variables to $\mathbb{I}$ with respect to the current assignment. For example, consider the array $\mathcal{X}$ of variables, the variables i and y, and a valuation γ that is correct on $\{i\}$: the call required-dynamic-input-variables$(\textsc{Element}(\mathcal{X}, i, y), \gamma)$ returns only the variable $\mathcal{X}[\gamma[i]]$. Indeed, each variable $x_j \in \mathcal{X}$ is a dynamic input variable and is furthermore a required input variable under γ if $\gamma[i] = j$. Note that the correct value $\gamma[i]$ of the *static* input variable i is thus needed in order to decide which of the dynamic input variables x_j is a required input variable under γ, and that the other dynamic input variables are *not* required.

Consider a valuation γ and an invariant $\mathbb{I}$ with input variables $\mathcal{X}$ and output variables $\mathcal{Y}$. We call the update of the values of the output variables $\mathcal{Y}$ for a subset $\mathcal{X}' \subseteq \mathcal{X}$ the *enforcing* of $\mathbb{I}$ under γ *given* $\mathcal{X}'$. If the running time to enforce $\mathbb{I}$ is not in terms of the cardinality of $\mathcal{X}'$, but in terms of the cardinality of $\mathcal{X}$ or some other quantity, then enforcing $\mathbb{I}$ at most once per probe or move is preferable to enforcing it multiple times. For example, consider the set $\mathcal{X}$ of variables, the variable y, the static $\textsc{Sum}(\mathcal{X}, y)$ invariant $\mathbb{I}$, and the valuation γ. An enforcing of $\mathbb{I}$ iterates over *all* variables in $\mathcal{X}$, updating $\gamma[y]$ to $\sum_{x \in \mathcal{X}} \gamma[x]$, which has a time complexity of $O(|\mathcal{X}|)$. It is therefore preferable to enforce $\mathbb{I}$ at most once per probe or move, after γ has been made correct on *all* of $\mathcal{X}$, rather than multiple times.

The enforcing of an invariant $\mathbb{I}$ can often be made *incremental* [24]: it only makes use of each required (static or dynamic) input variable x to $\mathbb{I}$ under the current valuation γ where the value of x has been updated under γ since the start of the probe or move. For example, a non-incremental enforcing of a $\textsc{Sum}(\mathcal{X}, y)$ invariant was shown in the previous paragraph; for a neighbour α' of the current assignment α, the set $\mathcal{X}'$ of updated variables in $\mathcal{X}$ between α and α', and the valuation γ that is correct on $\mathcal{X}$ with respect to α', an incremental enforcing of $\textsc{Sum}(\mathcal{X}, y)$ iterates over *only* the variables in $\mathcal{X}'$, updating $\gamma[y]$ to $\mathrm{v}_\alpha(y) + \sum_{x \in \mathcal{X}'} (\gamma[x] - \mathrm{v}_\alpha(x))$, which has a time complexity of $O(|\mathcal{X}'|)$. In the algorithms we present in Sections 3.3 and 3.4, the subset $\mathcal{X}'$ of updated variables is always a singleton, as we assume that the enforcing of most invariants is incremental, so that an invariant being enforced multiple times per probe or move does not cause a performance loss.

In our implementation, each probe and move has a distinct associated positive integer identifier, called its *timestamp* because it increases over time in our implementation (even though distinctness would be enough for our purposes). Our valuation data structure stores the following for each variable x:

- its value under the current assignment, denoted $\gamma_\mathrm{a}[x]$;
- its current value, denoted $\gamma[x]$ (as prescribed by Definition 3.1 above); and
- the timestamp when it was last updated, denoted $\gamma_\mathrm{t}[x]$.

Consider a probe or a move with timestamp τ of a neighbour of the current assignment α. When the value of variable x under γ is retrieved: if $\gamma_\mathrm{t}[x] = \tau$, then $\gamma[x]$ is returned, else $\gamma_\mathrm{a}[x]$ is returned. When the value of x under γ is updated to v: if it is a probe, then $\gamma_\mathrm{t}[x] \coloneqq \tau$ and $\gamma[x] \coloneqq v$, else (it is a move and) $\gamma_\mathrm{a}[x] \coloneqq v$. Since no operation is needed to undo the possibly multiple updates of the current value of each variable under γ after a

probe was made, the re-establishing of the loop condition LC2 on page 11 (making γ correct with respect to α) takes zero time before the next neighbour is probed, as that condition holds on the $\gamma_a[x]$, not on the $\gamma[x]$.

Another implementation of the valuation data structure would be to store the following for each variable:

- its value under the current assignment;
- its current value (as prescribed by Definition 3.1 above); and
- a Boolean that is **true** if and only if the variable has been updated during the current move or probe.

However, this implementation would take time linear in the number of variables before each move or probe in order to set each Boolean to **false**.

3.2 Determinable and Undeterminable Dynamic Cycles

At the end of Section 2.8, we showed that an invariant graph with cycles can be modified into an invariant graph without cycles, and we stated that dynamic cycles do not have to be handled this way. As a motivating example for this, consider again the invariant subgraph in Figure 5 an assignment α, and a valuation γ that is correct on $\{x_1, i_1, x_2, i_2\}$ with respect to α, with $\gamma[i_1] = 1$ and $\gamma[i_2] = 2$. For the dynamic cycle $\langle y_1, \mathbb{I}_2, y_2, \mathbb{I}_1, y_1\rangle$ the values $v_\alpha(y_1)$ and $v_\alpha(y_2)$ are determined, namely $v_\alpha(x_1)$ and $v_\alpha(x_2)$ respectively. There may thus be assignments such that the value of each variable on a dynamic cycle is guaranteed to be determined.

We now introduce the concept of a dynamic cycle being determinable or undeterminable under a valuation:

Definition 3.4 (determinable and undeterminable dynamic cycles under a valuation)**.** Consider an invariant graph with a dynamic cycle c and consider a valuation γ. If there exists at least one variable x in c, where x is both an output variable of a dynamic invariant $\mathbb{I}$ and transitively a required dynamic input variable to $\mathbb{I}$ under γ, then c is said to be an *undeterminable dynamic cycle* under γ; otherwise c is said to be a *determinable dynamic cycle* under γ.

In the motivating example above, the dynamic cycle $\langle y_1, \mathbb{I}_2, y_2, \mathbb{I}_1, y_1\rangle$ is determinable under γ since neither y_1 nor y_2 are transitively required dynamic input variables to $\mathbb{I}_1$ and $\mathbb{I}_2$ respectively under γ. Consider now an assignment α' and a valuation γ' that is correct on $\{x_1, i_1, x_2, i_2\}$ with respect to α', with $\gamma'[i_1] = 2$ and $\gamma'[i_2] = 1$. The dynamic cycle $\langle y_1, \mathbb{I}_2, y_2, \mathbb{I}_1, y_1\rangle$ is undeterminable under γ' since y_1 is transitively a required dynamic input variable to $\mathbb{I}_1$ under γ (and the same is true for y_2 and $\mathbb{I}_2$).

Consider an invariant graph with a single dynamic cycle c, an assignment α, the array $\mathcal{X} = [x_1, \ldots, x_n]$ of variables, the variables i and y, the Element$(\mathcal{X}, i, y)$ invariant $\mathbb{I}$, which denotes $y \Longleftarrow \mathcal{X}[i]$, as the only dynamic invariant in c, and the valuation γ that is correct on $\{i\}$ with respect to α. Note that since $\mathbb{I}$ is in c and y is the only output variable of $\mathbb{I}$, the edge $(\mathbb{I}, y)$ is in c. If c is undeterminable, then output y is transitively a required dynamic input to $\mathbb{I}$ under γ, and it is not guaranteed that $v_\alpha(y)$ is determined. Otherwise, cycle c is determinable and $v_\alpha(y)$ is guaranteed to be determined.

We now expand on this reasoning. Consider an invariant graph, an assignment α, and the valuation γ that is correct with respect to α. If the invariant graph contains no undeterminable dynamic cycles under γ, then the value of each variable is determined under α, and α satisfies the loop conditions (see Section 3.1) and is therefore eligible to be moved to.

3.3 Output-to-Input Propagation

Given a neighbour α' of the current assignment, the propagation of an invariant graph in the output-to-input style makes the current valuation correct on the *probed* variables with respect to α'. Hence, this style can only be used to probe α', but not for moving to α', as the latter requires the current valuation to be made correct (that is correct with respect to *all* the variables).

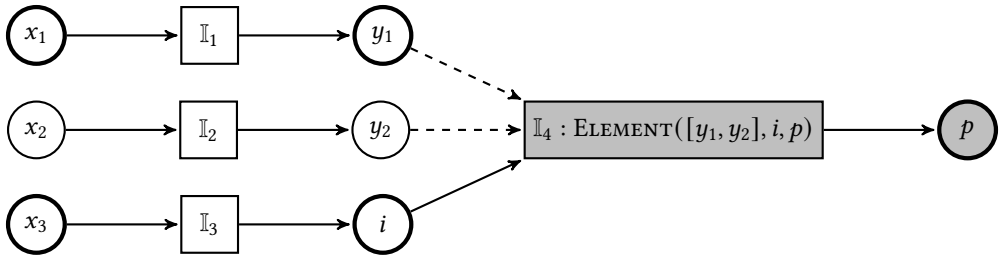

Fig. 6. An invariant graph with seven variables, $\{x_1, x_2, x_3, y_1, y_2, i, p\}$, three search variables, $\{x_1, x_2, x_3\}$, one probed variable, p, three static invariants, $\{\mathbb{I}_1, \mathbb{I}_2, \mathbb{I}_3\}$, and one dynamic invariant, $\mathbb{I}_4$. The variables that must be marked by a valid marking strategy are depicted with thick circles. The white nodes are in level 1, and the grey ones in level 2 (see Section 3.4.1).

Output-to-input propagation is called the *mark/sweep strategy* in iOpt [26] and the *mark-sweep approach* in Kangaroo [20]. We decided to call it the output-to-input invariant graph propagation style as the mark/sweep algorithm [7, 22, 13] is not required for it.

For the move to α', the input-to-output propagation style (see Section 3.4) or a computationally more expensive output-to-input algorithm (such as the one described in [13] or the one of Kangaroo) can be used.

We first describe how output-to-input propagation is sped up in practice and then explain the output-to-input propagation algorithm.

3.3.1 Marking. To speed up output-to-input propagation, some variables are typically marked by a marking strategy [7, 22, 13, 26, 20]. The invariant graph propagation algorithm (in Section 3.3) can only update the values of marked variables, so the set of marked variables must not be underestimated:

Definition 3.5 (valid marking strategy)**.** Consider an invariant graph with variables $\mathcal{V}$ and probed variables $\mathcal{P} \subseteq \mathcal{V}$, the set $\mathcal{I}_\mathcal{P}$ of invariants that have variables of $\mathcal{P}$ as outputs, and a neighbour α' of the current assignment α. A *valid marking strategy* marks each variable x that is updated between α and α' where either $x \in \mathcal{P}$ or x is transitively a required (static or dynamic) input variable to an invariant in $\mathcal{I}_\mathcal{P}$, under the valuation that is correct with respect to α'.

Note that this definition mentions the valuation that is correct with respect to α' and mentions the set of updated variables (whether search variables or not), but neither is known before propagation. Hence, the set of marked variables typically contains the actual set of updated variables. If an invariant depends on no updated search variables, then none of its input and output variables is updated, else its output variables are potentially updated.

When propagating an invariant graph in the output-to-input style, each non-marked variable is skipped. There is a running-time gain when skipping variables during propagating, but a running-time cost to perform both the marking and the check if a variable is marked or not.

For example, consider the invariant graph in Figure 6 (whose grey backgrounds of some nodes will be explained in Section 3.4.1). Consider a neighbour α' where x_1 and x_3 are the only updated search variables: their dependent variables y_1, i, and p are potentially updated. Using a valid marking strategy, the set of marked variables is at least $\{x_3, i, p\}$. However, since the value of i is not known until after propagation, variables x_1 and y_1 must also be marked because i could take the value 1; if i takes the value 2, then the marking of x_1 and y_1 was superfluous.

We distinguish three valid marking strategies:

- *ad-hoc marking*: each variable is marked if it is, or depends on, an updated search variable of the probe; marking before a propagation takes non-constant time by depth-first search, starting at the updated search

variables (see Theorem 3.4 in Section 3.5); checking during a propagation if a visited variable is marked takes constant time;
- *prepared marking*: each variable x is considered marked if the intersection between the subset $\mathcal{S}_x$ of the search variables that x depends on and the set $\mathcal{S}_\mathrm{u}$ of updated search variables of the probe is non-empty; each set $\mathcal{S}_x$ is computed and stored before search; checking during a propagation if a visited variable is marked, by checking if the intersection with $\mathcal{S}_\mathrm{u}$ is non-empty, takes non-constant time;
- *total marking*: each variable is considered marked, hence both marking before a propagation and checking during a propagation whether a visited variable is marked or not takes zero time. However, as all variables are considered marked, no variables will be skipped during propagation.

For example, in the invariant graph of Figure 6, the set of marked variables is $\{x_1, y_1, x_3, i, p\}$ under ad-hoc and prepared marking, and all variables are marked under total marking.

For any invariant graph, the same variables are marked by the first two marking strategies. Their difference is the trade-off in running time for a particular probe: ad-hoc marking is better for checking if a variable is marked, while prepared marking is better for marking the variables, amortising its effort of preparing the marking before search starts. In our experiments (see Section 4) we examine this trade-off, considering that preparation happens only once but probing extremely often.

3.3.2 Output-to-Input Propagation Algorithm (only for Probing). In output-to-input propagation, the invariant graph is propagated in depth-first order in the reverse direction of the edges in the invariant graph: each invariant is enforced after all invariants that have its input variables as outputs have recursively been enforced [13, 25, 26, 20].

Consider an invariant graph with the variables $\mathcal{V}$, search variables $\mathcal{S} \subseteq \mathcal{V}$, and probed variables $\mathcal{P} \subseteq \mathcal{V}$. Consider a neighbour α' of the current assignment α, and let the current valuation γ be correct with respect to α. The procedure of probing α' in output-to-input style is given in Algorithm 1, where:

- The function update-valuation(γ, x, v) sets the value of variable x to $v \in \mathbb{Z}$ under valuation γ, that is: $\gamma[x] \coloneqq v$.
- The function is-marked(y) returns **true** if variable y is marked by the marking strategy, which we assume is valid, else returns **false**. Checking if a variable is marked takes constant time for ad-hoc marking, non-constant time for prepared marking, and zero time for total marking.
- The function defining-invariant(y) has the precondition $y \notin \mathcal{S}$ and returns the invariant that has output variable y.
- The function static-input-variables$(\mathbb{I})$ returns the set of static input variables to invariant $\mathbb{I}$.
- The function required-dynamic-input-variables$(\mathbb{I}, \gamma)$, already mentioned in Section 3.1, returns the set of required dynamic input variables to invariant $\mathbb{I}$ under valuation γ, with the precondition that γ is correct on the static input variables to $\mathbb{I}$.
- The function has-changed(x, γ) returns **true** if the value that variable x took at the start, that is $\mathrm{v}_\alpha(x)$, does not equal $\gamma[x]$, else returns **false**. In our implementation, for a probe with associated timestamp τ, this means that the truth value of $\gamma_\mathrm{t}[x] = \tau \wedge \gamma_\mathrm{a}[x] \neq \gamma[x]$ is returned.
- The function enforce$(\mathbb{I}, x, \gamma)$ enforces invariant $\mathbb{I}$ under γ given its input variable x.
- The function output-variables$(\mathbb{I})$ returns the set of output variables of invariant $\mathbb{I}$.

First, the current valuation γ is made correct on $\mathcal{S}$ with respect to α', lines 2–3. The set $\mathcal{W}$ of visited variables, on which γ is correct with respect to α', is then initialised to $\mathcal{S}$, line 4. The set C of variables that are currently recursed on is initialised to $\varnothing$, line 5. The variables in $\mathcal{P}$ are iterated over, lines 6–7, calling the help procedure output-to-input′ defined in lines 8–23. The latter takes a variable y as input, line 8. If y is currently recursed on, that is another call output-to-input′(y) is currently being executed, then first output-to-input′(y) and

Algorithm 1: The propagation of an invariant graph in output-to-input style.

Data: $\mathcal{V}$ is the set of variables; $\mathcal{S} \subseteq \mathcal{V}$ is the set of search variables; α is the current assignment; γ is the valuation that is correct with respect to α; $\mathcal{P} \subseteq \mathcal{V}$ is the set of probed variables; α' is a neighbour of α.
Precondition: The invariant graph has no static cycles.
Postcondition: If the procedure does not abort, then the valuation γ is correct on $\mathcal{P}$ with respect to α', else the invariant graph contains an undeterminable dynamic cycle under γ.

```
1  procedure output-to-input(α′):
2      foreach x ∈ S do
3          update-valuation(γ, x, α′(x))
4      W := S                  // data invariant: valuation γ is correct on W with respect to α′
5      C := ∅                  // data invariant: output-to-input′(y) is currently running for all y ∈ C
6      foreach p ∈ P do
7          output-to-input′(p)
8      procedure output-to-input′(y):
9          if y ∈ C then
10             abort
11         else if is-marked(y) ∧ y ∉ W then
12             C := C ∪ {y}
13             I := defining-invariant(y)
14             foreach x ∈ static-input-variables(I) do
15                 output-to-input′(x)                    // x ∈ W after recursing
16                 if has-changed(x, γ) then
17                     enforce(I, x, γ)
18             foreach x ∈ required-dynamic-input-variables(I, γ) do
19                 output-to-input′(x)                    // x ∈ W after recursing
20                 if has-changed(x, γ) then
21                     enforce(I, x, γ)
22             W := W ∪ output-variables(I)
23             C := C \ {y}
```

then output-to-input(α') abort because the invariant graph thus has an undeterminable dynamic cycle under γ, lines 9–10. Else if y is marked and not visited yet, then the procedure continues, else no further operations are performed, line 11. Variable y is added to the variables that are recursed on currently, line 12. The invariant $\mathbb{I}$ that defines y is retrieved, line 13. The static input variables to $\mathbb{I}$ are iterated over, line 14: the help procedure recurses on each such x, line 15; if the value that x took at the start now does not equal its current value under γ, line 16, then x is used to enforce $\mathbb{I}$, possibly updating the values that the output variables of $\mathbb{I}$ take under γ, line 17. The required dynamic input variables to $\mathbb{I}$ under γ are then iterated over, line 18: the help procedure recurses on each such x, line 19; if the value that x took at the start now does not equal its current value under γ, line 20, then x is used to enforce $\mathbb{I}$, possibly updating the values that the output variables of $\mathbb{I}$ take under γ, line 21. Note that the

two iterations cannot be merged: recall from the example after Definition 3.3 that the values of the static input variables to a dynamic invariant are needed in order to identify its required dynamic input variables.

Note that $\mathbb{I}$ may be enforced multiple times if it has multiple updated input variables. As enforcing an invariant updates each of its output variables, each output variable of $\mathbb{I}$ has now been visited, line 22. Variable y is removed from the variables that are currently recursed on, line 23.

If Algorithm 1 aborts, then the invariant graph contains an undeterminable dynamic cycle under γ and α' is not fully probed and thus not to be considered for a move.

Note that for the algorithms given in [13, 20], an invariant $\mathbb{I}$, or a node in [13] that here corresponds to an invariant node, can be *lazy*, allowing recursion only on a strict subset of the (static and dynamic) input variables of $\mathbb{I}$. Each dynamic invariant can be implemented as a lazy invariant, but not vice-versa. For example, given the variable set $\mathcal{X}$ and the variable y, the Product$(\mathcal{X}, y)$ invariant functionally defines the output variable y to be the product of the input variables x of $\mathcal{X}$, denoted by $y \Longleftarrow \prod_{x \in \mathcal{X}} x$. When performing propagation in output-to-input style, if some (static) input variable x in $\mathcal{X}$ takes value 0, then the value of each other variable in $\mathcal{X}$ is irrelevant, as the value of y cannot change. Algorithm 1 can be extended to allow for lazy invariants by stopping iteration prematurely on lines 14–17 and 18–21 if (i) the current defining invariant $\mathbb{I}$ is lazy and (ii) the value of each output variable of $\mathbb{I}$ cannot change given the values of the input variables that have already been iterated over.

We now prove that Algorithm 1 is correct when it does not abort.

Theorem 3.1 (correctness of Algorithm 1)**.** Consider an invariant graph $\mathcal{G}$ with the variables $\mathcal{V}$, search variables $\mathcal{S} \subseteq \mathcal{V}$, and probed variables $\mathcal{P} \subseteq \mathcal{V}$. Consider a current assignment α, a neighbour α' of α, and the valuation γ that is correct with respect to α. Assume a valid marking strategy has been used and that the help procedure output-to-input′ does not abort during the execution of output-to-input(α'). The updated valuation γ upon running output-to-input(α') is correct on $\mathcal{P}$ with respect to α'.

Proof. Given the set $\mathcal{I}_{\mathcal{P}}$ of invariants that have the probed variables as outputs, let $\mathcal{R}_{\mathcal{P}} \supseteq \mathcal{P}$ be the set of probed variables and the variables that are transitively required (static or dynamic) input variables to some invariant in $\mathcal{I}_{\mathcal{P}}$ under the valuation that is correct with respect to α'.

We first prove that procedure output-to-input′ has a finite number of executions. We then prove that if procedure output-to-input′(y) finishes for some variable y without aborting, then γ is correct on $\{y\}$ with respect to α. Finally, we prove that if Algorithm 1 finishes without aborting, then the valuation γ is correct on $\mathcal{P} \cup \mathcal{S} \cup \mathcal{R}_{\mathcal{P}}$ with respect to α' and hence correct on $\mathcal{P}$ with respect to α', as required.

Finite number of executions: Up until and including line 4, the valuation γ is made correct on $\mathcal{S}$ with respect to α' and $\mathcal{W}$ is initialised to $\mathcal{S}$, so the data invariant of $\mathcal{W}$ holds.

For any search variable $y \in \mathcal{R}_{\mathcal{P}} \cap \mathcal{S}$, the recursive help procedure output-to-input′(y) will not enter the **then** clause at line 11 and will thus perform no more operations (as γ is already correct on $\mathcal{S}$ with respect to α').

For any non-marked variable $y \in \mathcal{R}_{\mathcal{P}}$, the execution of output-to-input′(y) will not enter the **then** clause at line 11. Since the marking strategy is assumed valid, the valuation γ is correct on all non-marked variables with respect to α'.

Assume that the data invariant of $\mathcal{W}$ holds and consider an invariant $\mathbb{I}_y$ that has an output variable $y \in \mathcal{R}_{\mathcal{P}}$, the static input variables $\mathcal{X}_{\mathrm{s}}$, and the required dynamic input variables $\mathcal{X}_{\mathrm{d}}$ under the valuation that is correct with respect to α', where each required (static or dynamic) input variable in $\mathcal{X}_{\mathrm{s}} \cup \mathcal{X}_{\mathrm{d}}$ is either not marked or already visited (that is $\mathcal{W} \supseteq \mathcal{X}_{\mathrm{s}} \cup \mathcal{X}_{\mathrm{d}}$). Since we assume that the data invariant of $\mathcal{W}$ holds, the valuation γ is correct on $\mathcal{X}_{\mathrm{s}} \cup \mathcal{X}_{\mathrm{d}}$ with respect to α'. Therefore, the set of required dynamic input variables to $\mathbb{I}_y$ under γ is $\mathcal{X}_{\mathrm{d}}$. During the execution of output-to-input′(y), after recursing on each required (static or dynamic) input variable to $\mathbb{I}_y$, both irrespective of γ and under γ, lines 14–21, the invariant $\mathbb{I}_y$ has been enforced for each required (static or

dynamic) input variable $x \in \mathcal{X}_s \cup \mathcal{X}_d$, and thus γ has been made correct on $\{y\}$ with respect to α'. After execution of output-to-input$'(y)$ finishes, we have $y \in \mathcal{W}$.

Consider the current execution of output-to-input$'(y)$ for $y \in \mathcal{R}_\mathcal{P}$ where y is marked and $y \notin \mathcal{W}$. If y is already being recursed on, then $y \in C$ and the **then** clause at line 9 will be entered, which contradicts our assumption. So, during the execution of any output-to-input$'(y)$, there cannot be any recursive calls to output-to-input$'(y)$. Since (i) graph $\mathcal{G}$ has a finite number of nodes and edges, and since (ii) during the execution of output-to-input$'(y)$ for any $y \in \mathcal{R}_\mathcal{P}$, there are no other recursive calls to output-to-input$'(y)$, there is a finite number of calls to output-to-input$'$. Thus, procedure output-to-input$'$ has a finite number of executions.

Procedure output-to-input$'$ *finishes without aborting:* Consider the marked variable $y \in \mathcal{R}_\mathcal{P} \setminus \mathcal{S}$ that is an output of invariant $\mathbb{I}_y$ and consider the execution of output-to-input$'(y)$. When that execution finishes, for each variable x that is transitively a required (static or dynamic) input variable to $\mathbb{I}_y$ under the valuation that is correct with respect to α' the execution of output-to-input$'(x)$ has also finished.

The execution of output-to-input$'(y)$ for some $y \in \mathcal{R}_\mathcal{P}$ is the first call to output-to-input$'$ to finish. If $y \in \mathcal{W}$ or y is not marked, then γ is already correct on $\{y\}$ with respect to α'. Otherwise, $y \notin \mathcal{S}$ and y is an output of some invariant $\mathbb{I}_y$, each required (static or dynamic) input variable x to $\mathbb{I}_y$ under the valuation that is correct with respect to α' must be either a search variable or not marked. Otherwise, some other invariant $\mathbb{I}_x$ would have x as an output and the execution of output-to-input$'(x)$ would have finished before output-to-input$'(y)$, which is a contradiction. Therefore, γ is correct on each required (static or dynamic) input variable to $\mathbb{I}_y$ under the valuation that is correct with respect to α'. Since the data invariant of $\mathcal{W}$ holds before the execution of output-to-input$'(y)$, after execution of output-to-input$'(y)$ finishes, we have that γ is also correct on the output variables of $\mathbb{I}_y$ with respect to α', the set $\mathcal{W}$ has each output variable of $\mathbb{I}_y$, and the data invariant of $\mathcal{W}$ holds.

Consider each subsequent finished execution of the procedure output-to-input$'(y)$ for a variable $y \in \mathcal{P} \setminus \mathcal{S}$, where invariant $\mathbb{I}_y$ has y as an output and required (static or dynamic) input variables $\mathcal{X}$ under the valuation that is correct with respect to α'. When execution of output-to-input$'(y)$ finishes, for each required (static or dynamic) input variable $x \in \mathcal{X}$, either $x \in \mathcal{W}$ or x is not marked. Otherwise, $x \notin \mathcal{W}$ and x is marked, thus some execution of output-to-input$'(x)$ has not finished, which is a contradiction. So, when output-to-input$'(y)$ finishes, the valuation γ is correct on $\mathcal{X}$ with respect to α'. The set $\mathcal{W}$ of visited nodes has each search variable and each variable $x \in \mathcal{R}_\mathcal{P}$ where procedure output-to-input$'(x)$ has finished before output-to-input$'(y)$. The valuation γ is correct with respect to α' on (i) each variable $x \in \mathcal{R}_\mathcal{P}$ that is not marked, (ii) each variable $x \in \mathcal{R}_\mathcal{P}$ where procedure output-to-input$'(x)$ has finished before output-to-input$'(y)$ has finished, and (iii) the search variables. Therefore, the data invariant of $\mathcal{W}$ holds. After execution of output-to-input$'(y)$ finishes, the valuation γ is also correct on the output variables of $\mathbb{I}_y$ with respect to α'; the set $\mathcal{W}$ of visited nodes has each output variable of $\mathbb{I}_y$, as γ is correct on each output variable of $\mathbb{I}_y$; and the data invariant of $\mathcal{W}$ holds. Therefore, given any variable $y \in \mathcal{V}$, if the procedure output-to-input$'(y)$ finishes without aborting, then γ is correct on $\{y\}$ with respect to α'.

If Algorithm 1 finishes without aborting: Since output-to-input$'$ is executed iteratively for each probed variable, after output-to-input(α') finishes the data invariant of $\mathcal{W}$ holds and valuation γ is correct on $\mathcal{R}_\mathcal{P} \supseteq \mathcal{P}$ with respect to α'. Therefore, if output-to-input$'$ finishes without aborting, then γ is correct on $\mathcal{P} \cup \mathcal{S} \cup \mathcal{P}_\mathcal{R}$ with respect to α'.

□

3.4 Input-to-Output Propagation

Given a neighbour α' of the current assignment, the propagation of an invariant graph in the input-to-output style makes the current evaluation correct on *all* variables with respect to α'. Hence, this style can be used for both probing α' and moving to α'.

The input-to-output propagation style is called the *topological ordering strategy* in iOpt [26] and is also used in Localizer [17], Comet [23], Hexaly [4], and OscaR.cbls [6]. We decided to call it the input-to-output invariant graph propagation style so as to make its name consistent with our name of the output-to-input propagation style.

We first show how the variables and invariants of the invariant graph are partitioned into levels before search starts in order to help detect undeterminable dynamic cycles during invariant graph propagation. We then show how the variables and invariants in each level are topologically sorted during each probe or move in order both to actually detect undeterminable dynamic cycles and to make input-to-output propagation more efficient. Finally, we give an input-to-output propagation algorithm that can perform both probes and moves. The algorithm we give requires that any undeterminable dynamic cycle is detected during invariant graph propagation. However, the detection of undeterminable dynamic cycles could theoretically be done using some method besides the partition and topological sort of the invariant graph.

3.4.1 Partitioning the Invariant Graph into Levels. When an invariant graph is propagated in the input-to-output style, the nodes of the invariant graph are placed into levels before search starts [17], in order to help detect undeterminable dynamic cycles during propagation and make the propagation efficient.

Consider an invariant graph $\mathcal{G}$; an assignment α; and a valuation γ that is to be made correct on α, where $\mathcal{G}$ contains no static cycles; for each dynamic cycle c in $\mathcal{G}$, each edge in c to a dynamic invariant $\mathbb{I}$ originates from a dynamic input variable to $\mathbb{I}$; and $\mathcal{G}$ is propagated in the input-to-output style. The (dynamic) cycles of $\mathcal{G}$ induce one or more strongly connected components. We partition the nodes of $\mathcal{G}$ into levels, such that all nodes of each strongly connected component are in the same level. For each dynamic invariant $\mathbb{I}$, we place each static input to $\mathbb{I}$ into a shallower level than the level that $\mathbb{I}$ is in. When an invariant graph is propagated in the input-to-output style, we propagate the levels from shallowest to deepest. Therefore, before some level ℓ that contains some strongly connected component is to be propagated (and after each level shallower than ℓ has been propagated), for each variable x in a level shallower than ℓ the value $\gamma[x]$ is correct with respect to α. Since each static input variable to each dynamic invariant $\mathbb{I}$ is in a shallower level than $\mathbb{I}$, the required (static and dynamic) input variables to each invariant in ℓ under γ are known. If the nodes in ℓ and the edges that correspond to the required (static and dynamic) input variables to the invariants in ℓ induce one or more cycles, then those cycles are undeterminable dynamic cycles and α' cannot be moved to or probed.

Each dynamic invariant $\mathbb{I}$ must thus be in a deeper level than each of its static input variables, since the values of the static input variables are needed to identify the required dynamic input variables of $\mathbb{I}$. Given an invariant graph, the level of each node is defined by a mathematical constraint, as defined next:

Definition 3.6 (level constraints)**.** The *level* of an invariant $\mathbb{I}$ with the set $\mathcal{X}_\mathrm{s}$ of static input variables and the set $\mathcal{X}_\mathrm{d}$ of dynamic input variables is mathematically constrained as follows:

$$\mathrm{lv}(\mathbb{I}) = \begin{cases} \max(\{\mathrm{lv}(x) \mid x \in \mathcal{X}_\mathrm{s}\}) & \text{if } \mathcal{X}_\mathrm{d} = \varnothing \\ \max(\{\mathrm{lv}(x) + 1 \mid x \in \mathcal{X}_\mathrm{s}\} \cup \{\mathrm{lv}(x) \mid x \in \mathcal{X}_\mathrm{d}\}) & \text{otherwise} \end{cases}$$

where the *level* of a variable x is mathematically constrained as follows:

$$\mathrm{lv}(x) = \begin{cases} \mathrm{lv}(\mathbb{I}) & \text{if } x \text{ is an output of the invariant } \mathbb{I} \\ 1 & \text{otherwise (and hence } x \text{ is a search variable)} \end{cases}$$

The level constraints defined here correspond to the level constraints in [17].

For example, in the invariant graph of Figure 2, all variables and invariants are in level 1, because there are no dynamic invariants and hence no dynamic input variables, and therefore the deepest level of any node is 1. Also, in the invariant subgraph of Figure 5, the variables $\{x_1, x_2, i_1, i_2\}$, coloured white, are in level 1, while the variables $\{y_1, y_2\}$ and dynamic invariants $\{\mathbb{I}_1, \mathbb{I}_2\}$, all coloured grey, are in level 2, because i_1 and i_2 are static input variables to the dynamic invariants $\mathbb{I}_1$ and $\mathbb{I}_2$ respectively, with both variables in level 1. Further, in the invariant graph of Figure 6, the variables and static invariants $\{x_1, x_2, x_3, \mathbb{I}_1, \mathbb{I}_2, \mathbb{I}_3, y_1, y_2, i\}$, coloured white, are all in level 1, while the probed variable p and the dynamic invariant $\mathbb{I}_4$, all coloured grey, are in level 2, because the static input variable i to the dynamic invariant $\mathbb{I}_4$ is in level 1.

A solution to the level constraints of the nodes induces a partitioning of the invariant graph, as each node is in exactly one level in any solution.

We now prove a necessary and sufficient condition for the satisfiability of the level constraints:

Lemma 3.1 (satisfiable level constraints)**.** Consider an invariant graph with nodes $\mathcal{N}$. The level constraint of each node in $\mathcal{N}$ is satisfiable if and only if the invariant graph contains no cycle with an edge from a static input to a dynamic invariant.

Proof. We first prove that if the invariant graph contains a cycle with an edge from an input variable to a dynamic invariant, then the level constraints are unsatisfiable. We then prove that if the level constraints are satisfiable, then the invariant graph contains no cycle with an edge from a static input variable to a dynamic invariant.

Assume that the invariant contains a cycle c, with $(x, \mathbb{I}) \in c$, where x is a static input to dynamic invariant $\mathbb{I}$, we prove that the level constraints are unsatisfiable. Since x and $\mathbb{I}$ are in a cycle, variable x depends on $\mathbb{I}$ (and vice versa). Additionally, since the level of each node is at least the level of each node it depends on and the level of x is greater than $\mathbb{I}$, we have $\text{lv}(x) > \text{lv}(\mathbb{I})$ and $\text{lv}(\mathbb{I}) \geq \text{lv}(x)$, which is unsatisfiable.

Conversely, assume that the level constraints of the nodes are satisfiable, we prove by contradiction that the invariant graph contains no cycle with an edge from a static input to a dynamic invariant. For any input x to invariant $\mathbb{I}$, we have $\text{lv}(x) \leq \text{lv}(\mathbb{I})$, which is satisfiable. If an output variable y of $\mathbb{I}$ depends on $\mathbb{I}$, then there exists some cycle c containing edges $(x, \mathbb{I})$ and $(\mathbb{I}, y)$. Additionally, since the level of each node is at least the level of each node it depends on and the level constraints are satisfiable, $\text{lv}(\mathbb{I}) \leq \text{lv}(x)$ is satisfiable. Thus, $\text{lv}(x) \leq \text{lv}(\mathbb{I})$ and $\text{lv}(\mathbb{I}) \leq \text{lv}(y)$ are satisfiable, making $\text{lv}(x) = \text{lv}(\mathbb{I})$ satisfiable. Therefore, by Definition 3.6, invariant $\mathbb{I}$ cannot be dynamic, as $\text{lv}(x) > \text{lv}(\mathbb{I})$ is not satisfiable, which is a contradiction.

Thus, the level constraints are satisfiable if and only if the invariant graph contains no cycle with an edge from a static input variable to a dynamic invariant. □

The levels are static throughout search. Additionally, if an algorithm, such as the one in Section 3.4.3 below, has the precondition that the level constraints of an invariant graph must be satisfiable, then it by extension has the precondition that the invariant graph contains no cycle that contains an edge from a static input variable to a dynamic invariant. This precondition is implicit in Localizer [17], Comet [23], and Oscar.cbls [6], but explicit in the algorithm in Section 3.4.3 below.

An example of an invariant subgraph that contains an edge from a static input variable to a dynamic invariant is depicted in Figure 7 (whose grey and white backgrounds of the nodes will be explained later in this section). In order to partition this subgraph into levels, the dynamic invariant $\mathbb{I}_2$ must be replaced by its static counterpart: the variables $\{x_1, x_2, x_3, i\}$, coloured white, are then in level 1, while the output variables $\{y_1, y_2\}$, the dynamic invariant $\mathbb{I}_1$, and the now static invariant $\mathbb{I}_2$, all coloured grey, are then in level 2.

In practice, we have not observed any invariant graph containing a cycle with an edge from a static input variable to a dynamic invariant. Therefore, we do not know how prevalent such invariant graphs are.

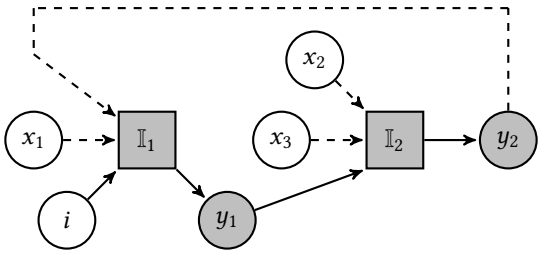

Fig. 7. A cyclic invariant subgraph with six variables, $\{x_1, x_2, x_3, i, y_1, y_2\}$; two dynamic invariants, $\{\mathbb{I}_1, \mathbb{I}_2\}$; two static input variables, $\{i, y_1\}$, with outgoing solid edges; four dynamic input variables, $\{x_1, x_2, x_3, y_2\}$, with outgoing dashed edges; and two output variables, $\{y_1, y_2\}$. The cycle $\langle y_1, \mathbb{I}_2, y_2, \mathbb{I}_1, y_1 \rangle$ is dynamic. This is a modified version of the subgraph in Figure 5 where y_1 has been made a static input variable to $\mathbb{I}_2$, the variable i_2 has been removed, the variable i_1 has been renamed into i, and the variable x_3 has been added. The edge $(y_1, \mathbb{I}_2)$ in the cycle is from a static input variable to a dynamic invariant. The white nodes are in level 1, and the grey ones in level 2 (see Section 3.4.1).

Consider an invariant graph with cycles C. In order to make the level constraints satisfiable, for each dynamic cycle $c \in C$ and each edge $(x, \mathbb{I}) \in c$, where x is a static input variable to the dynamic invariant $\mathbb{I}$, the latter must be made static by the modeller or solver by replacing it with its static counterpart (see Lemma 3.1 and Section 2.7).

Note that Algorithm 1 of the previous section detects undeterminable dynamic cycles by aborting when recursing on some variable that is already being recursed on. Thus, Algorithm 1 does not require the invariant graph to be partitioned into levels and by extension allows the invariant graph to contain cycles with edges from static input variables to dynamic invariants.

Since we only reason on invariant graphs without cycles that contain an edge from a static input variable to a dynamic invariant and since each variable is an output of at most one invariant, the following assertions hold:

(i) the level constraints of all nodes of the invariant graph are satisfiable; this follows from the invariant graph having no cycles that contain an edge from a static input variable to a dynamic invariant;
(ii) the levels partition the nodes of the invariant graph; this follows from (i);
(iii) given any pair of variables x and y, where y depends on x, we have that y is in the same level as x or in a deeper level than x, thus $\text{lv}(y) \geq \text{lv}(x)$; this follows from ($i$) and Definition 3.6; and
(iv) given a static input variable x to a dynamic invariant $\mathbb{I}$, we have that $\mathbb{I}$ is in a deeper level than x, thus $\text{lv}(\mathbb{I}) > \text{lv}(x)$; this also follows from (i) and Definition 3.6.

We now show that every cycle in an invariant graph is local to a single level:

Lemma 3.2 (each cycle is local to a level)**.** All nodes of each cycle in a cyclic invariant graph are in the same level.

Proof. The level of each invariant $\mathbb{I}$ in a graph is at least the level of each node that $\mathbb{I}$ depends on. Since each node in a cycle depends on all other nodes in that cycle and since the level constraints of all nodes are satisfiable by assertion (i), the levels of all nodes on the cycle are equal. □

Given assertions (i) to (iv), an invariant graph $\mathcal{G}$ can be partitioned into levels such that the level constraint of each node in $\mathcal{G}$ is satisfiable and each node in $\mathcal{G}$ only depends on nodes on the same or shallower levels. Additionally, given Lemma 3.2, each cycle in $\mathcal{G}$ is local to a single level. Therefore, each level ℓ in $\mathcal{G}$ can be propagated after each level shallower than ℓ has been propagated, and independently of the propagation of any level deeper than ℓ.

For each dynamic invariant $\mathbb{I}$, we now show that its set of required dynamic input variables is known before level $\text{lv}(\mathbb{I})$ is propagated, by the exploitation of assertion (iv). Consider an invariant graph with the variables $\mathcal{V}$;

a level $\ell > 1$; the set $\mathcal{L} = \{x \mid x \in \mathcal{V} \wedge \mathrm{lv}(x) < \ell\}$ of all variables in levels shallower than ℓ; an assignment α; the valuation γ that is correct on $\mathcal{L}$ with respect to α; and a dynamic invariant $\mathbb{I}$ with static input variables $\mathcal{X}_\mathrm{s}$, dynamic input variables $\mathcal{X}_\mathrm{d}$, and level $\mathrm{lv}(\mathbb{I}) = \ell$. Since assertion (iv) implies that the level of each variable in $\mathcal{X}_\mathrm{s}$ is shallower than ℓ, the valuation γ is already correct on $\mathcal{X}_\mathrm{s} \subseteq \mathcal{L}$ with respect to α. For any dynamic invariant $\mathbb{I}$, its set of required dynamic input variables under γ is given by the values of the static input variables under γ. Thus, the subset of the required dynamic input variables to $\mathbb{I}$ under γ is known because $\mathcal{X}_\mathrm{s} \subseteq \mathcal{L}$. Since each static input variable $x \in \mathcal{X}_\mathrm{s}$ is in a shallower level than ℓ, the value of x under γ will not be updated during propagation of any level deeper than $\mathrm{lv}(x)$, with $\ell > \mathrm{lv}(x)$.

For example, consider the Element$(\mathcal{X}, i, y)$ invariant $\mathbb{I}$, which is denoted $y \Longleftarrow \mathcal{X}[i]$ for the array $\mathcal{X} = [x_1, \ldots, x_n]$ of dynamic input variables, the static input variable i, and the output variable y. Since input-to-output propagation is performed by increasing levels and the levels of both $\mathbb{I}$ and y are deeper than the level of the static input variable i, the current valuation γ will be correct on $\{i\}$ before $\mathbb{I}$ is enforced. Therefore, the only required dynamic input variable to $\mathbb{I}$, namely $\mathcal{X}[\gamma[i]]$, is known when $\mathbb{I}$ is enforced during propagation of level $\mathrm{lv}(\mathbb{I})$.

3.4.2 Topologically Sorting each Level and Detecting Undeterminable Dynamic Cycles. Consider an invariant graph $\mathcal{G}$ with invariants $\mathcal{I}$ where the level constraints of all nodes are satisfiable, a level ℓ, an assignment α, and the valuation γ that is correct on all variables in levels shallower than ℓ with respect to α. For each invariant $\mathbb{I}$, only the values of the required (static or dynamic) variables under γ are needed when $\mathbb{I}$ is enforced. Given assertions (i) to (iv) from Section 3.4.1, Lemma 3.2, and that γ is correct on each variable in levels shallower than ℓ with respect to α, the required (static or dynamic) input variables to each invariant in level ℓ under γ will not change.

Consider also the invariant subgraph $\mathcal{G}_{\ell,\gamma}$ with the invariants $\mathcal{I}_{\ell,\gamma} = \{\mathbb{I} \mid \mathbb{I} \in \mathcal{I} \wedge \mathrm{lv}(\mathbb{I}) = \ell\}$ and the variables $\mathcal{V}_{\ell,\gamma}$, where each variable $x \in \mathcal{V}_{\ell,\gamma}$ is either an output variable of an invariant in $\mathcal{I}_{\ell,\gamma}$ or a required (static or dynamic) input variable to an invariant in $\mathcal{I}_{\ell,\gamma}$ under γ, where $\mathcal{G}_{\ell,\gamma}$ only contains the edges from each invariant $\mathbb{I} \in \mathcal{I}_{\ell,\gamma}$ to its output variables and the edges to $\mathbb{I}$ from its required (static or dynamic) input variables under γ. If $\mathcal{G}_{\ell,\gamma}$ is acyclic, then its nodes can be topologically sorted, otherwise $\mathcal{G}_{\ell,\gamma}$ contains one or more cycles and its nodes cannot be topologically sorted.

Therefore, topologically sorting the invariant subgraph $\mathcal{G}_{\ell,\gamma}$ becomes a satisfaction problem, which is satisfiable if and only if there are no static cycles or undeterminable cycles in level ℓ under γ. Additionally, a solution to the satisfaction problem induces a topological ordering of the nodes in level ℓ.

We now define the mathematical constraints on what we call the topological number of each node in an invariant graph, before we prove that if an invariant graph contains a static or an undeterminable dynamic cycle, then the constraints of some nodes are unsatisfiable. Finally, we show that if the topological-number constraints of the nodes are satisfiable, then they induce a topological ordering of the nodes of the invariant graph.

Definition 3.7 (topological-number constraints)**.** Consider a valuation γ. The *topological number* of an invariant $\mathbb{I}$ with the set $\mathcal{X}_\mathrm{s}$ of static input variables and the set $\mathcal{X}_{\mathrm{d},\gamma}$ of required dynamic input variables under γ is mathematically constrained as follows:

$$\mathrm{t}_\gamma(\mathbb{I}) = 1 + \max\left\{\mathrm{t}_\gamma(x) \mid x \in \mathcal{X}_\mathrm{s} \cup \mathcal{X}_{\mathrm{d},\gamma}\right\}$$

where the *topological number* of a variable x is mathematically constrained as follows:

$$\mathrm{t}_\gamma(x) = \begin{cases} \mathrm{t}_\gamma(\mathbb{I}) & \text{if } x \text{ is an output of the invariant } \mathbb{I} \\ 0 & \text{otherwise (and hence } x \text{ is a search variable)} \end{cases}$$

The topological-number constraints defined here correspond to the topological constraints with respect to a state in [17].

For example, consider the invariant subgraph in Figure 5 and a valuation γ where $\gamma[i_1] = 1$ and $\gamma[i_2] = 2$. The topological number of the output variable y_1 under γ is $\mathrm{t}_\gamma(y_1) = 1 + \max\left\{\mathrm{t}_\gamma(i_1), \mathrm{t}_\gamma(x_1)\right\} = 1 + 0 = 1$, because x_1 is a

search variable, and the topological number of the output variable y_2 under γ is $\mathrm{t}_\gamma(y_2) = 1 + \max\left\{\mathrm{t}_\gamma(i_2), \mathrm{t}_\gamma(y_1)\right\} = 1 + 1 = 2$, because $\mathrm{t}_\gamma(y_1) = 1$. Upon a slight change of the example to $\gamma[i_1] = 2$ and $\gamma[i_2] = 1$, the topological-number constraints of y_1 and y_2 are unsatisfiable, and the subgraph contains the undeterminable dynamic cycle $\langle y_1, \mathbb{I}_2, y_2, \mathbb{I}_1, y_1\rangle$ under γ.

If a topological-number constraint is unsatisfiable, then the invariant graph contains a static or an undeterminable dynamic cycle and the corresponding assignment cannot be moved to, as the value of each variable on each static or undeterminable cycle is not guaranteed to be determinable.

For any propagation algorithm that requires the topological number constraints of all nodes to be satisfiable, such as the algorithm in Section 3.4.3 below, the invariant graph cannot contain any static cycles. However, topologically sorting the nodes of an invariant graph is not required for the detection of static cycles, as static cycles are independent of assignments and valuations. Since both the static cycles of any invariant graph can be detected and the invariant graph can be transformed into an invariant graph without static cycles before search starts (see Section 2.8), we reason on invariant graphs without static cycles for the rest of this section.

If a level only has static invariants, then the topological numbers of the nodes in that level do not depend on the current valuation, and the level can be topologically sorted once, before propagation starts. Otherwise, the topological number of each node of that level might change between probes and moves, and the nodes in that level must be topologically sorted each time before the level is propagated.

We now prove by contradiction that an invariant graph without static cycles has an undeterminable dynamic cycle within a level if and only if the topological-number constraint of some node in that level is unsatisfiable.

Theorem 3.2 (an undeterminable dynamic cycle implies unsatisfiable topological-number constraints)**.** Consider an invariant graph $\mathcal{G}$ without static cycles; with nodes $\mathcal{N}$, variables $\mathcal{V} \subseteq \mathcal{N}$, and search variables $\mathcal{S} \subseteq \mathcal{V}$; the level $\ell > 1$; the set $\mathcal{L} = \{u \mid u \in \mathcal{N} \wedge \mathrm{lv}(u) < \ell\}$ of all nodes in shallower levels than ℓ; an assignment α; and the valuation γ that is correct on $\mathcal{L} \cap \mathcal{V}$ with respect to α, where the topological-number constraint of each node $u \in \mathcal{L}$ is satisfied. The topological-number constraint of each node $u \in \mathcal{N}$ with $\mathrm{lv}(u) = \ell$ is satisfiable if, and only if, level ℓ has no undeterminable dynamic cycles.

Proof. For any invariant $\mathbb{I}$, the level of any variable z that $\mathbb{I}$ depends on cannot be greater than $\mathrm{lv}(\mathbb{I})$. The set of transitively required (static or dynamic) input variables to any invariant under any valuation is a subset of the variables that the invariant depends on.

Assume that $\mathcal{G}$ has no undeterminable dynamic cycles under γ within level ℓ and that the topological-number constraint is not satisfiable for some invariant $\mathbb{I}$ in level ℓ with required (static and dynamic) input variables $\mathcal{X}_\gamma = \mathcal{X}_\mathrm{s} \cup \mathcal{X}_{\mathrm{d},\gamma}$ under γ. By definition, the topological-number constraint of $\mathbb{I}$ is not satisfiable. If the topological-number constraints of all variables in $\mathcal{X}_\gamma$ are satisfiable, then the topological-number constraint of $\mathbb{I}$ is satisfiable, which is a contradiction. Therefore, the topological-number constraint is not satisfiable for some transitively required (static or dynamic) input variable $x \in \mathcal{X}_\gamma$ to $\mathbb{I}$ under γ. By definition, variable x cannot be in a deeper level than ℓ. But if $\mathrm{lv}(x) < \ell$, then $x \in \mathcal{L}$ and the topological-number constraint of x is satisfiable, which is a contradiction. Therefore, we have $\mathrm{lv}(x) = \ell$. Additionally, the level constraint of each search variable is 1, thus $x \notin \mathcal{S}$. Since there are a finite number of nodes in level ℓ of $\mathcal{G}$, there exists some transitively required input variable $y \in \mathcal{V} \setminus \mathcal{S}$ to $\mathbb{I}$ where the following assertions hold:

- y is an output variable of some invariant $\mathbb{I}_y$;
- $\mathrm{lv}(y) = \ell$;
- the topological-number constraints of both y and $\mathbb{I}_y$ are unsatisfiable; and
- y is transitively a required input variable to $\mathbb{I}_y$.

Since y is transitively a required input variable to $\mathbb{I}_y$, level ℓ contains some cycle c. Since $\mathcal{G}$ contains no static cycles, the cycle c is an undeterminable dynamic cycle under γ, which is a contradiction.

Conversely, assume that there is an undeterminable dynamic cycle c under γ within level ℓ and that the topological-number constraint of each node in ℓ is satisfiable. Consider some invariant $\mathbb{I}$ in c with some output variable y. By definition, each node in c is in level ℓ and so is y. Therefore, y is also transitively a required (static or dynamic) input variable to $\mathbb{I}$. By the topological-number constraints, we must then have that $\mathrm{t}_\gamma(y)$ is transitively greater than $\mathrm{t}_\gamma(y)$, which is a contradiction as the topological-number constraint of y is assumed to be satisfiable. □

If a level ℓ contains no undeterminable dynamic cycles under a valuation γ, then the topological number constraints of the nodes in ℓ are satisfiable and induce a topological sort of the nodes in level ℓ.

We now show that given an invariant graph without static cycles and a valuation, where the invariant graph has no undeterminable dynamic cycles under that valuation, enforcing the invariants in order by their topological numbers will result in the correct valuation.

Theorem 3.3 (enforcing invariants in a level by topological order is correct)**.** Consider an invariant graph $\mathcal{G}$ with nodes $\mathcal{N}$; variables $\mathcal{V} \subseteq \mathcal{N}$; search variables $\mathcal{S} \subseteq \mathcal{V}$; a level ℓ; the set $\mathcal{L} = \{u \mid u \in \mathcal{N} \wedge \mathrm{lv}(u) < \ell\}$ of all nodes in shallower levels than ℓ; an assignment α; the valuation γ that is correct on $\mathcal{S} \cup (\mathcal{L} \cap \mathcal{V})$ with respect to α, where $\mathcal{G}$ has no dynamic cycles that contain an edge from a static input variable to a dynamic invariant, has no static cycles, and has no undeterminable dynamic cycles (on any level) under the valuation that is correct with respect to α. If the invariants in level ℓ are enforced by increasing topological numbers, then γ is made correct on all variables in level ℓ with respect to α.

Proof. Consider the search variables $\mathcal{S} \subseteq \mathcal{V}$ and the invariants $\mathcal{I} = \mathcal{N} \setminus \mathcal{V}$. Since $\mathcal{G}$ contains no dynamic cycles that contain an edge from a static input variable to a dynamic invariant, the level constraint of each node is satisfiable. Additionally, for each invariant $\mathbb{I} \in \mathcal{N}$ and each node n that $\mathbb{I}$ depends on, we have $\mathrm{lv}(n) \leq \mathrm{lv}(\mathbb{I})$. For each invariant $\mathbb{I} \in \mathcal{N}$ whose topological-number constraint is satisfiable and each node n that is or defines a variable that transitively is a required (static or dynamic) input variable to $\mathbb{I}$, we have $\mathrm{t}_\gamma(n) < \mathrm{t}_\gamma(\mathbb{I})$.

We prove that enforcing the invariants in level ℓ by increasing topological numbers makes γ correct on all variables in level ℓ with respect to α. We do this by induction on the topological number for the nodes in ℓ: the base case is for the nodes that have zero as topological number (and thus $\ell = 0$), and the inductive case is for the nodes that have a non-negative topological number (and thus $\ell \geq 0$).

The base case holds as only the search variables $\mathcal{S}$ have zero as topological number (and thus $\ell = 0$) and γ is assumed correct on $\mathcal{S}$ with respect to α.

For the inductive case, assume for some $\tau \geq 0$ that all the invariants of the set $\left\{\mathbb{I} \mid \mathbb{I} \in \mathcal{I} \wedge \mathrm{lv}(\mathbb{I}) = \ell \wedge \mathrm{t}_\gamma(\mathbb{I}) \leq \tau\right\}$ have been enforced by increasing topological numbers and that γ is already correct on all variables in level ℓ that have at most τ as topological number. We now show that for the case when $\tau + 1$, after enforcing each invariant in $\mathcal{I}_{\ell,\tau+1} = \left\{\mathbb{I} \mid \mathbb{I} \in \mathcal{I} \wedge \mathrm{lv}(\mathbb{I}) = \ell \wedge \mathrm{t}_\gamma(\mathbb{I}) = \tau + 1\right\}$, the valuation γ is correct also on all output variables of the invariants in $\mathcal{I}_{\ell,\tau+1}$ with respect to α.

Consider an invariant $\mathbb{I} \in \mathcal{I}_{\ell,\tau+1}$ with required (static or dynamic) input variables $\mathcal{X}_\gamma \subseteq \mathcal{V}$ under γ and output variables $\mathcal{Y} \subseteq \mathcal{V}$. Since each required (static or dynamic) input variable $x \in \mathcal{X}_\gamma$ under γ is in the same or a shallower level than $\mathbb{I}$ and since $\mathrm{t}_\gamma(x) < \mathrm{t}_\gamma(\mathbb{I})$, the valuation γ is correct on $\mathcal{X}_\gamma$ with respect to α by our assumption, as $\forall x \in \mathcal{X} : x \in \mathcal{L} \vee (\mathrm{lv}(x) = \ell \wedge \mathrm{t}_\gamma(x) \leq \tau)$. Enforcing $\mathbb{I}$ makes γ also correct on $\mathcal{Y}$. Since each output variable is an output of exactly one invariant and both the level and the topological number of each output variable equal the level and topological number respectively of the invariant it is an output variable of, after enforcing each invariant in level ℓ with topological number $\tau + 1$, the valuation γ is also correct with respect to the output variables of each invariant in level ℓ with topological number $\tau + 1$, and our statement holds for $\tau + 1$. □

3.4.3 Input-to-Output Propagation Algorithm (for Probing and Moving). Consider an invariant graph with the variables $\mathcal{V}$ and search variables $\mathcal{S} \subseteq \mathcal{V}$. Consider a neighbour α' of the current assignment α, and let the

Algorithm 2: The propagation of an invariant graph in input-to-output style.

Data: $\mathcal{V}$ is the set of variables; $\mathcal{S}$ is the set of search variables; α is the current assignment; γ is the valuation that is correct with respect to α; and α' is a neighbour of α.
Precondition: The invariant graph has no static cycles and no dynamic cycle of the invariant graph contains an edge from a static input variable to a dynamic invariant.
Postcondition: If the procedure does not abort, then the valuation γ is correct with respect to α', else the invariant graph contains an undeterminable dynamic cycle under γ.

```
1  procedure input-to-output(α′):
2      foreach x ∈ S do
3          update-valuation(γ, x, α′(x))
4      ℓ_max := max {lv(x) | x ∈ V}
5      W := create-array(ℓ_max, ∅)
       /* data invariant: For each level ℓ, the value γ[x] of each variable x ∈ W[ℓ] has been updated
          since the start of the execution.                                                    */
6      W[1] := {x | x ∈ S ∧ has-changed(x, γ)}
7      Q := create-priority-queue()
8      for 1 ≤ ℓ ≤ ℓ_max do
9          if ¬assign-topological-numbers(V, ℓ, γ, t_γ) then
10             abort
11         foreach x ∈ W[ℓ] do
12             enqueue(Q, x, t_γ(x))
13         while not empty(Q) do
14             x := dequeue(Q)
15             if has-changed(x, γ) then
16                 foreach 𝕀 ∈ listening-invariants(x) do
17                     enforce(𝕀, x, γ)
18                     foreach y ∈ output-variables(𝕀) where has-changed(y, γ) do
19                         if lv(y) = ℓ ∧ y ∉ W[ℓ] then
20                             enqueue(Q, y, t_γ(y))
21                         W[lv(y)] := W[lv(y)] ∪ {y}
```

current valuation γ be correct with respect to α. The procedure of probing or moving to α' in input-to-output style is given in Algorithm 2, with the following additional functions with respect to Algorithm 1:

- the function $\mathrm{lv}(x)$ returns the level of variable $x \in \mathcal{V}$, assumed computed before search;
- the function create-array(s, v) returns an array of size $s \in \mathbb{Z}^+$, where indexing starts at 1 and each element is the value v;
- the function create-priority-queue() returns an empty minimum-priority queue;
- the function assign-topological-numbers$(\mathcal{V}, \ell, \gamma, \mathrm{t}_\gamma)$ has the preconditions that the valuation γ is correct on $\mathcal{S}$ and all variables in levels shallower than ℓ; if level ℓ has no undeterminable dynamic cycles under γ, then the function:

(1) satisfies the topological-number constraints of the variables in ℓ under γ, and thereby initialises the topological numbers $\mathrm{t}_\gamma(x)$ of the variables x in ℓ if $\ell = 1$, else it updates them, and
(2) returns **true**;
otherwise ℓ has one or more undeterminable dynamic cycles and **false** is returned;

- the function enqueue(Q, x, t) adds the variable x with priority t to the minimum-priority queue Q;
- the function empty(Q) returns **true** if the queue Q is empty, else **false**;
- the function dequeue(Q) removes a variable x with the lowest priority from the minimum-priority queue Q and returns x;
- the function listening-invariants(x) returns the set of all invariants to which the variable x is an input variable.

First, the current valuation γ is made correct on $\mathcal{S}$ with respect to α', lines 2–3 (just like in Algorithm 1). The deepest level is found, line 4. The array $\mathcal{W}$ of initially empty sets is created, where for level ℓ, the value $\gamma[x]$ of variable $x \in \mathcal{W}[\ell]$ has been updated since the start of the execution and $\mathrm{lv}(x) = \ell$, line 5. Each updated search variable between α and α' is added to $\mathcal{W}[1]$, line 6, as all search variables are in level 1. The minimum-priority queue Q that will hold the updated variables in the current level is created empty, line 7. Each level ℓ is iterated over in an increasing order, line 8. If the topological-number constraints of all variables in level ℓ are satisfiable, then the topological numbers of those variables are assigned, else the invariant graph contains an undeterminable dynamic cycle under γ and the algorithm aborts, lines 9–10. Each variable of $\mathcal{W}[\ell]$ is enqueued into Q, with its topological number as priority, lines 11–12. While Q is not empty, line 13, a variable x with the lowest topological number is dequeued from Q, line 14. If the value that x took at the start of the algorithm is not equal to its current value under γ (in other words, if $\mathrm{v}_\alpha(x) \neq \gamma[x]$), line 15, then each invariant $\mathbb{I}$ that x is a required input to under γ is iterated over, line 16. Invariant $\mathbb{I}$ is enforced using the updated value of x under γ, possibly updating the values of the output variables of $\mathbb{I}$ under γ, line 17. Each output variable y of $\mathbb{I}$ whose value under γ differs from the value y took at the start of the execution is iterated over, line 18. If y is also in level ℓ but not in the set $\mathcal{W}[\ell]$, line 19, then y is enqueued into Q, with its topological number as priority, line 20. Variable y is added to the set $\mathcal{W}[\mathrm{lv}(y)]$, line 21.

After an invariant graph is propagated in input-to-output style, the current valuation γ is correct on all variables, so the values of the probed variables under γ can be evaluated by the selection heuristic, and the algorithm can also be used for moving. Algorithm 2 is a direct implementation of Theorem 3.3 and is thus correct.

3.5 Time Complexity Analysis

We now give the running-time complexities of Algorithms 1 and 2 as well as of the ad-hoc marking strategy.

We say that an edge $(x, \mathbb{I})$ from an input variable x to an invariant $\mathbb{I}$ is *followed* under γ when $\mathbb{I}$ is enforced under γ given x (i.e., when the call enforce$(\mathbb{I}, x, \gamma)$ is made during propagation).

We now show that, for any invariant graph, each search variable that also is a probed variable can be removed from the invariant graph. Consider an invariant graph with a variable x that is both a search variable and a probed variable. Since no other probed variable or violation variable depends on x, the value that x takes under any assignment α does not affect the value that any other variable takes under α. Therefore, we can remove x from the invariant graph, replacing it with value v. If x is an objective variable, then v is an optimal value of x, else v is some value of x. For the sequel of this section, we can therefore reason on invariant graphs where the sets of search variables and probed variables do not intersect, without loss of generality.

Consider an invariant graph $\mathcal{G}$ with no static cycle; no dynamic cycle with an edge from a static input to a dynamic invariant; edges $\mathcal{E}$; nodes $\mathcal{N}$; variables $\mathcal{V} \subseteq \mathcal{N}$; search variables $\mathcal{S} \subseteq \mathcal{V}$; probed variables $\mathcal{P} \subseteq (\mathcal{V} \setminus \mathcal{S})$; a current assignment α; the current valuation γ that is correct with respect to α; a neighbour α' of α; and the

valuation γ' that is correct with respect to α'. The above, plus the following notation, are used in all the theorems below:

- $\mathcal{U} \subseteq \mathcal{V}$ is the set of variables updated between α and α';
- $\mathcal{D} \supseteq \mathcal{U}$ is the set of variables that are, or depend on, a variable in $\mathcal{U}$;
- $\mathcal{M} \supseteq \mathcal{D}$ is the set of marked variables;
- $C \subseteq \mathcal{N}$ is the set of nodes in levels that contain one or more cycles;
- $\mathcal{E}_{\mathcal{D}} \subseteq \mathcal{E}$ is the set of edges that go from a variable in $\mathcal{D}$ to some invariant;
- $\mathcal{V}_{\mathrm{c}} \subseteq \mathcal{V}$ is the set of variables that are on a level that has one or more cycles;
- $\mathcal{E}_C = \{(u,v) \mid (u,v) \in \mathcal{E} \wedge \mathrm{lv}(u) = \mathrm{lv}(v) \wedge \{u,v\} \cap C \neq \emptyset\}$ is the set of edges whose endpoints are on the same level, which has one or more cycles;
- $\mathcal{E}_{\mathrm{s},\mathcal{U}} \subseteq \mathcal{E}$ is the set of edges that go from a static input variable in $\mathcal{U}$ to some (static or dynamic) invariant;
- $\mathcal{E}_{\mathrm{d},\mathcal{U}} \subseteq \mathcal{E}$ is the set of edges that go from a dynamic input variable in $\mathcal{U}$ to some dynamic invariant;
- $\mathcal{E}_{\mathrm{r},\gamma'} \subseteq \mathcal{E}$ is the set of edges $(x,\mathbb{I})$ where variable x is a required (static or dynamic) input to invariant $\mathbb{I}$ under valuation γ' (for example, in Figure 6, if $\mathrm{v}_{\alpha'}(i) = 1$, then $\mathcal{E}_{\mathrm{r},\gamma'} = \{(x_1,\mathbb{I}_1),(x_2,\mathbb{I}_2),(x_3,\mathbb{I}_3),(y_1,\mathbb{I}_4),(i,\mathbb{I}_4)\}$, else $\mathrm{v}_{\alpha'}(i) = 2$ and $\mathcal{E}_{\mathrm{r},\gamma'} = \{(x_1,\mathbb{I}_1),(x_2,\mathbb{I}_2),(x_3,\mathbb{I}_3),(y_2,\mathbb{I}_4),(i,\mathbb{I}_4)\}$);
- $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}} \subseteq \mathcal{E}_{\mathrm{r},\gamma'}$ is the set of edges $(x,\mathbb{I})$ where variable x is a required (static or dynamic) input to invariant $\mathbb{I}$ under valuation γ' and $\mathbb{I}$ has an output variable in $\mathcal{M}$ (for example, in Figure 6, we have $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}} = \{(x_1,\mathbb{I}_1),(x_3,\mathbb{I}_3),(y_1,\mathbb{I}_4),(y_2,\mathbb{I}_4),(i,\mathbb{I}_4)\}$);
- $\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'} \subseteq \mathcal{E}_{\mathrm{d},\mathcal{U}}$ is the set of edges $(x,\mathbb{I})$ where variable $x \in \mathcal{U}$ is updated and a required dynamic input to invariant $\mathbb{I}$ under γ' (for example, in Figure 6, if $\mathrm{v}_{\alpha'}(i) = 1$, then $\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'} = \{(y,\mathbb{I}_4) \mid y \in (\{y_1\} \cap \mathcal{U})\}$, else $\mathrm{v}_{\alpha'}(i) = 2$ and $\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'} = \{(y,\mathbb{I}_4) \mid y \in (\{y_2\} \cap \mathcal{U})\}$);
- κ is the worst-case time complexity of enqueueing an element into a minimum-priority queue;
- μ is the worst-case time complexity of marking $\mathcal{G}$;
- ι is the worst-case time complexity of the is-marked function: this is constant time for ad-hoc marking, non-constant time for prepared marking, and zero time for total marking; and
- ϵ is the worst-case time complexity of the enforce function; the invariants are typically incremental and the enforce function then typically takes constant time (e.g., this is the case for AllDifferent and Sum).

Since each invariant has one or more input variables and one or more output variables, we have $|\mathcal{N}| \leq 2 \cdot |\mathcal{V}|$.

Theorem 3.4 (time complexity of ad-hoc marking)**.** The time complexity of ad-hoc marking is $O(|\mathcal{E}_{\mathcal{D}}|)$.

Proof. Consider the updated search variable $x \in \mathcal{S} \cap \mathcal{U}$ between α and α', the set $\mathcal{N}_x \subseteq \mathcal{D}$ that has x and only the variables that depend on x, and the set $\mathcal{E}_x \subseteq \mathcal{E}_{\mathcal{D}}$ that connects the nodes in $\mathcal{N}_x$. The sets $\mathcal{N}_x$ and $\mathcal{E}_x$ induce an invariant subgraph that is a directed rooted tree with root x. The nodes $\mathcal{N}_{\mathcal{S},\mathcal{U}} = \cup_{x \in \mathcal{S} \cap \mathcal{U}} \mathcal{N}_x$ and edges $\mathcal{E}_{\mathcal{S},\mathcal{U}} = \cup_{x \in \mathcal{S} \cap \mathcal{U}} \mathcal{E}_x$, with $\mathcal{E}_{\mathcal{S},\mathcal{U}} \subseteq \mathcal{E}_{\mathcal{D}}$, induce the invariant subgraph $\mathcal{G}_{\mathcal{S},\mathcal{U}}$. If $\mathcal{G}_{\mathcal{S},\mathcal{U}}$ is a directed rooted tree, then $\left|\mathcal{E}_{\mathcal{S},\mathcal{U}}\right| = \left|\mathcal{N}_{\mathcal{S},\mathcal{U}}\right| - 1$, else $\left|\mathcal{E}_{\mathcal{S},\mathcal{U}}\right| > \left|\mathcal{N}_{\mathcal{S},\mathcal{U}}\right| - 1$. Thus, $\left|\mathcal{E}_{\mathcal{S},\mathcal{U}}\right| \geq \left|\mathcal{N}_{\mathcal{S},\mathcal{U}}\right| - 1$, and the time complexity of depth-first search on $\mathcal{G}_{\mathcal{S},\mathcal{U}}$ is $O(\left|\mathcal{E}_{\mathcal{S},\mathcal{U}}\right|)$. Since $\mathcal{E}_{\mathcal{S},\mathcal{U}} \subseteq \mathcal{E}_{\mathcal{D}}$, to perform depth-first search on the invariant graph has the time complexity $O(|\mathcal{E}_{\mathcal{D}}|)$. □

Theorem 3.5 (time complexity of output-to-input propagation)**.** The time complexity of Algorithm 1 is:

$$\begin{cases} O\left(\mu + \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right| + \epsilon \cdot \left(\left|\mathcal{E}_{\mathrm{s},\mathcal{U}}\right| + \left|\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'}\right|\right)\right) & \text{if } \iota = 0 \\ O\left(\mu + \iota \cdot \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right| + \epsilon \cdot \left(\left|\mathcal{E}_{\mathrm{s},\mathcal{U}}\right| + \left|\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'}\right|\right)\right) & \text{otherwise} \end{cases}$$

Proof. To perform marking has time complexity μ. For each invariant $\mathbb{I}$ that defines a marked output variable with the set $\mathcal{X}_{\mathrm{s}}$ of (required) static input variables and the set $\mathcal{X}_{\mathrm{d},\gamma'}$ of required dynamic input variables under γ', we have that (i) for each required input variable $x \in \mathcal{X}_{\mathrm{s}} \cup \mathcal{X}_{\mathrm{d},\gamma'}$ there is an is-marked operation, which verifies if x is marked or not, and (ii) for each updated input variable $x \in (\mathcal{X}_{\mathrm{s}} \cap \mathcal{X}_{\mathrm{d},\gamma'}) \cap \mathcal{U}$ there is an enforce operation. For

all required input variables under γ', this results in at most $\left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right|$ is-marked operations and at most $\left|\mathcal{E}_{\mathrm{s},\mathcal{U}}\right| + \left|\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'}\right|$ enforce operations. However, since ι can be 0, there might be no is-marked operations. Each marked variable is recursed on at most once. Since a variable is defined by at most one invariant, the number of invariants that define a marked output variable is at most the number of marked output variables. Additionally, since $\mathcal{S} \cap \mathcal{P} = \emptyset$, the number of marked search variables is at most the number of invariants that define a marked variable. Therefore, the number of variables that are recursed on is at most $\left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right|$. The time complexity of Algorithm 1 is thus as announced. □

Theorem 3.6 (time complexity of input-to-output propagation)**.** The time complexity of Algorithm 2 is $O(\kappa \cdot |\mathcal{E}_{\mathcal{D}}| + |\mathcal{E}_C| + |C| + \epsilon \cdot (\left|\mathcal{E}_{\mathrm{s},\mathcal{U}}\right| + \left|\mathcal{E}_{\mathrm{d},\mathcal{U}}\right|))$.

Proof. Since propagation is performed iteratively similarly to how breadth-first search is performed, only variables that depend on updated search variables between α and α' can be enqueued to the minimum-priority queue. Therefore, there are at most $|\mathcal{E}_{\mathcal{D}}|$ enqueue operations and as many dequeue operations.

Since all dynamic cycles of $\mathcal{G}$ are determinable under γ, topologically sorting each level of the invariant graph can be done using depth-first search. Additionally, since the levels partition $\mathcal{N}$, the time complexity of topologically sorting the cyclic levels is $O(|\mathcal{E}_C| + |C|)$.

For each updated variable $x \in \mathcal{U}$ and each invariant that x is an input variable to, there is an enforce operation, so this results in $\left|\mathcal{E}_{\mathrm{s},\mathcal{U}}\right| + \left|\mathcal{E}_{\mathrm{d},\mathcal{U}}\right|$ enforce operations.

For each updated search variable $x \in \mathcal{S} \cap \mathcal{U}$, there is a constant-time update-valuation operation when γ is initialised, but since $\mathcal{S} \cap \mathcal{P} = \emptyset$ and the number of these operations is at most $|\mathcal{E}_{\mathcal{D}}|$, they are ignored as their cost is already accounted for by a previous term.

The time complexity of Algorithm 2 is thus as announced. □

3.6 Comparing Running Times and Recommending a Propagation Style

With the use of the notation of Section 3.5 and Theorems 3.4, 3.5, and 3.6, we now compare the running-time complexities of Algorithms 1 and 2, so as to recommend a propagation style based on features of an invariant graph.

First, we remove the term $\epsilon \cdot \left|\mathcal{E}_{\mathrm{s},\mathcal{U}}\right|$ as it appears in both Theorem 3.5 and Theorem 3.6, resulting in the comparison of the following complexities, respectively:

(1) $\begin{cases} O\left(\mu + \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right| + \epsilon \cdot \left|\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'}\right|\right) & \text{if } \iota = 0 \\ O\left(\mu + \iota \cdot \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right| + \epsilon \cdot \left|\mathcal{E}_{\mathrm{d},\mathcal{U},\mathrm{r},\gamma'}\right|\right) & \text{otherwise} \end{cases}$

(2) $O(\kappa \cdot |\mathcal{E}_{\mathcal{D}}| + |\mathcal{E}_C| + |C| + \epsilon \cdot \left|\mathcal{E}_{\mathrm{d},\mathcal{U}}\right|)$

We give a recommendation first for the marking strategy, then for the propagation style for invariant graphs without dynamic invariants, and finally for the propagation style for invariant graphs with dynamic invariants.

3.6.1 Recommendation for the Marking Strategy. The choice of prepared, ad-hoc, or total marking has an impact on the performance of Algorithm 1, as the marking strategy affects the time complexities μ and ι and potentially reduces the size of the set $\mathcal{M}$ of marked variables and by extension the size of the set $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}$:

- if total marking is used, then $\mu = 0$, $\iota = 0$, and $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}} \subseteq \mathcal{E}_{\mathrm{r},\gamma'}$;
- if ad-hoc marking is used, then $\mu = O(|\mathcal{D}|)$, $\iota = O(1)$, and $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}} \subseteq \mathcal{E}_{\mathrm{r},\gamma'}$, where for each edge $(x, \mathbb{I}) \in \mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}$, invariant $\mathbb{I}$ depends on an updated search variable; and
- if prepared marking is used, then $\mu = 0$, $\iota = O(|\mathcal{S} \cap \mathcal{U}|)$, and $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}} \subseteq \mathcal{E}_{\mathrm{r},\gamma'}$ where for each edge $(x, \mathbb{I}) \in \mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}$, invariant $\mathbb{I}$ depends on an updated search variable and for some output y of $\mathbb{I}$:
  - y is a probed variable, or

– y is transitively is a required (static or dynamic) input to some invariant $\mathbb{I}'$ under valuation γ', where $\mathbb{I}'$ defines a probed variable.

Therefore, if $|\mathcal{S} \cap \mathcal{U}| \cdot \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right|$ is smaller than $\left|\mathcal{E}_{\mathrm{r},\gamma'}\right|$, then prepared marking has better performance than ad-hoc marking. Analogously, if $|\mathcal{E}_{\mathcal{D}}| + \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right|$ is smaller than $\left|\mathcal{E}_{\mathrm{r},\gamma'}\right|$, then ad-hoc marking has better performance than prepared marking.

3.6.2 Recommendation for Invariant Graphs Without Dynamic Invariants. For any invariant graph without dynamic invariants (and therefore without dynamic cycles), the respective complexity is simplified further into:

(1) $\begin{cases} O\left(\mu + \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right|\right) & \text{if } \iota = 0 \\ O\left(\mu + \iota \cdot \left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right|\right) & \text{otherwise} \end{cases}$

(2) $O(\kappa \cdot |\mathcal{E}_{\mathcal{D}}|)$

Since $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}$ contains each edge to an invariant from each of its input variables (as there are no dynamic invariants), we have $\mathcal{E}_{\mathcal{D}} \subseteq \mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}$. If $\mathcal{E}_{\mathcal{D}}$ is smaller than $\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}$ by at least one order of magnitude, then we recommend the usage of the input-to-output style (Algorithm 2). Otherwise, since the opposite cannot be true, the recommended propagation style depends on κ and the marking strategy.

3.6.3 Recommendation for Invariant Graphs with Dynamic Invariants. As $\left|\mathcal{E}_{\mathrm{r},\gamma'}\right|$ and (by extension) $\left|\mathcal{E}_{\mathrm{r},\gamma',\mathcal{M}}\right|$ depend on the number of edges that originate from static input variables, we recommend the input-to-output invariant graph propagation style (Algorithm 2) for an invariant graph where the number of such edges is greater than the number of edges from dynamic input variables to dynamic invariants by at least one order of magnitude. *However*, if, despite a recommendation to the contrary, output-to-input propagation (Algorithm 1) is used, then we recommend the ad-hoc marking strategy for such an invariant graph.

Analogously, as $\left|\mathcal{E}_{\mathrm{d},\mathcal{U}}\right|$ depends on the number of edges that originate from dynamic input variables and as C and $\mathcal{E}_C$ are only non-empty when the invariant graph has dynamic invariants, we recommend the output-to-input invariant graph propagation style (Algorithm 1) for an invariant graph where the number of such edges is greater than the number of edges from static input variables to invariants by at least one order of magnitude, or the number of dynamic invariants is greater than the number of static invariants by at least one order of magnitude, or both.

Additionally, if neither $\left|\mathcal{E}_{\mathrm{r},\gamma'}\right|$ nor $\left|\mathcal{E}_{\mathrm{d},\mathcal{U}}\right|$ is greater than the other by at least one order of magnitude, then the recommended propagation style depends on κ and the marking strategy.

4 Experiments

We developed a CBLS solver, called Atlantis, that can propagate an invariant graph in both input-to-output and output-to-input styles.[1] For the output-to-input style, the solver supports the total, ad-hoc, and prepared marking strategies.

Given parameter values, an *invariant graph model* is a description of how an invariant graph is created. We designed six invariant graph models, for four classical problems — Golomb ruler [10, problem 6], magic square [10, problem 19], the travelling salesperson problem with time windows [3], and vessel loading [10, problem 8] — plus two handmade ones, called extreme dynamic and extreme static. A property of the set of designed invariant graph models is that they give a varied ratio between the number of edges from static input variables to (static or dynamic) invariants and the number of edges from dynamic input variables to dynamic invariants. Note that any set of invariant graph models with this property could have been designed, as we are only interested in the throughput of the propagation algorithms and not in the actual underlying problems or the time to solve their

[1]The source code of Atlantis is publicly available at https://github.com/astra-uu-se/atlantis/

instances. However, we selected some classical problems as they are well-known and some of them have real-life applications.

In the following subsections, we first describe the four classical problems and our designed invariant graph models for them (creating graphs that one would like a CBLS solver, such as fzn-oscar-cbls [5] or Yuck [16], to infer from a MiniZinc model), before we describe our handmade invariant graph models, detail our experiment method, and give the experiment results.

Note that we will always give an *upper* bound on the number of edges followed when the output-to-input propagation style is used. For example, if the value of a visited output variable of some invariant $\mathbb{I}$ has not been updated between the current assignment and the (probed) neighbouring assignment, then *no* edge to $\mathbb{I}$ is followed.

4.1 Golomb Ruler (GR)

A *Golomb ruler* of size n is a set of n strictly increasing non-negative integers whose $\frac{n\cdot(n-1)}{2}$ pairwise absolute-value differences are distinct.

Our invariant graph model for a Golomb ruler of size n creates an invariant graph with the array $\mathcal{S} = [x_1, \ldots, x_n]$ of n search variables, where x_i denotes the ith integer of the Golomb ruler; the set $\mathcal{Y}$ of $\frac{n\cdot(n-1)}{2}$ variables $y_{i,j}$; the static invariant $y_{i,j} \Longleftarrow x_j - x_i$ for each pair $(x_i, x_j) \in \mathcal{S}^2$ with $i < j$; the probed violation variable v; and the static violation invariant AllDifferent$(\mathcal{Y}, v)$. An implicit constraint maintains the values of the search variables x_i to be non-negative and strictly increasing (so that the value of $y_{i,j}$ can be determined without the use of an absolute-value operator): after such an initialisation, a probe consists of updating one search variable such that this remains the case.

The number of edges in the invariant graph between invariants and their static input variables is $\frac{3\cdot n\cdot(n-1)}{2}$. The invariant graph has no dynamic invariants and no cycles. The number of marked variables for any probe with ad-hoc marking is n. Since the value of a single search variable x_i is updated at each iteration, since there are $n-1$ variables in $\mathcal{Y}$ that depend on x_i, and since each variable in $\mathcal{Y}$ is an input to the AllDifferent invariant, up to $\frac{(n+2)\cdot(n-1)}{2}$ edges are followed when the output-to-input propagation style is used.

4.2 Magic Square (MS)

A *magic square* of size n is an $n \times n$ matrix of distinct values in $\{1, \ldots, n^2\}$ so that the $2 \cdot n + 2$ sums of its rows, columns, and two main diagonals are all equal to $s = \frac{n^2\cdot(n^2+1)}{2\cdot n}$.

Our invariant graph model for a magic square of size n creates an invariant graph with n^2 search variables, where $x_{r,c}$ denotes the element in row r and column c of the magic square; the static Sum invariant with output variable y_k for each row, column, and main diagonal k; the static violation invariant $v_k \Longleftarrow |y_k - s|$ for each row, column, and main diagonal k determining the value of the violation variable $v_k \in \mathcal{V}$ as the distance of k to the desired sum s; and the static invariant Sum$(\mathcal{V}, v)$ determining the value of the probed total-violation variable v. An implicit constraint maintains the n^2 search variables $x_{r,c}$ to be in $\{1, \ldots, n^2\}$ and distinct: after such an initialisation, a probe consists of swapping two search variables.

The number of edges in the invariant graph between invariants and their static input variables is $2 \cdot n^2 + 4 \cdot n + 2$. The invariant graph has no dynamic invariants and no cycles. The number of marked variables for any probe that switches the values of variables x_i and x_j with ad-hoc marking is 4 ± 1, as it depends on whether x_i, or x_j, or both are on the main diagonals. When the output-to-input propagation style is used, up to $8 \cdot n + 2$ edges are followed.

4.3 Travelling Salesperson Problem with Time Windows (TSPTW)

Consider n locations that are to be visited, where there is a travelling duration $\mu_{u,v}$ between each directed pair (u, v) of locations and there is for each location u an earliest visiting time e_u and a latest visiting time ℓ_u. A *travelling salesperson tour with time windows* (*TSPTW*) of size n is a Hamiltonian path of the weighted directed

graph induced by the n locations as nodes, visiting each location u exactly once and between times e_u and ℓ_u. The departure time at the last-visited location is to be minimised.

Our invariant graph model for a TSPTW of size n creates an invariant graph with the array $\mathcal{S} = [x_1, \ldots, x_{n+1}]$ of $n+1$ search variables, one for each location plus one for a dummy location $n+1$, where x_u denotes the possibly dummy location that is visited just before location u, and x_{n+1} thus denotes the last actual location of the tour. The additional variable x_{n+1} and the dummy location $n+1$ are needed for transforming the tour from a Hamiltonian path into a Hamiltonian cycle, for helping to express constraints from the problem in the model, and for denoting where the tour starts and ends. For each location $1 \leq u \leq n$, the invariant graph model creates:

- the variable t_u that denotes the travelling duration from location x_u to location u;
- the variable a_u that denotes the arrival time at location u;
- the variable d_u that denotes the departure time from location u; it differs from a_u if the salesperson arrives at u before e_u and has to wait there until e_u before departing;
- the variable d'_u that denotes the departure time from location x_u;
- the violation variable v_u that denotes the lateness (possibly zero) of arrival at location u compared to ℓ_u;
- the static Element invariant $t_u \Longleftarrow [\mu_{1,u}, \ldots, \mu_{n,u}, 0][x_u]$ that defines t_u; note that the invariant is static as each $\mu_{i,u}$ is a parameter;
- the static invariant $d_u \Longleftarrow \max(a_u, e_u)$ that defines d_u;
- the dynamic Element invariant $d'_u \Longleftarrow [d_1, \ldots, d_{n+1}][x_u]$ that defines d'_u;
- the static invariant $a_u \Longleftarrow d'_u + t_u$ that defines a_u; and
- the static violation invariant $v_u \Longleftarrow \max(0, a_u - \ell_u)$ that defines v_u.

The invariant graph model creates the additional parameters $t_{n+1} = d_{n+1} = 0$ for the dummy location. Additionally, the invariant graph model creates the dynamic Element invariant $o \Longleftarrow [d_1, \ldots, d_n, 0][x_{n+1}]$ that defines the probed objective variable o, which denotes the departure time from the last location of the tour (which is the arrival time at the dummy location), as well as the static invariant $\textsc{Sum}(\{v_1, \ldots, v_n\}, v)$ that defines the probed total-violation variable v. An implicit constraint maintains that we always have a Hamiltonian cycle during search: after such an initialisation, a probe is the execution of a 3-opt [15] on the vector of search variables.

The number of edges in the invariant graph between invariants and their static input variables is $4 \cdot n$. The number of edges between invariants and their dynamic input variables is n^n. The invariant graph has one cycle, with variables $\cup_{u=1}^{n} \{a_u, d_u, d'_u\}$ and the invariants that define them. The number of marked variables for any probe with ad-hoc marking is $3 \cdot n + 5$. When the output-to-input propagation style is used, up to $4 \cdot n + 5$ edges are followed.

4.4 Vessel Loading (VL)

Consider n rectangles, where each rectangle i has integer length ℓ_i (along the horizontal x axis) and width w_i (along the vertical y axis). A *vessel loading* of size n is a placement of the n rectangles within a single plane of a given rectangular bounding area of length λ and width ψ, whose origin is in its lower-left corner, such that each rectangle is parallel to the sides of that area. Additionally, each pair of rectangles (i, j) must be separated by a safety distance $s_{i,j}$ from each other, where $s_{i,j} = s_{j,i}$ is a nonnegative integer.

Our invariant graph model for a vessel loading of size n creates an invariant graph with the set $\{o_1, \ldots, o_n\}$ of search variables, where o_i is 1 if rectangle i has its given orientation, else 2 and i is rotated by 90 degrees; the set $\{x_1, \ldots, x_n\}$ of search variables, where x_i denotes the left-most position of rectangle i and takes a value in $\{0, \ldots, \lambda - \min(\ell_i, w_i)\}$; and the set $\{y_1, \ldots, y_n\}$ of search variables, where y_i denotes the bottom-most position of rectangle i and takes a value in $\{0, \ldots, \psi - \min(\ell_i, w_i)\}$. For each rectangle $1 \leq i \leq n$, the invariant graph model creates:

- the dynamic Element invariant $x'_i \Longleftarrow [x_i + \ell_i,\ x_i + w_i][o_i]$ that defines x'_i as the right-most position of rectangle i; and
- the dynamic Element invariant $y'_i \Longleftarrow [y_i + w_i,\ y_i + \ell_i][o_i]$ that defines y'_i as the top-most position of rectangle i.

For each pair of rectangles (i, j) with $1 \leq i < j \leq n$, the invariant graph model creates the static invariant $c^{\mathrm{r}}_{i,j} \Longleftarrow \max(0, x'_i + s_{i,j} - x_j)$ so that $c^{\mathrm{r}}_{i,j}$ is 0 if j is a safe distance right of i, and the required additional distance needed to safely separate them otherwise. Similarly, the invariant graph model creates three static invariants that define the variables $c^{\mathrm{l}}_{i,j}$, $c^{\mathrm{a}}_{i,j}$, and $c^{\mathrm{b}}_{i,j}$ for rectangle j being a safe distance left of, above, and below rectangle i respectively. Finally, the invariant graph model creates the static violation invariant $v_{i,j} \Longleftarrow \min(c^{\mathrm{r}}_{i,j}, c^{\mathrm{l}}_{i,j}, c^{\mathrm{a}}_{i,j}, c^{\mathrm{b}}_{i,j})$ that defines violation variable $v_{i,j} \in \mathcal{V}$ to be 0 if rectangles i and j are separated by their safety distance, and the minimum additional required (horizontal or vertical) distance needed to safely separate them otherwise. The invariant graph model creates the static invariant $\textsc{Sum}(\mathcal{V}, v)$ that defines the probed total-violation variable v. An implicit constraint maintains that each rectangle is placed within the bounding area: after such an initialisation, a probe consists of updating one or more of the variables o_i, x_i, and y_i for some rectangle i so that it remains within the bounding area.

The number of edges in the invariant graph between invariants and their static input variables is $\frac{9 \cdot n^2 - 5 \cdot n}{2}$. The number of edges between invariants and their dynamic input variables is $2 \cdot n$. The invariant graph has no cycles. The number of marked variables for any probe with ad-hoc marking is $5 \cdot n - 1$. When the output-to-input propagation style is used, up to $\frac{n \cdot (n+7)}{2}$ edges are followed.

4.5 Extreme Dynamic (ED)

Our *extreme dynamic* invariant graph model for size n creates an invariant graph with the array $\mathcal{X} = [x_1, \ldots, x_n]$ of n search variables and the array $\mathcal{Y} = [y_1, \ldots, y_n]$ of n variables; the search variable i; the probed variable o; the set of dynamic Element invariants $y_j \Longleftarrow [x_1, \ldots, x_n][i]$; and the dynamic Element invariant $o \Longleftarrow [y_1, \ldots, y_n][i]$. Note that there are no violation variables and no violation invariants. A probe consists of updating some search variable x_j, which is a dynamic input variable. We opted not to allow probes where i is updated, as the cost of such a probe might be different from a probe where some search variable x_j is updated.

The acyclic invariant graph is in Figure 8. The number of edges between invariants and their static input variables is $n + 1$. The number of edges between invariants and their dynamic input variables is $n^2 + n$. The number of marked variables for any probe with ad-hoc marking is $n + 1$. When the output-to-input propagation style is used, up to 4 edges are followed.

We designed this invariant graph model so that the number of edges from dynamic input variables to dynamic invariants is quadratic in n, while the number of edges from static input variables to invariants is only linear in n. Additionally, the number of edges that are followed during propagation is linear in n for input-to-output style, but constant in output-to-input style.

4.6 Extreme Static (ES)

Our *extreme dynamic* invariant graph model for size n creates an invariant graph with the array $\mathcal{X} = [x_1, \ldots, x_n]$ of n search variables and the static invariant $\textsc{Sum}(\mathcal{X}, s)$ that defines the probed variable s to be their sum. Note that there are no violation variables and no violation invariants. A probe consists of updating some search variable x_i.

The number of edges in the invariant graph between invariants and their static inputs is n. There are no dynamic invariants. The number of marked variables for any probe with ad-hoc marking is 2. When the output-to-input propagation style is used, up to $n + 1$ edges are followed.

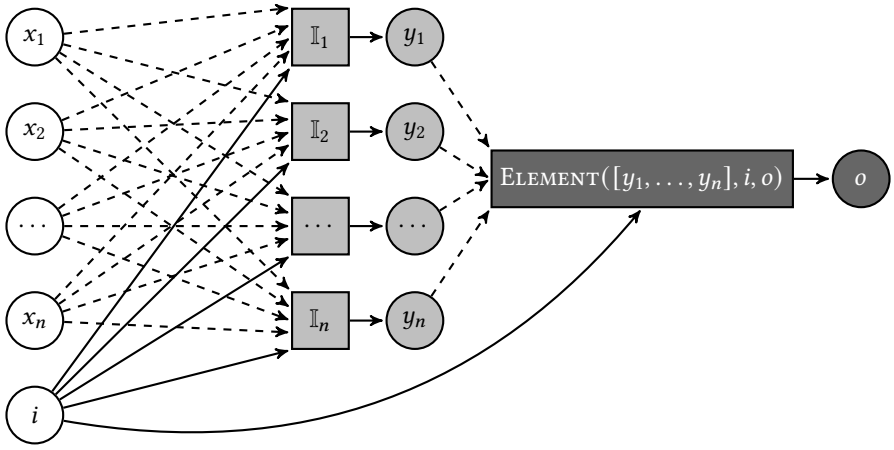

Fig. 8. Invariant graph for an ED of size n, where each invariant $\mathbb{I}_j$ is Element$([x_1, \ldots, x_n], i, y_j)$. Each dashed edge $(z, \mathbb{I})$ goes from the dynamic input variable z to the dynamic invariant $\mathbb{I}$. Each solid edge $(z, \mathbb{I})$ goes from the static input variable z to the invariant $\mathbb{I}$. The white nodes are in level 1, the light grey ones in level 2, and the dark grey ones in level 3.

We designed this invariant graph model so that the invariant graph has a single static invariant, no dynamic invariants, no dynamic inputs, and a number of edges from static input variables to invariants that is linear in n. Additionally, the number of edges that are followed during propagation is constant for input-to-output style, but linear in n for output-to-input style.

4.7 Method

Solving to satisfaction or optimality depends on the initialisation, neighbours, selection heuristic, and meta-heuristic, as well as on luck each time randomisation is performed, but all this is orthogonal to our purpose, namely measuring the throughput, that is the number of probes per second, of the invariant graph propagation styles, in order to compare them. So we need neither record the time taken to find solutions nor compare best-found objective values, and this neither between invariant graph propagation styles nor with the state of the art for each of the first four invariant graph models. Additionally, given a created invariant graph for particular parameter values, an invariant graph propagation style, and a valid marking strategy, roughly the same number of computations is performed per iteration whether the instance is easily satisfied, difficult to satisfy, or infeasible. The difficulty and realism of the problem instances are thus unimportant, so we generated random parameter values instead of retrieving instances from existing repositories. Furthermore, during search, we assume that the number of probes is typically one order of magnitude greater than the number of moves: the number of probes an invariant graph propagation algorithm can perform per second is thus of utmost importance.

For each invariant graph model, we generated 9 instances, of sizes 16, 32, 64, 96, 128, 196, 256, 512, and 1024. The throughput is measured for each instance when the input-to-output propagation style (denoted "input-to-output"), output-to-input propagation style with ad-hoc marking (denoted "output-to-input – ad-hoc"), output-to-input propagation style with prepared marking (denoted "output-to-input – prepared"), and output-to-input propagation style with total marking (denoted "output-to-input – total") are used.

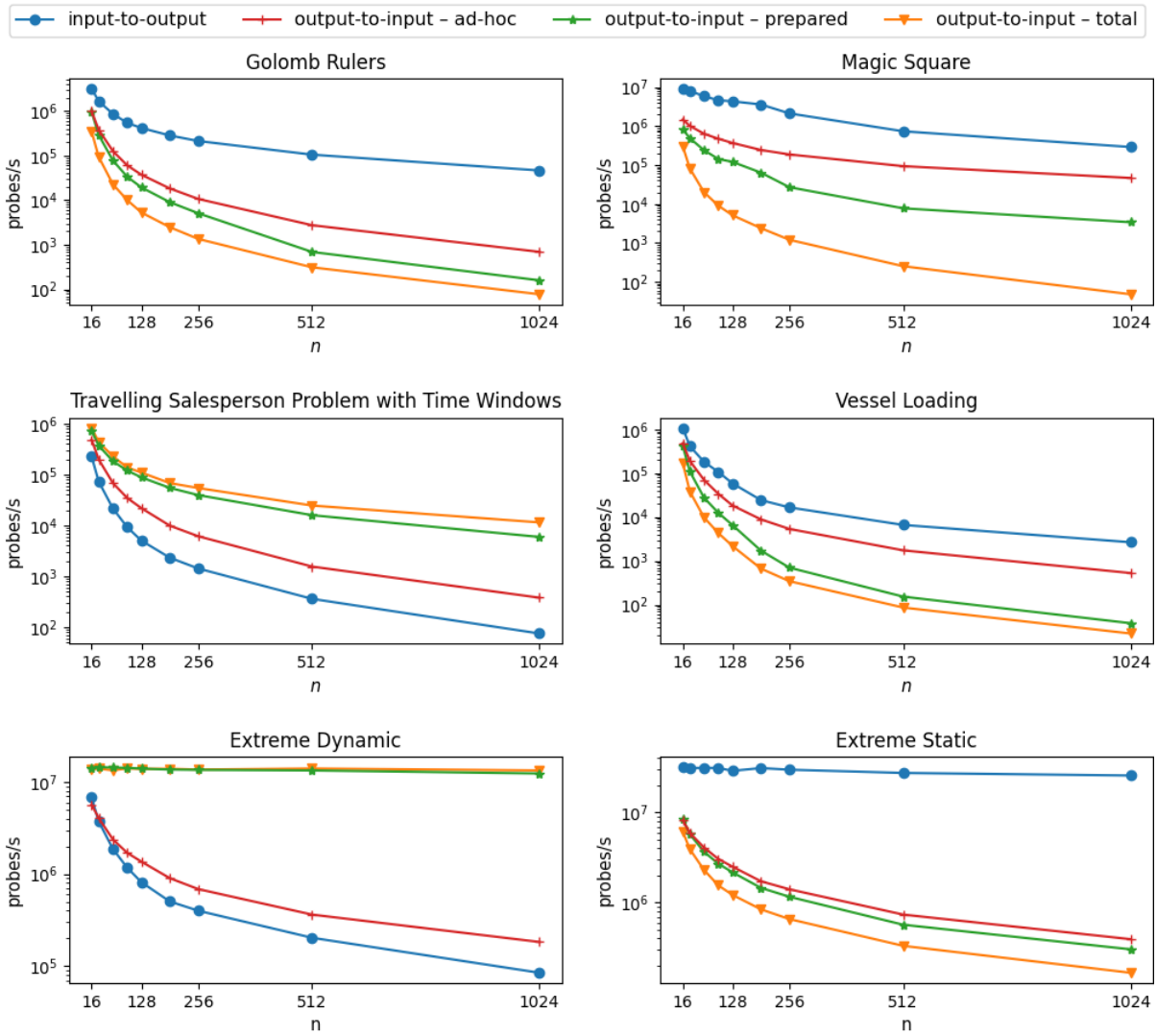

Fig. 9. The number of probes per second for various invariant graphs with input-to-output and output-to-input propagation, where output-to-input propagation uses either ad-hoc marking, or prepared marking, or total marking. The horizontal axis of each graph corresponds to the size of the invariant graph model. Note that solving to satisfaction or optimality is orthogonal to our purpose.

4.8 Results

We ran our experiments on a desktop computer with an ASUS PRIME Z590-P motherboard, a 3.5 GHz Intel Core i9 11900K processor, and four 16 GB 3200 MT/s DDR4 memories, running Ubuntu 22.04.4 LTS with GCC (the GNU Compiler Collection) 11.

The results are shown in Figure 9.[2] For each invariant graph where the number of edges from static input variables to invariants is greater than the number of edges from dynamic input variables to dynamic invariants by at least one order of magnitude (namely GR, MS, VL, and ES), input-to-output propagation outperforms output-to-input propagation. Conversely, for the invariant graphs where the number of edges from dynamic input variables to dynamic invariants is greater than the number of edges from static input variables to invariants by at least one order of magnitude, output-to-input propagation outperforms input-to-output propagation (TSPTW and ED) for total and prepared marking. For ad-hoc marking, output-to-input propagation also outperforms input-to-output propagation on TSPTW and ED. For output-to-input propagation, on the invariant graph models where input-to-output propagation outperforms output-to-input propagation (namely GR, MS, VL, and ES), ad-hoc marking outperforms prepared marking, which outperforms total marking. Conversely, for the invariant graph models where output-to-input propagation outperforms input-to-output propagation (namely TSPTW and ED), both prepared and total marking outperform ad-hoc marking, where total marking slightly outperforms prepared marking on TSPTW, and total marking and prepared marking have similar performance on ED. These results support our recommendations in Section 3.6 based on Theorems 3.4, 3.5, and 3.6.

5 Conclusion and Future Work

We have detailed and theoretically compared two invariant graph propagation styles, namely input-to-output and output-to-input, and three marking strategies for the latter, namely ad-hoc, prepared, and total. We have presented algorithms for these invariant graph propagation styles, high-level descriptions for these marking strategies, theorems on the time complexities of these algorithms and marking strategies, as well as a recommendation based on those time complexities. Our experiments support that recommendation. There are additional experiments in Appendix A that support that recommendation.

Our current work is the design and implementation of our CBLS solver Atlantis, which supports all propagation styles mentioned here. For future work, we intend to make Atlantis a backend to MiniZinc, in order to run experiments using MiniZinc models.

Note that for Algorithm 1 and in our experiments, when the output-to-input propagation style is used, the values of all probed variables are determined, while only the values of a chosen subset of (possibly non-probed) variables are determined in [13]. For example, consider an invariant graph with multiple probed variables, of which a subset can be chosen. The output-to-input propagation style can be extended by only determining the value of each variable that transitively is a required (static or dynamic) input variable to a chosen variable, skipping all other variables and potentially improving performance. For future work, we intend to implement a selection heuristic that exploits this in our Atlantis solver.

Acknowledgements

Supported by grant 2018-04813 of the Swedish Research Council (VR).

References

[1] S. Attieh, N. Dang, C. Jefferson, I. Miguel, and P. Nightingale. 2019. Athanor: High-level local search over abstract constraint specifications in Essence. In *IJCAI 2019*. S. Kraus, editor. IJCAI Organization, 1056–1063.

[2] S. Attieh, N. Dang, C. Jefferson, I. Miguel, and P. Nightingale. 2025. Athanor: Local search over abstract constraint specifications. *Artificial Intelligence*, 340, (Mar. 2025), 104277.

[3] E. K. Baker. 1983. An exact algorithm for the time-constrained traveling salesman problem. *Operations Research*, 31, 5, (Oct. 1983), 938–945.

[2] The data from the experiments is publicly available at https://github.com/astra-uu-se/atlantis/tree/JAIR-results

[4] T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua. 2011. LocalSolver 1.x: a black-box local-search solver for 0-1 programming. *4OR – A Quarterly Journal of Operations Research*, 9, 3, (Sept. 2011), 299–316. LocalSolver is now called Hexaly and is available at https://www.hexaly.com.
[5] G. Björdal, J.-N. Monette, P. Flener, and J. Pearson. 2015. A constraint-based local search backend for MiniZinc. *Constraints*, 20, 3, (July 2015), 325–345.
[6] R. De Landtsheer and C. Ponsard. 2013. OscaR.cbls: An open source framework for constraint-based local search. In *ORBEL-27, the 27th annual conference of the Belgian Operational Research Society*. Available as https://www.orbel.be/orbel27/pdf/abstract293.pdf; the OscaR.cbls solver is available at https://bitbucket.org/oscarlib/oscar/branch/CBLS.
[7] A. Demers, T. Reps, and T. Teitelbaum. 1981. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL 1981*. ACM, 105–116.
[8] M. M. Flood. 1956. The traveling-salesman problem. *Operations Research*, 4, 1, (Feb. 1956), 61–75.
[9] A. M. Frisch, M. Grum, C. Jefferson, B. Martinez Hernandez, and I. Miguel. 2007. The design of Essence: a constraint language for specifying combinatorial problems. In *IJCAI 2007*. Morgan Kaufmann, 80–87.
[10] I. P. Gent and T. Walsh. 1999. CSPLib: a benchmark library for constraints. In *CP 1999* (LNCS). J. Jaffar, editor. Vol. 1713. CSPLib: A problem library for constraints is available at https://www.csplib.org/. Springer, 480–481.
[11] F. Glover and M. Laguna. 1993. Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, 70–150.
[12] R. Hoover. 1987. *Incremental Graph Evaluation*. PhD thesis. Department of Computer Science, Cornell University, Ithaca, NY, USA.
[13] S. E. Hudson. 1991. Incremental attribute evaluation: a flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13, 3, 315–341.
[14] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. 1983. Optimization by simulated annealing. *Science*, 220, 4598, (May 1983), 671–680.
[15] S. Lin. 1965. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44, 10, (Dec. 1965), 2245–2269.
[16] M. Marte. 2015. Yuck: A local-search constraint solver with FlatZinc interface. Maintained at https://github.com/informarte/yuck/. Originally released as YACS, see https://www.minizinc.org/challenge/2015/description_yacs.txt. (2015).
[17] L. Michel and P. Van Hentenryck. 2000. Localizer. *Constraints*, 5, 1–2, 43–84.
[18] L. Michel and P. Van Hentenryck. 1997. Localizer: A modeling language for local search. In *CP 1997* (LNCS). G. Smolka, editor. Vol. 1330. Springer, 237–251.
[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. 2007. MiniZinc: Towards a standard CP modelling language. In *CP 2007* (LNCS). C. Bessière, editor. Vol. 4741. The MiniZinc toolchain is available at https://www.minizinc.org. Springer, 529–543.
[20] M. H. Newton, D. N. Pham, A. Sattar, and M. Maher. 2011. Kangaroo: An efficient constraint-based local search system using lazy propagation. In *CP 2011* (LNCS). J. Lee, editor. Vol. 6876. Springer, 645–659.
[21] C. Pralet and G. Verfaillie. 2013. Dynamic online planning and scheduling using a static invariant-based evaluation model. In *ICAPS 2013*. D. Borrajo, S. Kambhampati, A. Oddi, and S. Fratini, editors. AAAI Press, 171–179.
[22] T. Reps, T. Teitelbaum, and A. Demers. 1983. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5, 3, (July 1983), 449–477.
[23] P. Van Hentenryck and L. Michel. 2005. *Constraint-Based Local Search*. The MIT Press.
[24] P. Van Hentenryck and L. Michel. 1999. Localizer: A modeling language for local search. *INFORMS Journal on Computing*, 11, 1, (winter 1999), 1–14.
[25] B. Vander Zanden, R. Halterman, B. A. Myers, R. McDaniel, R. Miller, P. Szekely, D. A. Giuse, and D. Kosbie. 2001. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems*, 23, 6, 776–796.
[26] C. Voudouris, R. Dorne, D. Lesaint, and A. Liret. 2001. iOpt: A software toolkit for heuristic search methods. In *CP 2001* (LNCS). T. Walsh, editor. Vol. 2239. Springer, 716–729.

A Additional Experiments

We experiment with four additional invariant graph models that we designed for two classical problems – the travelling salesperson problem [8] and again the travelling salesperson problem with time windows (see Section 4.3) – plus two handmade ones, called sum tree and element tree. We use the same method as in Sections 4.7.

A.1 Travelling Salesperson Problem (TSP)

Consider n locations that are to be visited, where there is a travelling duration $\mu_{u,v}$ between each directed pair (u, v) of locations. A *travelling salesperson tour* (*TSP*) of size n is a Hamiltonian cycle of the weighted directed

graph induced by the n locations as nodes, visiting each location exactly once. The total travelling duration of the tour is to be minimised.

Our invariant graph model for a TSP of size n creates an invariant graph with the array $\mathcal{S} = [x_1, \ldots, x_n]$ of n search variables, one for each location where x_u denotes the location that is visited just before location u. For each location $1 \leq u \leq n$, the invariant graph model creates the variable t_u that denotes the travelling duration from location x_u to location u and the static Element invariant $t_u \Longleftarrow [\mu_{1,u}, \ldots, \mu_{n,u}][x_u]$ that defines t_u; note that the invariant is static as each $\mu_{i,u}$ is a parameter. Additionally, the invariant graph model creates the static Sum invariant $o \Longleftarrow \sum_{i=1}^{n} t_i$ that defines the probed objective variable o, which denotes the total travelling duration of the tour. As for TSPTW (see Section 4.3), an implicit constraint maintains that we always have a Hamiltonian cycle during search; after such an initialisation, a probe is the execution of a 3-opt [15] on the vector of search variables.

The number of edges in the invariant graph between invariants and their static input variables is $2 \cdot n$. There are no dynamic invariants. The number of marked variables for any probe with ad-hoc marking is 7. When the output-to-input propagation style is used, up to $2 \cdot n$ edges are followed.

A.2 Travelling Salesperson Problem with Time Windows, revisited (TSPTW)

We design another invariant graph model for the TSPTW problem of Section 4.3. Our TSPTWS invariant graph model for a TSPTW of size n creates an invariant graph with the array $\mathcal{S} = [x_1, \ldots, x_n]$ of n search variables, one for each location, where x_u denotes the u^{th} location that is visited in the tour. For each index $1 \leq i \leq n$ in the sequence, the invariant graph model creates:

- the variable e'_i that denotes the earliest visiting time of location x_i;
- the variable ℓ'_i that denotes the latest visiting time of location x_i;
- the variable d_i that denotes the departure time from location x_i;
- the static Element invariant $e'_i \Longleftarrow [e_1, \ldots, e_n][x_i]$ that defines e'_i; note that the invariant is static as each e_j is a parameter; and
- the static Element invariant $\ell'_i \Longleftarrow [\ell_1, \ldots, \ell_n][x_i]$ that defines ℓ'_i; note that the invariant is static as each ℓ_j is a parameter.

For the first visited location, we have $d_1 = e'_1$. For each index $2 \leq i \leq n$ in the sequence, the invariant graph model creates:

- the variable t_i that denotes the travelling duration from location x_{i-1} to location x_i;
- the static Element invariant $t_i \Longleftarrow \left[[\mu_{1,1}, \ldots, \mu_{1,n}], \ldots, [\mu_{n,1}, \ldots, \mu_{n,n}]\right][x_{i-1}][x_i]$ that defines t_i and takes a two-dimensional matrix of parameters and two static variables as inputs; note that the invariant is static as each $\mu_{j,k}$ is a parameter;
- the variable a_i that denotes the arrival time at location x_i;
- the static Sum invariant $a_i \Longleftarrow d_{i-1} + t_i$ that defines a_i;
- the static invariant $d_i \Longleftarrow \max(a_i, e'_i)$ that defines the remaining d_i;
- the violation variable v_i that denotes the lateness (possibly zero) of arrival at location x_i compared to ℓ'_i; and
- the static violation invariant $v_i \Longleftarrow \max(0, a_i - \ell'_i)$ that defines v_i.

Note that for each $2 \leq i \leq n$, the variable d_i depends on a_i, which in turn depends on d_{i-1}, with $d_1 = e'_1$. Additionally, the invariant graph model creates the static invariant $\textsc{Sum}(\{v_2, \ldots, v_n\}, v)$ that defines the probed total-violation variable v. The probed objective variable, which denotes the departure time from the last location of the tour, is d_n. An implicit constraint maintains the n search variables to be in $\{1, \ldots, n\}$ and take distinct values: after such an initialisation, a probe consists of swapping two search variables.

The number of edges in the invariant graph between invariants and their static input variables is $11 \cdot n - 9$. There are no dynamic invariants. The number of marked variables for any probe with ad-hoc marking is 15. When the output-to-input propagation style is used, up to $11 \cdot n - 9$ edges are followed.

A.3 Sum Tree (ST)

Our *Sum Tree* invariant graph model for size n creates an acyclic invariant graph with:

- the sets $\mathcal{X}_0$, $\mathcal{X}_2$, $\mathcal{X}_4$, $\mathcal{X}_6$, and $\mathcal{X}_8$ of n^4 variables, n^3 variables, n^2 variables, n variables, and 1 variable respectively, with $x_{j,k} \in \mathcal{X}_j$; and
- the sets $\mathcal{I}_1$, $\mathcal{I}_3$, $\mathcal{I}_5$, and $\mathcal{I}_7$ of n^3 invariants, n^2 invariants, n invariants, and 1 invariant respectively, where each invariant $\mathbb{I}_{j,k}$ in $\mathcal{I}_j$ is a static Sum invariant that takes its unique n input variables from $\mathcal{X}_{j-1}$ and defines one output variable in $\mathcal{X}_{j+1}$.

The search variables are $\mathcal{X}_0$ and the only probed variable is $x_{8,1}$. The resulting acyclic invariant graph for size $n = 2$ is in Figure 10. We initialise the value of each variable in $\mathcal{X}_0$ to a random value. A probe consists of randomly updating some search variable in $\mathcal{X}_0$.

The number of edges between invariants and their static input variables is $n^4 + n^3 + n^2 + n$. The number of marked variables for any probe with ad-hoc marking is 5. When the output-to-input propagation style is used, up to $n^4 + n^3 + n^2 + n$ edges are followed.

We designed this invariant graph model so that the probed variable is transitively defined by a number of static invariants that increases polynomially in n. Additionally, the number of edges that are followed during propagation is constant for input-to-output style, but polynomial in n for output-to-input style.

A.4 Element Tree (ET)

Our *Element Tree* invariant graph model for size n creates an acyclic invariant graph with:

- the array $\mathcal{X}_0$ of n search variables $x_{0,k}$;
- the sets $\mathcal{X}_2, \mathcal{X}_4, \mathcal{X}_6$, and $\mathcal{X}_8$ of n^3 variables, n^2 variables, n variables, and 1 variable respectively, with $x_{j,k} \in \mathcal{X}_j$;
- the sets $\mathcal{Y}_0$, $\mathcal{Y}_2$, $\mathcal{Y}_4$, and $\mathcal{Y}_6$ of n^3 search variables, n^2 search variables, n search variables, and 1 search variable respectively; and
- the sets $\mathcal{I}_1$, $\mathcal{I}_3$, $\mathcal{I}_5$, and $\mathcal{I}_7$ of n^3 invariants, n^2 invariants, n invariants, and 1 invariant respectively, where each invariant $\mathbb{I}_{j,k}$ in $\mathcal{I}_j$ is a dynamic Element invariant, where:
  - it takes its unique static input variable from $\mathcal{Y}_{j-1}$;
  - if $j = 1$, then it takes $\mathcal{X}_0$ as its dynamic input variables, otherwise it takes its unique n dynamic input variables from $\mathcal{X}_{j-1}$; and
  - it defines one output variable in $\mathcal{X}_{j+1}$.

The only probed variable is $x_{8,1}$. The resulting acyclic invariant graph for size $n = 2$ is in Figure 11. We initialise the values of the variables in $\mathcal{Y}_j$ with $j \in \{0, 2, 4, 6\}$ such that each variable in $\mathcal{X}_0$ transitively is a required dynamic input variable under the corresponding valuation to the invariant that defines $x_{8,1}$. A probe consists of updating some search variable in $\mathcal{X}_0$, which is a dynamic input variable. We opted not to allow probes where any variable in $\mathcal{Y}_j$ with $j \in \{0, 2, 4, 6\}$ is updated, as the cost of such a probe might be different from a probe where some search variable in $\mathcal{X}_0$ is updated.

The number of edges between invariants and their static input variables is $n^3 + n^2 + n + 1$. The number of edges between invariants and their dynamic input variables is $n^4 + n^3 + n^2 + n$. The number of marked variables for any probe with ad-hoc marking is $n^3 + n^2 + n + 2$. When the output-to-input propagation style is used, up to 8 edges are followed.

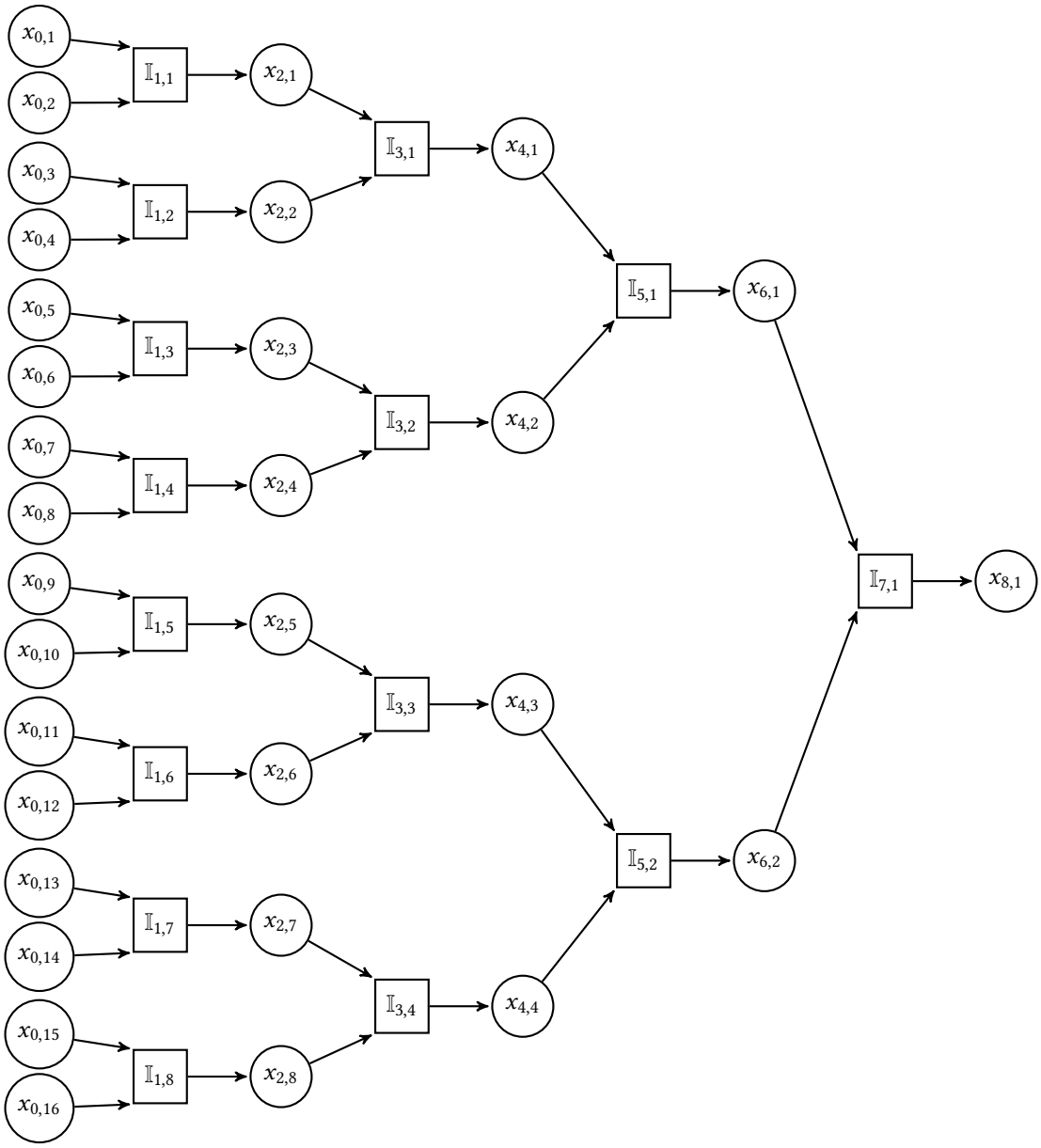

Fig. 10. Invariant graph for a sum tree of size $n = 2$, where each invariant $\mathbb{I}_{j,k}$ is a static Sum invariant. Each solid edge goes from a static input variable to an invariant (which is static in the case of this figure).

We designed this invariant graph model so that the probed variable is transitively defined by a number of dynamic invariants that increases polynomially in n. Additionally, the number of edges that are followed during propagation is constant for output-to-input style, but polynomial in n for output-to-input style.

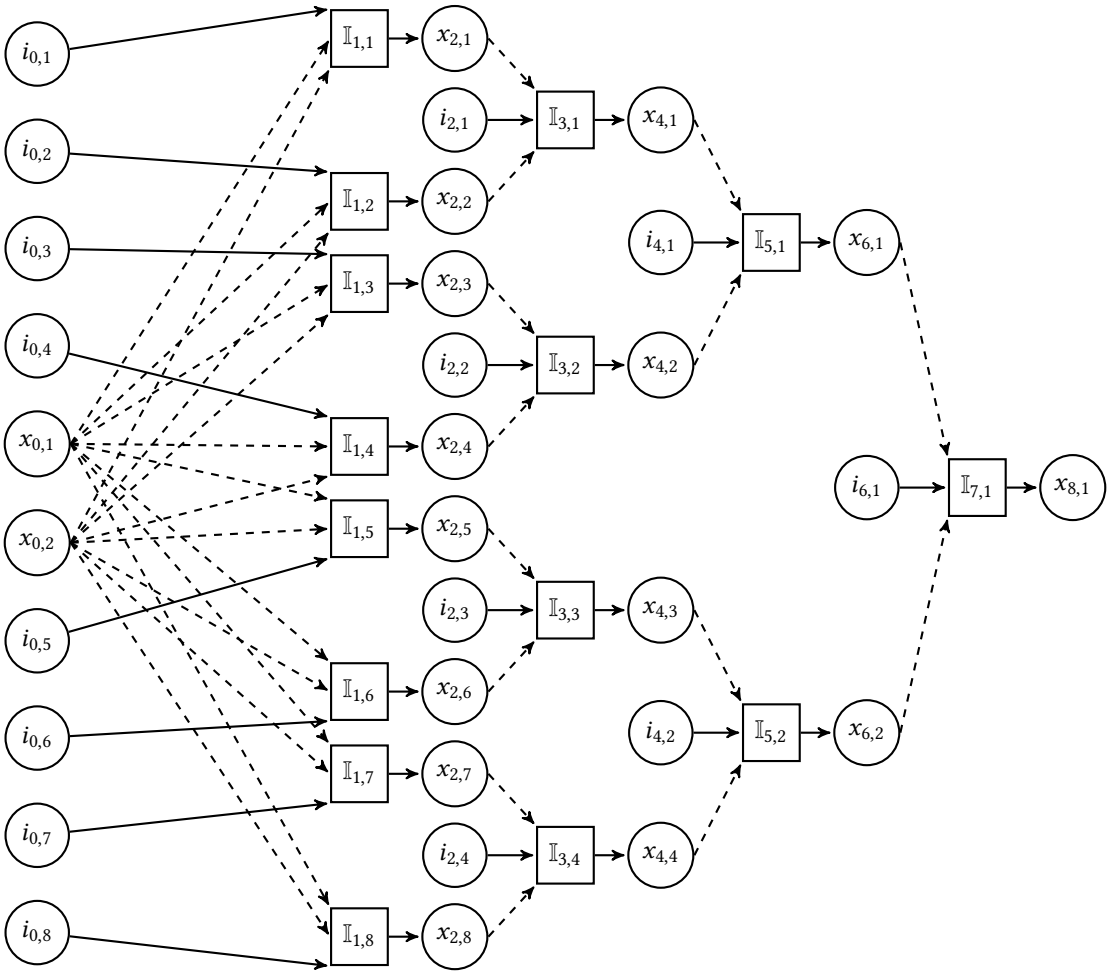

Fig. 11. Invariant graph for an element tree of size $n = 2$, where each $\mathbb{I}_{j,k}$ is a dynamic Element invariant. Each dashed edge goes from a dynamic input variable to a dynamic invariant. Each solid edge goes from a static input variable to an invariant (which is dynamic in the case of this figure).

A.5 Results

We ran our experiments on a desktop computer with an ASUS PRIME Z590-P motherboard, a 3.5 GHz Intel Core i9 11900K processor, and four 16 GB 3200 MT/s DDR4 memories, running Ubuntu 22.04.4 LTS with GCC (the GNU Compiler Collection) 11.

The results are shown in Figure 12. For each invariant graph where the number of edges from static input variables to invariants is greater than the number of edges from dynamic input variables to dynamic invariants by at least one order of magnitude (namely TSP, TSPTWS, and ST), input-to-output propagation outperforms output-to-input propagation. Conversely, for the invariant graph where the number of edges from dynamic input variables to dynamic invariants is greater than the number of edges from static input variables to invariants by at least one order of magnitude (namely ET), output-to-input propagation outperforms input-to-output propagation

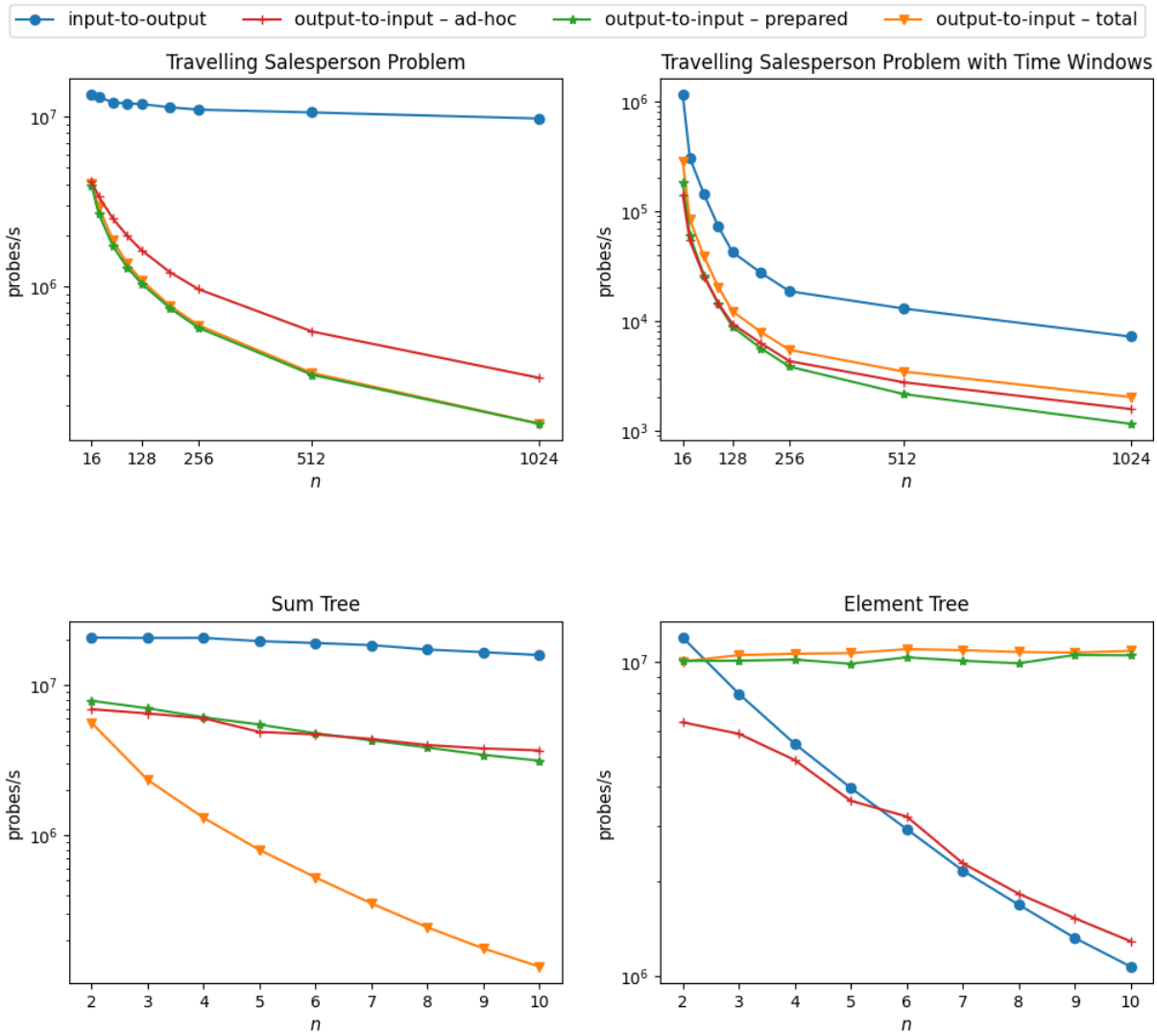

Fig. 12. The number of probes per second for various invariant graphs with input-to-output and output-to-input propagation, where output-to-input propagation uses either ad-hoc marking, or prepared marking, or total marking. The horizontal axis of each graph corresponds to the size of the invariant graph model. Note that solving to satisfaction or optimality is orthogonal to our purpose.

for total and prepared marking. For ad-hoc marking, output-to-input propagation has slightly better performance than input-to-output propagation on ET when the number of dynamic input variables to each invariant is at least 6 corresponding to $n \geq 6$ in Section A.4, but has worse performance for $n < 6$. For invariant graph models where input-to-output propagation outperforms output-to-input propagation (namely TSP, TSPTWS, and ST): ad-hoc marking outperforms prepared marking and total marking on TSP, which have similar performance; total marking outperforms ad-hoc marking, which outperforms prepared marking, on TSPTWS; and ad-hoc and prepared

marking have similar performance on ST, and both outperform total marking. For output-to-input propagation, on the invariant graph model where output-to-input propagation outperforms input-to-output propagation (namely ET), total marking has slightly better performance than prepared marking, which outperforms ad-hoc marking. These results support our recommendations in Section 3.6 based on Theorems 3.4, 3.5, and 3.6.

Received 25 October 2024; revised 22 April 2025; accepted 7 May 2025