JuLS : A Julia Local Search solver

Axel Navarro axelnav@amazon.com
Arthur Dupuis dupuisar@amazon.com
Ilan Coulon ilan.coulon@gmail.com
Ezra Reich rezra@amazon.com
Mehdi Oudaoud mehdoud@amazon.com
Maxime Mulamba mulmaxim@amazon.com

Abstract

JuLS is a lightweight, open source and modular optimization solver that combines Constraint-Based Local Search and Constraint Programming to handle both generic and black-box constraints and objectives. Currently deployed in Amazon's production environment, it successfully addresses complex logistics and scheduling challenges in middle-mile operations. Developed by scientists for scientists, this generic solver framework welcomes contributions and supports the modeling of a wide range of optimization problems. Its adaptable architecture and open design philosophy make it an ideal platform for both academic research and industrial applications. Backtests between JuLS and OR-Tools demonstrate comparable performance on standard optimization tasks. However, JuLS distinguishes itself through its seamless integration capabilities with external tools, particularly excelling in handling black-box constraints and objectives. This unique feature positions JuLS as a versatile solution for complex, real-world optimization scenarios that often require interfacing with proprietary or external systems.

1 Framework

1.1 Constrained Optimization Problem

A *Constrained Optimization Problem* (COP) is represented with a tuple (X, D, C, f), where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a domain set, such that $D(x)$ is a finite set of potential values for x, $C = \{c_1, \ldots, c_m\}$ is a set of constraints and $f : D(x_1) \times \ldots \times D(x_n) \rightarrow \mathbb{R}$ is an objective function to be minimized. An **assignment** (or a solution) $\sigma = (v_1, \ldots, v_n) \in D$ is a sequence where v_i (or $\sigma(x_i)$) denotes the value of variable x_i drawn from $D(x_i)$. A constraint is an arbitrary function $c_i : \mathbb{R}^n \rightarrow \{0, 1\}$ that determines whether an assignment σ satisfies the constraint ($c_i(\sigma) = 1$) or not. A feasible solution is an assignment that satisfies all m constraints in C.

Applying changes to an assignment involves modifying a subset of variables, referred to as **decision variables**, noted $dec(X) \subset X$. Other variables are useful to either instantiate the constraints or the objective function and are called **intermediate variables**.

1.2 Constraint-Based Local Search

Constraint Based Local Search (CBLS) [4] is a problem-solving approach that combines local search [1] techniques with a clear separation between problem modeling and search strategies. A CBLS solver is built upon three essential components: a *neighbourhood heuristic*, a *move evaluator*, and a *move selection heuristic*. This tripartite structure provides an interface allowing to express problems declaratively in a specific modeling language, while tuning the optimization strategy.

Definition 1.1 (Move)**.** Given a COP (X, D, C, f) where $dec(X) = \{x_1, \ldots, x_k\}$ is the set of decision variables, a move m is a set of changes to the current assignment σ. Each change is

represented by a tuple (x, v), where $x \in dec(X)$ is a decision variable and $v \in D(x)$ is a new value for x. Formally,

$$m = \{(x, v) \mid x \in dec(X), v \in D(x), \text{ and } v \neq \sigma(x)\}$$

The application of a move m to an assignment σ results in a new assignment σ', where :

$$\forall x \in dec(X) \qquad \sigma'(x) = \begin{cases} v & \text{if } (x, v) \in m \\ \sigma(x) & \text{otherwise} \end{cases}$$

A **neighbourhood** is a function that maps an assignment σ to a set of assignments that are close to σ, denoted by $\mathcal{N}(\sigma)$. The measure of how close two assignments are is typically the number of decision variables that need to be reassigned [3]. In practice, a neighbourhood heuristic is a function deciding which set of moves should be evaluated during the next step.

A **move evaluator** is a function that is responsible for evaluating the impact of a move on the feasibility and objective of an assignment, i.e compute $f(\sigma')$ and $c_i(\sigma')$ for $i \in [m]$. In practice, the move evaluator is used to evaluate all the moves of a neighbourhood.

A **move selection heuristic** is a function that picks a move amongst the neighbourhood based on their evaluation by the move evaluator. This move is then applied to the model assignment σ as well as to the move evaluator, and the next iteration can start again with a new state.

The main contribution of this paper is to present a solver incorporating an efficient move evaluator by combining two different paradigms detailed in the next sections.

2 Methodology

2.1 CBLS propagation

2.1.1 Invariant

To evaluate the impact of a move m on feasibility and objective, changes applied to decision variables must propagate to the problem's intermediate variables. This process is called **CBLS propagation** [10] [2] [9] and relies on a specific declaration of constraints, the **invariants**. The idea behind this propagation mechanism is to leverage the unique structure of optimization problems by only assessing the specific intermediate variables impacted by a move.

Definition 2.1 (Invariant)**.** An invariant I is a fixed relation between a set of input variables $\mathcal{X} \subset X$ and a single output variable $y \in X$. Derived notations are $y = def(I)$ and $\mathcal{X} = exp(I)$.

An invariant allows to determine the value of its output variable $def(I)$ based on the values taken by its input variables $exp(I)$. As a pedagogical example, an invariant I can represent the constraint $z = x + y$ with $def(I) = z$ and $exp(I) = \{x, y\}$. An invariant can also represent a violation, for instance the constraint $x \leq B$ can be represented by an invariant maintaining the relation $y = max\{0, x - B\}$. In this case y is a violation variable which is a mathematical programming representation of a constraint. A key benefit of this modeling approach is its flexibility in handling constraints. Whether dealing with standard mathematical constraints or domain-specific business rules, any constraint can be implemented as an invariant. The only requirement is that $def(I)$ must be computable from the values of $exp(I)$.

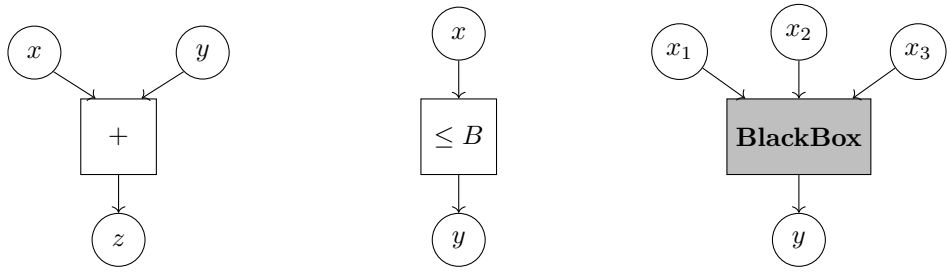

Figure 1: Graphical representation of invariants (rectangles), variables (circles) and their dependencies (arcs)

2.1.2 Graph structure

Let $\mathcal{I}$ be the set of invariants representing the constraints C and objective function f for our COP (X, D, C, f). The dependencies between variables and invariants create a Directed Acylic Graph (DAG) G as illustrated in Figure 2. Under this assumption, we can derive from G a topological ordering of the invariants $(I_1, I_2, \ldots, I_N)$ using Kahn's algorithm [7].

This ordering ensures that for two invariants I_i and I_j such that $i < j$, the intermediate variable $def(I_i)$ does not depend on $def(I_j)$. Thus, by applying the CBLS propagation algorithm detailed in Figure 2 to the order $(I_1, \ldots, I_N)$, we can update all intermediate variables impacted by a move and thereby evaluate its impact on feasibility and the objective.

Algorithm 1: CBLS propagation

Data: Move m, Ordered invariants $\mathcal{I} = (I_1, \ldots, I_N)$
foreach *tuple* (x, v) *in move* m **do**
  Update variable x with new value v
  Mark every child invariant I of x as touched
while $\exists$ *touched invariants remaining in* $\mathcal{I}$ **do**
  $I \leftarrow$ next touched invariant in $\mathcal{I}$
  $y \leftarrow def(I)$
  Update y value with values from $exp(I)$
  if y *is a violation variable and* $y > 0$ **then**
    return $+\infty$ (move infeasible)
  foreach *child invariant* I *of* y **do**
    Mark I as touched;
return last invariant result;

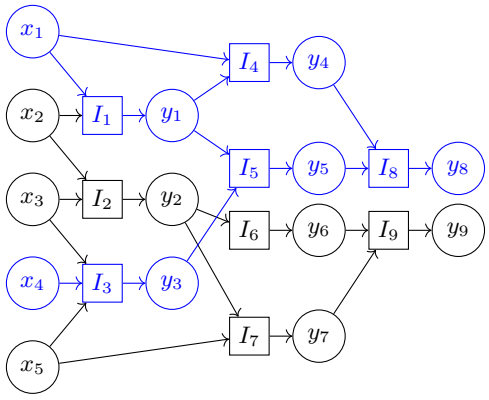

Figure 2: CBLS propagation algorithm and graphical example with touched invariants and updates variables (blue) if decision variables moved are x_1 and x_4

The topological ordering $(I_1, \ldots, I_N)$ on G ensures that for each invariant I considered, the intermediate variable $y = def(I)$ can be updated since all parent variables $\mathcal{X} = exp(I)$ have been updated beforehand. An infeasible move is identified by a violation variable that takes a positive value. In this case, the algorithm returns an infinite objective. The Algorithm 1 limits the number of intermediate variables to update during the evaluation of a move, thus quickly assessing its impact on the solution.

2.2 CP enumeration

Constraint Programming (CP) is a paradigm for solving combinatorial problems by expressing them as a set of constraints over variables. To find a feasible assignment, a CP solver systematically explores the solution space, using a tree search coupled with **CP propagation functions** defined below.

Definition 2.2 (CP Propagation Function)**.** A CP propagation function for a constraint c is a function f_c that prunes a set of domain D removing values that cannot be part of any solution satisfying c.

In Constraint Programming, the tree search is branching on values from variable domains. Removing values with propagation function allows to greatly reduce the number of branches to explore by omitting parts of the tree that would only lead to infeasible solutions. To maximize the domain pruning at each node, we are using the **Fix-point algorithm** detailed in Appendix A. By completing the whole tree search, we can compute an exhaustive list of feasible solutions, satisfying the constraints propagated during the search. This process is called **CP enumeration**.

In a local search context, we consider a given solution σ and a selected subset of decision variables $\hat{X} \subset dec(X)$ to be moved. By fixing non-selected decision variables to their values in σ beforehand, a CP enumeration provides us with an exhaustive list of feasible solutions where only variables in $\hat{X}$ can differ from their values in σ. This list therefore corresponds to all feasible moves for $\hat{X}$.

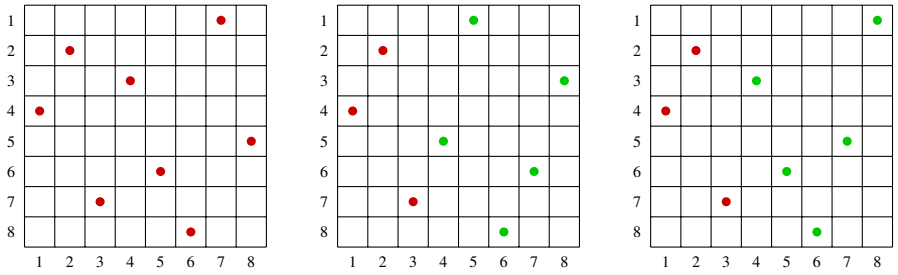

Figure 3: 8-queens solution σ and feasible moves m_1, m_2 if variables $\{x_1, x_2, x_3\}$ are fixed

For example, a classical problem solved in CP is the 8-queens problem [8], it consists in placing eight chess queens on an 8x8 board. The problem's constraints dictate that no two queens can threaten each other. In this case, we have 8 decision variables $(x_1, \ldots, x_8)$, where x_i defines the position of the queen in column i. Referring to Figure 3, if we start with the solution σ on the left and aim to evaluate all possible moves for the variables $\hat{X} = \{x_4, x_5, x_6, x_7, x_8\}$, we would need to assess $8^5 = 32,768$ potential moves. However, through CP enumeration, we can swiftly determine that only two of these moves are feasible (m_1 and m_2). This allows us to focus our CBLS propagation solely on these two viable moves, significantly reducing the computational effort .

2.3 Combining CBLS propagation and CP enumeration

To combine CBLS propagation and CP enumeration in our move evaluation process we have to split invariants into two categories: **Black-box invariants** and **CP invariants**. Black-box invariants are typically invariants that rely on complex algorithms while CP invariants are typically classical constraints such as an invariant making sure that a set of variables must remain all different.

It is therefore possible to partition the set of invariants $\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2$ with $\mathcal{I}_1$ as CP invariants and $\mathcal{I}_2$ as Black-box invariants. This partition leads to a new optimization framework shown in 4. The idea of this framework is to leverage CP enumeration for CP invariants and CBLS propagation for Black-box invariants. At each iteration, we can select with a neighbourhood heuristic a subset of decision variables $\hat{X} \subset dec(X)$, for which we want to make new decisions. Then, using CP enumeration, we can return the feasible moves for $\hat{X}$ satisfying the CP invariants $\mathcal{I}_1$. These moves can then be evaluated through CBLS propagation of the Black-box invariants $\mathcal{I}_2$. The move selection heuristic can then select the best move to apply m^* according to its predefined criteria and update the solution σ with the selected move.

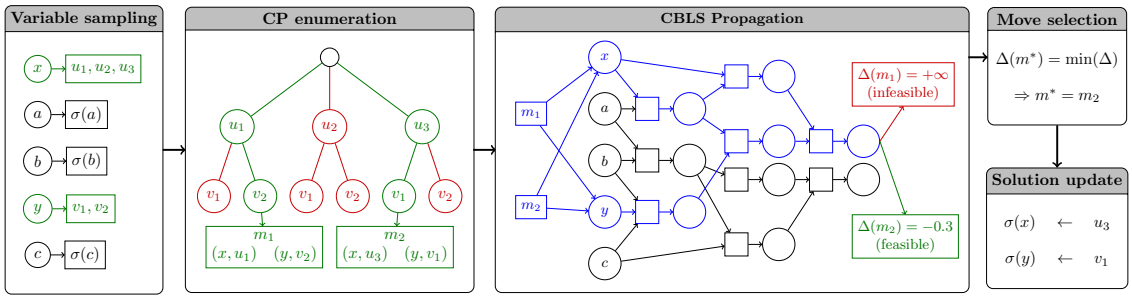

Figure 4: Green and red respectively represent feasible and infeasible moves, built from relaxing the decision variables x, y. During CBLS Propagation, only the black-box invariants in blue are involved in the evaluation of candidate moves m_1, m_2. In this example, m_1 is not filtered out by CP enumeration, but does not satisfy black-box constraints, hence its objective value of $+\infty$

3 Implementation

3.1 Model

The implementation of a model for optimizing a given Constraint Optimization Problem (COP) requires the following fields:

- `neighbourhood_heuristic` A heuristic that defines how to explore the solution space. The traditional approach to use CP enumeration is listing all possible moves when relaxing a subset of decision variables.
- `move_filter` : A move filter which can be implemented using CP enumeration, sampling methods, or a combination of both.
- `dag` : A Directed Acyclic Graph (DAG) that captures all problem invariants and their inter-dependencies.
- `move_selection_heuristic` : A heuristic that determines which move to choose from the evaluated options. Traditionally, this selects the feasible move that produces the greatest impact.

Here is the pseudo code for one iteration :

```
1  moves = get_potential_moves(model.neighbourhood_heuristic)
2  filtered_moves = filter_moves(model.move_filter, moves)
3  Threads.@threads for (i, move) in filtered_moves
4      evaluated_moves[i] = eval(model.dag, move)
5  end
6  picked_move = pick_a_move(model.move_selection_heuristic, evaluated_moves)
7  apply_move!(model, picked_move)
```

For efficiency purposes, CBLS propagations (line 4) are executed in parallel. This parallel execution necessitates a **stateless evaluation** of the DAG - the invariant states must remain unchanged during the evaluation phase. Instead, state updates occur only when `apply_move!` is called.

There are two ways to perform a stateless evaluation in the DAG. The first one is used during optimization, called `DeltaRun`. Invariants impacted by a move receive input objects of type `Delta` and propagate this impact to subsequent invariants only if the output variable is affected by these changes. The second use of the DAG is the complete evaluation of a solution, called `FullRun`. Here, all invariants of the DAG are propagated (in the same topological order) to evaluate the objective and feasibility of any given solution.

3.2 Invariant

To implement an invariant `inv <:Invariant`, four functions must be defined:

- `eval(inv, <:Delta)`: A stateless evaluation to propagate the impact of changes in input variables $exp(\texttt{inv})$ on the output variable $def(\texttt{inv})$.
- `commit!(inv, <:Delta)`: Updates the invariant's state based on changes propagated by parent invariants.
- `eval(inv, <:FullMessage)`: A stateless evaluation of a complete solution to compute the value of output variable $def(\texttt{inv})$ according to the values of input variables $exp(\texttt{inv})$. This evaluation must be entirely independent of the current state of our invariants.
- `init!(inv, <:FullMessage)`: Initializes the invariant's state based on the initial values of the input variables.

Why do we need to maintain a state for invariants ? Let's take the example of a simple capacity invariant $x \leq B$ represented by the relation $y = max\{0, x - B\}$. During a `DeltaRun`, the invariant only receives a value difference δ_x of the input variable x. Here, it's impossible to compute the violation y without knowing the current value of x before this change. This value is stored as a state in the invariant structure and enables efficient evaluation.

Invariants are then added to our DAG using the `add_invariant!` function, specifying their parent invariants. The topological ordering of invariants is automatically performed during initialization once all invariants have been added to the DAG.

3.3 CP model from DAG structure

Once the initial step of modeling the COP as a DAG of invariants is complete, it becomes possible to convert a subset of this DAG into a Constraint Programming (CP) model. This conversion requires defining a `build!` function that transforms a CP invariant I and its variable $def(I)$ into a CP constraint. For instance, the invariant representing a capacity constraint through the violation relation $y = max(0, \sum_{i=1}^{n} x_i - B)$ can be converted into a generic CP constraint `SumLessThan`$(\{x_i\}_{i=1}^{n}, B)$.

During the DAG construction process, one can specify which invariants should be converted to CP constraints by adding the `using_cp = true` parameter to the `add_invariant!` call. This selection creates an induced subgraph $\tilde{G}$ of invariants and variables to be converted to CP. Figure 5 illustrates an example DAG for the knapsack problem, with the induced subgraph $\tilde{G}$ highlighted in green. To ensure a valid conversion of $\tilde{G}$ to CP, all parents of nodes in $\tilde{G}$ must also be included in $\tilde{G}$. Consequently, $\tilde{G}$ forms a DAG that encompasses all decision variables.

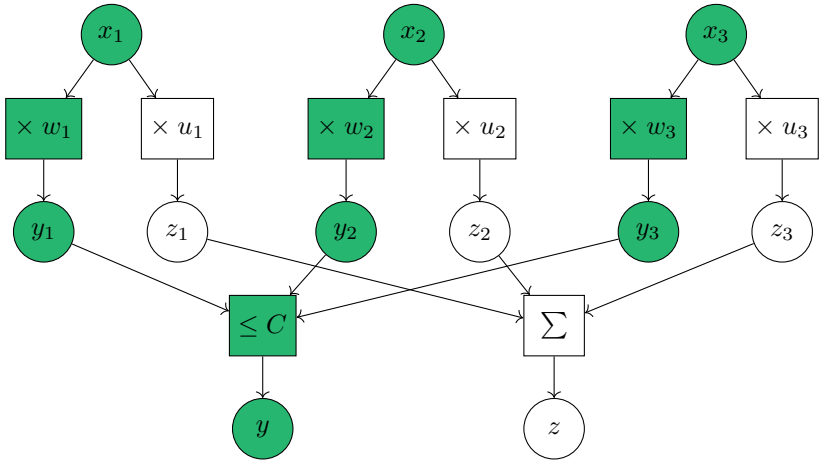

Figure 5: Invariants DAG for Knapsack Problem with weight w, value u and CP conversion for the weighted capacity constraint (green)

4 Experiments

JuLS's primary strength lies in its ability to handle problems with Black-Box constraints and objectives, making it challenging to showcase this advantage on well-known generic problems. The purpose of these experiments is to demonstrate the solver's modularity and its capability to find quality solutions for certain generic problems within fixed time frames, comparing it with OR-tools [11]. The real potential of JuLS becomes apparent when applied to ad-hoc problems requiring external tool integration, which is left to the user's specific applications. The solvers were compared on the same machine (48 vCPU, 192GB of RAM).

For the solver benchmark, we consider :

- **Knapsack Problem** : instances sampled from Joris Jooken generator [6].
- **Travelling Salesman Problem** (TSP) : TSPLIB dataset [12]
- **Graph Coloring Problem** (GCP) : DIMACS challenge instances [5]

The invariants DAG invariants for these problems are depicted Figure 5, 6 and 7. For each problem, we compare both solvers' performance within 1 minute of running time. At the end of the running time, we measure the gap between both solutions.

4.1 Knapsack Problem

The Knapsack Problem serves as an excellent case study to showcase the advantages of combining Constraint Programming (CP) and Local Search (LS) methodologies. Our approach employs an exhaustive neighborhood heuristic examining n variables concurrently, which theoretically requires evaluating 2^n moves per iteration. However, by implementing a move filter based on CP enumeration, we substantially reduce the search space by eliminating infeasible moves before evaluation illustrated by Table 1.

Table 1: CP enumeration average performance on Knapsack problem

Size (n)	**Potential moves** (2^n)	**Filtered moves**	**Percentage**	**Time**
10	1 024	22	2.1	2.9 ms
15	32 768	65	0.2	2.9 ms
20	1 048 576	364	0.03	10 ms

Our experimental setup utilized a greedy initialization, followed by an exhaustive neighborhood search examining $n = 10$ randomly chosen variables per iteration. The move selection is greedy, meaning the best feasible move is picked at each iteration.

Number of items	**Average gap (%)**
400	0.04
600	0.08
800	0.05
1000	0.01
1200	0.04

Table 2: Average gaps (%) between JuLS and OR-tools for different knapsack instance size obtained in 1 minute of running time

4.2 Traveling Salesman Problem

The DAG modeling of TSP is detailed in Appendix B. The primary constraint involves maintaining distinctness among all decision variables. To evaluate only feasible solutions, we employ a k-OPT neighborhood heuristic that evaluates permutations of solution subsets. Our implementation uses $k = 2$ for the neighborhood structure, combined with simulated annealing using parameters $T = 1.0$, $\alpha = 0.99$, and $T_{min} = 0.1$.

Number of nodes	**Average gap (%)**
51-100	14.90
101-200	14.92
201-500	22.75
501-1000	32.30
1001-5000	25.88

Table 3: Average gaps (%) between JuLS and OR-tools for different TSP instance size obtained in 1 minute of running time

For the TSP, JuLS's performance lags behind OR-tools due to its DAG structure containing numerous invariants ($O(n^2)$), which makes evaluation costly. Matching OR-tools' performance would require implementing specific invariants to integrate their highly efficient routing algorithm.

4.3 Graph Coloring Problem

The DAG modeling of GCP is detailed in Appendix C. Our neighbourhood heuristic is considering a random vertex at each iteration and sample uniformly a new color for this node. The move selection heuristic is a simulated annealing using parameters $T = 100$, $\alpha = 0.9999$, and $T_{min} = 0.01$.

Instance name	JuLS	OR-Tools	SOTA
DSJC250.5	35	35	28
DSJC500.1	15	15	12
DSJC500.5	71	126	48
DSJC500.9	169	×	126
DSJC1000.1	29	26	20
DSJC1000.5	121	×	83
DSJC1000.9	313	×	222
R250.5.col	70	67	65
R1000.1c.col	114	×	98
R1000.5.col	259	×	234
DSJR500.1c	100	×	85
DSJR500.5	134	204	122
le450_25c.col	29	28	25
le450_25d.col	30	27	25
flat300_28_0.col	40	40	29

Table 4: Comparison of SOTA objective (best number of colors) for different GCP instance obtained in 1 minute of running time for JuLS and OR-Tools

A key advantage of JuLS here lies in its nature as a local search solver, ensuring the provision of feasible solutions even for large-scale graph.

Conclusion

JuLS demonstrates competitive performance comparable to OR-tools for some classic optimization problems. However, the true strength of JuLS emerges in its seamless integration with external blackbox constraints and objectives, a feature particularly valuable in industrial applications. This unique capability positions JuLS as the only open-source library currently offering such functionality, addressing a critical need in complex, real-world optimization scenarios.

It is important to note that the effectiveness of JuLS heavily relies on the modeling of the problem as a Directed Acyclic Graph (DAG) of invariants. The structure and composition of this DAG can significantly impact JuLS's performance. A crucial aspect of this modeling process is the representation of constraints in a generic Constraint Programming (CP) form, which allows JuLS to leverage powerful filtering techniques.

In essence, JuLS combines the reliability of Local Search and Constraint Programming methods with the flexibility to handle blackbox constraints, making it a powerful tool for both academic and industrial optimization challenges. As the field of optimization continues to evolve, JuLS stands out as a versatile and robust solution, particularly suited for complex problems that require integration with external systems or proprietary objective functions.

References

[1] Aldair Alvarez, Pedro Munari, and Reinaldo Morabito. Iterated local search and simulated annealing algorithms for the inventory routing problem. *International Transactions in Operational Research*, 25(6):1785–1809, 2018.

[2] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. Localsolver 1. x: a black-box local-search solver for 0-1 programming. *4or*, 9(3):299–316, 2011.

[3] Diptesh Ghosh. Neighborhood search heuristics for the uncapacitated facility location problem. *European Journal of Operational Research*, 150(1):150–162, 2003. O.R. Applied to Health Services.

[4] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT press, 2009.

[5] David S. Johnson and Michael A. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[6] Jorik Jooken, Pieter Leyman, and Patrick De Causmaecker. A new class of hard problem instances for the 0–1 knapsack problem. *European Journal of Operational Research*, 301(3):841–854, 2022.

[7] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

[8] Craig Letavec and John Ruggiero. The n-queens problem. *INFORMS Trans. Edu.*, 2(3):101–103, May 2002.

[9] Laurent Michel and Pascal Van Hentenryck. Constraint-based local search. In *Handbook of Heuristics*, pages 1–38. Springer, 2017.

[10] Aina Niemetz, Mathias Preiner, and Armin Biere. Propagation based local search for bit-precise reasoning. 51(3), 2017.

[11] Laurent Perron. Or-tools.

[12] Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

A Fix-point algorithm

To reduce the size of the domains, we apply the **Fix-Point algorithm**. This algorithm first applies the CP propagation function of the problem's invariants using a queue.

For a given invariant I, if its propagation function f_I has reduced the domain of a variable x, then we add to the queue all the invariants impacted by x noted $inv(x) = \{I \in \mathcal{I} \mid x = def(I) \text{ or } x \in exp(I)\}$

Algorithm 2: Fix-point algorithm

Data: Variables X, Domains D^0, Invariants $\mathcal{I}$
Result: Smallest possible domain set D

```
Q ← I
D ← D^0
while |Q| > 0 do
    I ← dequeue(Q)
    D' ← f_I(D)
    for x ∈ exp(I) ∪ {def(I)} do
        if D(x) ≠ D'(x) then
            enqueue(Q, inv(x))
return D
```

This algorithm is then used within a tree search. This search is performed with a **Depth-First-Search** (DFS) coupled with domain pruning by Fix-Point at each node. During the search, if all variables are assigned, it means we have found a feasible assignment. Branching on variables and values is done heuristically.

B Invariant DAG for Travelling Salesman Problem

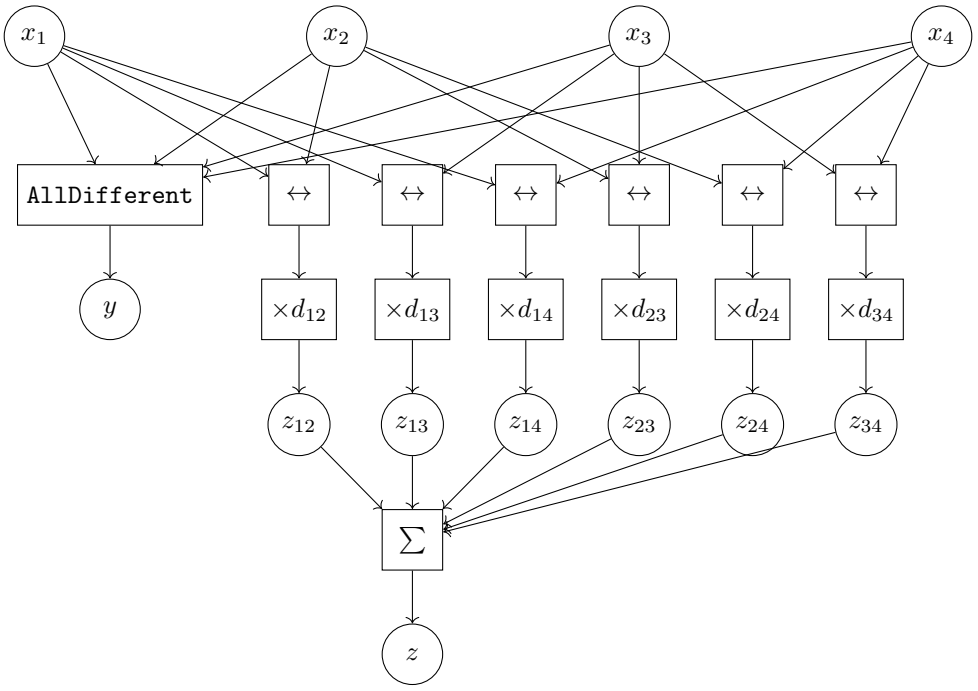

Figure 6: Invariant DAG for a TSP instance with 4 cities and distances between city i and j define by $d_{ij} = d_{ji}$

For a TSP with n cities, we model a decision variable x_i for each city i where $x_i = j \in \{1, \ldots, n\}$ indicates that city i is in position j of the tour. The chosen tour representation is arbitrary; shifting all positions by one and moving the variable with value n to position 1 yields the same objective value.

All variables are connected to an `AllDifferent` invariant to ensure each city occupies a distinct position. For each pair of variables $\{i, j\}$, we define an output variable z_{ij} such that:

$$z_{ij} = \begin{cases} d_{ij} & \text{if } x_i \text{ and } x_j \text{ are consecutive} \\ 0 & \text{otherwise} \end{cases}$$

This variable is defined through a combination of two invariants: first, a consecutiveness invariant (denoted by $\leftrightarrow$). Variables x_i and x_j are considered consecutive if $|x_i - x_j| = 1$ or $n - 1$. The resulting boolean value is then multiplied by the distance d_{ij}. The objective function z is the sum of all z_{ij} variables.

C Invariant DAG for Graph Coloring Problem

A graph coloring instance with n nodes is modeled using n decision variables x. The input includes a maximum number of colors k for the graph. Thus, $x_i = j \in \{1, \ldots, k\}$ indicates that color j is assigned to node i.

The objective function z is defined as the maximum value among all x_i variables, providing an upper bound that, in a minimization context, becomes equal to the minimum number of colors used. For each edge $\{i, j\}$ in the graph, we add a difference constraint represented by a violation variable y_{ij}, where:

$$y_{ij} = \begin{cases} 1 & \text{if } x_i = x_j \\ 0 & \text{otherwise} \end{cases}$$

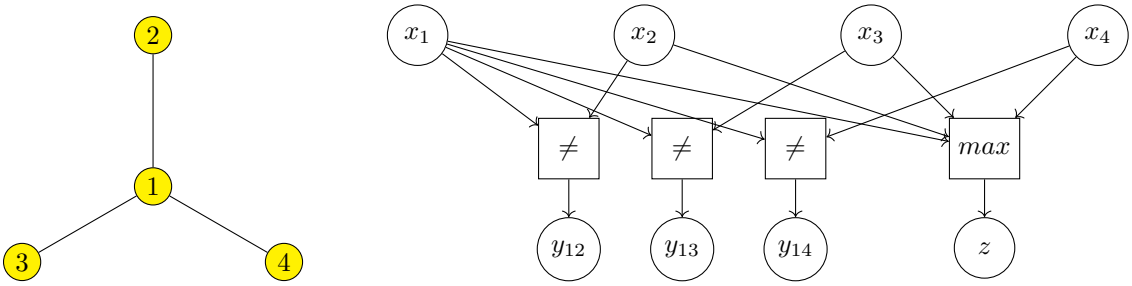

Figure 7: Invariant DAG for a specific Graph Coloring Problem instance (yellow)