Sequence Variables and Search Heuristics for Vehicle Routing Problems in Constraint Programming

Augustin Delecluse

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

June 2025

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Pierre **Schaus** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Peter **Van Roy**	UCLouvain/ICTEAM, Belgium
Pr. Yves **Deville**	UCLouvain/ICTEAM, Belgium
Pr. Pascal **Van Hentenryck**	Georgia Tech, USA
Dr. Paul **Shaw**	IBM, France

Sequence Variables and Search Heuristics for Vehicle Routing Problems in Constraint Programming

by Augustin Delecluse

© Augustin Delecluse 2025
ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.A

Artificial intelligence writing tools were used to reformulate portions of the text and enhance the coherence of the document. The tools used were Chat-GPT (version o4 and 3o, OpenAI) and Antidote (version 11).

Abstract

Constraint Programming (CP) is an optimization paradigm well suited for solving Vehicle Routing Problems (VRPs), thanks to its declarative framework. Yet, despite this relative ease of modeling, CP solvers tend to make poor branching decisions while searching for optimal solutions on a VRP. This, combined with the difficulty in dealing with optional visits in some VRPs variants, hinders the performance of CP solvers on such problems.

This thesis proposes two ways to strengthen the CP solving of VRPs. First, several heuristics are introduced that automate more informed decisions (*e.g.* nearest neighbor selection) within the CP search, which discover better VRPs solutions more quickly. Second, insertion sequence variables introduced in previous work are modified, formalized, and enhanced. Sequence variables allow easy handling of optional visits and support efficient heuristic strategies based on insertions into existing paths, both of which are valuable for efficient solving of VRPs. The proposed modifications further improve their performance, simplify the modeling and implementation of custom heuristics, and clarify their domain and consistency properties. On benchmarks such as the Dial-A-Ride Problem we approach the best-known objective values, and on the Traveling Salesman Problem with Time Windows we match them; demonstrating that CP performance can be greatly improved by incorporating our contributions.

Acknowledgments

Completing a PhD is no easy task, and it would have been impossible without the encouragement and help of many people, often in ways they may not even realize. Whether you were unraveling technical questions with me or helping me unwind through shared moments of fun, each of you lightened the journey.

First, I owe my deepest gratitude to my parents for their unwavering support throughout my studies. My heartfelt thanks also go to my siblings Martin, Zoé, Guillaume and Sacha, for their encouragement and patience, even when I launched into complicated computer-science explanations at family gatherings.

I am equally grateful to my friends and my colleagues from the INGI department. Especially my teammates from the Artificial Intelligence and Algorithm group: Xavier, Vianney, Alexandre, Guillaume, Benoît, Lucile, Damien and Emma, who helped me to solve stubborn technical problems and refine various graphic designs. The AsCII team deserves special mention for constantly fostering a cheerful atmosphere, and Vanessa for not only being the soul of the department but also regularly trouncing me at belote.

My appreciation extends to Alexander, Alice and Achille, who endured my quirks both at work and in our shared house. Luckily for all of us, my friends Myrtille, Réglisse, Célia, Laura, Louis D., Louis P., Margot, Diego, Océane and Elodie were also there to keep spirits high.

Finally, I want to thank Peter, Yves, Pascal and Paul for reading this thesis and assessing its merit. Above all, I thank Pierre, my thesis supervisor, whose guidance has been indispensable to my growth as a computer scientist and researcher.

Contents

Introduction 1

The Vehicle Routing Problem (VRP) is a class of optimization problems with many variants, where the goal consists in finding optimal paths for a fleet of vehicles under various constraints [BRV16]. Finding the best solutions to those problems is a difficult task, and many different approaches exist that try to balance speed and solution quality. Given the large number of problem variants, finding approaches that can be easily adapted to a new variation, while remaining efficient, is challenging.

Constraint Programming (CP) is a strong candidate to deal with VRPs [KS06], as its declarative paradigm allows to easily add new variations to a problem formulation. However, despite CP's simple modeling, it is not always the best approach when tackling VRPs. Part of the inefficiencies lies in the way CP attempts to create a solution on VRPs, as the exploration of solution candidates does not always exploit information in the problem, and may take a large amount of time to discover good solutions. Furthermore, the modeling of optional visits present in some VRP variants is not always straightforward, and may hinder the performance of CP solvers.

Recently, work conducted by Thomas, Kameugne, and Schaus introduced a new way to tackle VRPs in CP, through the use of insertion sequence variables [TKS20]. This approach proved to be more effective than other CP alternatives on several VRPs [Tho23]. Yet, their formal definition of sequence variables was relatively complex, and we believed that an even better work could be achieved.

1.1 Research Goals

This thesis attempts to keep the ease of modeling from CP, while enhancing its performance when solving VRPs. In particular, it tries to get closer to optimal solutions found on complex VRPs, where CP may be dominated by other approaches. It also aims to deepen our knowledge about the newly proposed insertion sequence variables, formalizing their definition, studying possible extensions, and applying them on additional VRPs.

1.2 Contributions

The main contributions of this thesis are

- Several black-box[1] value selection heuristics for CP, that try to balance speed and information. They aim to automate greedy decisions on problems such as the Traveling Salesman Problem.
- Further work on insertion sequence variables:
  - A formal definition of the domain of insertion sequence variables.
  - An implementation of an insertion sequence variable domain, presenting data-structures, invariant and properties of the domain.
  - A simplification of the API and algorithms defined over insertion sequence variables compared to previous versions.
  - An introduction of dedicated constraint algorithms and consistency levels for insertion sequence variables.
  - An application of insertion sequence variables on several optimization problems, such as the Dial-A-Ride Problem and the Traveling Salesman Problem with Time Windows.

1.3 Publications

A significant part of this work has been published in:

- A. Delecluse and P. Schaus. "Black-Box Value Heuristics for Solving Optimization Problems with Constraint Programming (Short Paper)". In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024).* Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024

  This paper presents several value selection heuristics for CP, that can be used in a black-box context. They automate greedy search strategies when applied on the Traveling Salesman Problem.

- A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables for Routing Problems". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022

[1] A value-selection heuristic is black-box (domain-independent) if it can be plugged, unchanged, into any CP model because it relies solely on generic information provided by the solver at run time. White-box heuristics, by contrast, are tailored to a given problem or model and encode explicit semantic knowledge, so they are not reusable across unrelated problems.

> This paper presents a first variation of the sequence variables introduced previously [TKS20]. The previous implementation was simplified, and we obtained better performance on several VRPs, including new best known solutions for the classical benchmark of the Traveling Salesman Problem with Time Windows [Lóp20].

Another research paper is still in preparation at the time of writing this thesis. It presents a second variation of sequence variables in A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables: A Constraint Programming Computational Domain for Routing and Sequencing". Manuscript in preparation. 2025.

Finally, another contribution of this thesis is the implementation of insertion sequence variables in the CP solver MaxiCP [Sch+24].

1.4 Outline

The necessary background about CP and VRPs is first introduced in Chapter 2. It finishes by identifying some of the challenges faced when solving VRPs with CP. One of them is the derivation of efficient black-box search strategies, which is directly addressed in Chapter 3. There, value heuristics developed in the thesis are presented, which aim to automate greedy strategies for VRPs. Chapter 4 presents the core content of the thesis: insertion sequence variables. It gives a formal definition of sequence variables, proposes an implementation, and presents related constraints and search algorithms. The chapter ends with their application on various VRP variants and a discussion regarding limitations and future work. Finally, the last chapter summarizes the work developed in this thesis and highlights some future research directions.

Background 2

This chapter provides the necessary background to fully grasp the content of the thesis. Vehicle Routing Problems (VRPs), which is the main class of problems targeted by this work, are first presented in section 2.1. Section 2.2 then presents Constraint Programming (CP), the paradigm explored in this thesis for tackling the VRP. Section 2.3 presents Large Neighborhood Search, a meta heuristic commonly used in VRPs and compatible with CP. Finally, section 2.4 presents some challenges when attempting to solve VRPs with CP, possibly with LNS, and ends with the research questions studied in this thesis.

2.1 Vehicle Routing Problems

Vehicle Routing Problem (VRP) is a class of problems dealing with transportation. In many cases, such problems involve finding itineraries for a fleet of vehicles that must visit a set of locations, possibly by minimizing a given objective, such as the travel length. A VRP instance and a feasible solution are shown in Figure 2.1.

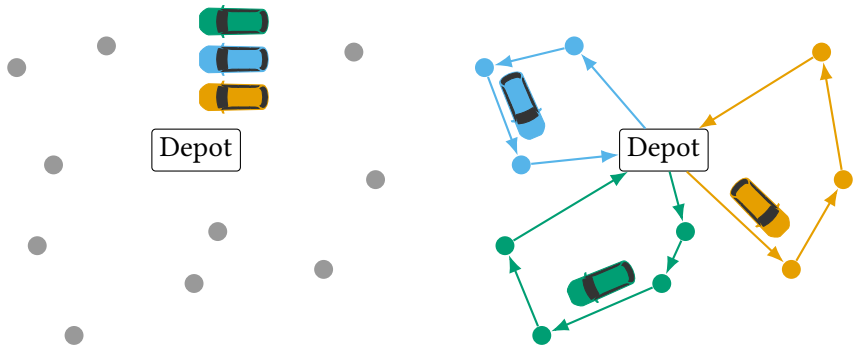


Figure 2.1: Example of a VRP with 3 vehicles that must visit a set of locations, each starting from and returning to a common depot (left). A possible solution is shown on the right.

We will first present the simplest (and most well known) VRP: the Traveling Salesman Problem.

2.1.1 Traveling Salesman Problem - TSP

The Traveling Salesman Problem (TSP) is an optimization problem defined over n cities. A matrix $\mathbb{R}^{n \times n}$ defines the distance between cities, and the goal of the problem consists in finding a tour for a salesman that minimizes the traveled distance, while visiting each city exactly once and returning to the starting city. A TSP instance and its optimal solution are shown in Figure 2.2.

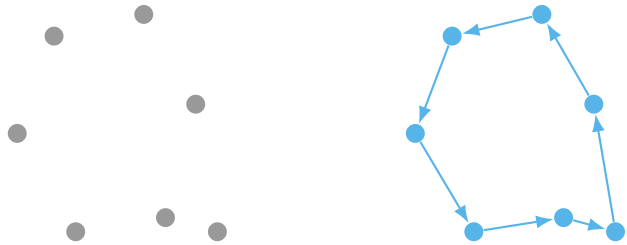


Figure 2.2: A TSP instance (left) and its corresponding optimal solution (right). The distances between cities are Euclidean.

This problem is NP-hard, so no exact polynomial-time algorithm for solving the TSP is known, and such an algorithm is considered unlikely to exist [Kar09]. There are two main lines of research devoted to solving the TSP, which can also be used to describe the works on VRPs:

1. **Exact methods**, which aim at finding the best solution and proving its optimality.

2. **Heuristic methods**, which target rapidly finding a good quality solution, with no guarantee on its optimality.

For instance, one exact approach for solving the TSP is the Concorde solver, which managed to solve to optimality an instance of 85,900 cities [App+09]. Regarding heuristics approaches, the Lin-Kernighan heuristic (LKH) initially proposed by Lin and Kernighan in [LK73] and later refined by Helsgaun in [Hel00], provides within a few seconds optimal or near-optimal solutions to instances with thousands of cities.

This thesis does not aim to fully cover the literature or approaches on the TSP, but rather to use the TSP in some examples, as a TSP instance and candidate solution can be easily visualized. The reader is referred to [Coo+11] for a more complete overview of the techniques used in the Concorde solver, and to [GP06; Law85] for a broader cover of existing techniques on the TSP.

Although the TSP already has direct applications in real-life (for instance, finding the best path of a postal truck could be defined as a TSP in some cases), this problem can be extended by adding several constraints in order to cover more complex situations.

2.1.2 Extending the TSP

One big difference between the TSP and many other VRP variants is the presence of several vehicles that may be used. A VRP variant might also involve additional constraints further restricting the types of tours that are valid in a solution. Some of the most commonly encountered constraints and aspects are

Depots that define the starting and ending location of each vehicle. Two main situations exist: *homogeneous* depots, where all vehicles use the same depot locations, and *heterogeneous* depots, where the depots are different between vehicles.

Optional visits are encountered when the visits of some location may be omitted. In some of those problems, a penalty cost might occur when not visiting a location by any vehicle.

Time windows are used to define valid times for the visit of nodes. When working with such constraints, in many cases, a time matrix is also employed. Compared to a distance matrix, that defines transition cost between nodes, this matrix defines the transition time between locations. Moreover, visiting a node may induce a given processing time, and waiting at a node (*i.e.* arriving too early at a location and waiting until the beginning of its time windows to perform the task related to the visit) may be allowed.

Time dependent travel times where the travel time between nodes changes over time, which is often the case in real-life problems due to traffic congestion.

Capacity allow representing the transport of goods in a vehicle. Visiting a node in the problem leads to a load change in a vehicle, whose available capacity is limited.

Precedences enforce an ordering between visits that must be respected in a vehicle path.

Pickup and deliveries define pairs of nodes (a pickup and its corresponding delivery) that are linked together. The nodes within a pair must be visited by the same vehicle, and the visit of the pickup must occur before the visit of its corresponding delivery. Between the visit of the pickup and the delivery, a load change occurs in the vehicle.

Given the wide variety of vehicle routing problems that may be defined by this simple non-exhaustive list of constraints, an extensive cartography of

all existing VRPs falls outside the scope of this work. The reader is referred to [BRV16] for a wider overview of the problem variants.

The majority of VRP problem classes are NP-hard. Consequently, significant effort and time have been dedicated to finding effective approaches for a given VRP variant. Compared to other works that attempt at finding the best approach for a single class of VRP, this thesis aims at developing techniques that may be simply adapted between different classes of VRP, while still being *reasonably* efficient. The next section presents Constraint Programming, a paradigm that can be used to swiftly adapt between VRP variants.

2.2 Constraint Programming

Constraint Programming (CP) is a declarative paradigm used to solve combinatorial problems, and is a widely used approach for solving them [KS06]. As expressed by Freuder in [Fre96], CP *"represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."*

CP is sometimes described as "Model + Search" as it solves combinatorial problems by combining a declarative model with a backtracking search, that attempt to find solutions encoded in the model. Those two components are described next. The notations from this section are largely borrowed from [MSV21].

2.2.1 Model

A CP model is used to define the combinatorial problem being tackled. It is used to solve constraint satisfaction problems (CSPs), which are commonly described by a triplet $\langle \mathcal{X}, \mathcal{D}, C \rangle$, where $\mathcal{X} = \{x_0, \ldots, x_{n-1}\}$ is the finite set of variables composing the problem, $\mathcal{D} = \{\mathcal{D}_0, \ldots, \mathcal{D}_{n-1}\}$ are the domains used by those variables, and C is the finite set of constraints restricting the values present in the domains of the variables. The next sections will present each of those concepts.

2.2.1.1 Variable

A variable in constraint programming is a decision entity that must be assigned a value from a domain; these assignments are governed by constraints defining the problem. Various types of variables exist, the most common one being integer variables. In the context of the TSP, such a variable can be used to represent (the index of) which city is visited after another one, or the (integer) distance between a location and the next one in a TSP tour.

A non-exhaustive list of other types of variables encountered in CP is presented next. They all share the same fundamental concept: they are decision entities whose values must satisfy the problem's constraints.

Boolean Variables that represent an unknown boolean value. Many CP solvers consider boolean variables as integer variables that only represent values 0 and 1, corresponding to *false* and *true*, respectively.

Set Variables representing a set of elements, such as integers. The size of the set and the elements composing it are to be decided. They were introduced in [Pug93; Ger93; Ger95; Ger06].

Graph Variables representing unknown graphs, where the nodes and edges composing the graphs are to be decided. They were first introduced in [DDD05].

Optional Task Interval Variables are mainly used in scheduling problems for representing when a task begins and ends, and if it must be executed or not. They were introduced in CP Optimizer and the following work [LR08; Lab+09b; Lab+18b], and are also present in the Or-Tools solver [PFa].

Interval Sequence Variables represent an ordering over a (sub)set of optional task interval variables. The order of appearance of the optional tasks, and which ones belong to the sequence, is to be decided. Such variables were introduced in [Lab+18b; Lab+18a]. They are also present in the OrTools solver. [PFa; PFb] These can be used to model VRPs, where tasks are related to the visits of nodes, and the ordering represents the visits performed by a vehicle.

Insertion Sequence Variables are similar to interval sequence variables, but represent an ordering over a (sub)set of integers instead of interval variables. Compared to interval sequence variables, which are mainly targeted for scheduling applications, those variables are mainly targeted for routing, for reasons highlighted in section 4. They were first introduced by Thomas, Kameugne, and Schaus in [TKS20], and more detailed in the thesis of Thomas [Tho23].

String Variables represent a string, whose length and characters composing it are to be decided. They were developed in [SFP15; Sco+17].

BitVector variables were introduced in [MV12b], in order to represent problems in verification and cryptography. They represent a vector of bits, where the activated bits are to be decided.

2.2.1.2 Domain

A domain $\mathcal{D}$ is used to represent a set of values. A variable $x \in \mathcal{X}$ is associated with one domain $\mathcal{D}$, that represents the set of values the variable can take. The operation $\mathcal{D}(x)$ over a variable x allows retrieving its domain. In the context of a TSP with n cities, the domain $\mathcal{D}$ of an integer variable x_0, defining the city visited after location 0 in a TSP tour, may correspond to $\{1, 2, \ldots, n-1\}$. Whenever the domain of a variable is a singleton (*i.e.* it contains only one value), the variable is said to be *fixed*, and *unfixed* otherwise.

Each domain type is specific to the variable it is associated with (*e.g.* integer domains are specific for integer variables). Different data structures may be used to represent a given domain, with different trade-offs in both computational time and memory.

Integer domains are ordered and inherit their order from the integers. The lower bound $\lfloor \mathcal{D} \rfloor$ of an integer domain $\mathcal{D}$ describes the smallest value contained within it. Similarly, the upper bound $\lceil \mathcal{D} \rceil$ relates to its largest value. An integer domain is said to be *dense* if all integers between the lower and upper bound belong to its domain, and *sparse* otherwise.

Example 2.2.1. Assuming a domain $\mathcal{D} = \{0, 1, 3, 4\}$, its lower bound is $\lfloor \mathcal{D} \rfloor = 0$, its upper bound is $\lceil \mathcal{D} \rceil = 4$ and the domain is sparse, as $2 \notin \{\lfloor \mathcal{D} \rfloor, \ldots, \lceil \mathcal{D} \rceil\}$. Given that the domain is not a singleton ($|\mathcal{D}| > 1$), a variable x_i with domain $\mathcal{D}$ is unfixed.

A candidate solution σ assigns to each decision variable x a value in its domain: $\sigma(x) \in \mathcal{D}(x) \forall x \in \mathcal{X}$.

Handling backtracks A CP model is often combined with a backtracking depth-first search. The domains encoded within a model need thus to be saved and restored on backtrack. There are two main mechanisms for saving and restoring a state: copying and trailing. The first method, copying, creates a new fresh copy of all values in the state for each node considered during the search tree exploration. On backtrack, the state corresponding to a given node in the search tree is retrieved. The second method, trailing, can be seen as an optimized lazy version of copying. It stores undo operations, by encoding information on how to reconstruct the state after a change has been performed. Those undo operations are only stored for values in the state that have been changed since the last save, compared to copying which must fully retain the full state. Trailing is therefore well suited when only a small portion of the state is changed.

A popular and efficient CP solver relying on copy is Gecode [STL10], while trailing is found in modern CP solvers such as MiniCP [MSV21], Ace [Lec23], Choco-Solver [PF22] but also in older solvers such as CHIP [Meh+88]

and ILOG Solver [Sol95]. The reader is referred to [MSV21] for a more detailed description of state management, as well as copy and trailing mechanisms. Trade-offs between those two methods are also discussed in [Sch99].

The methods developed in this thesis have been implemented in trail-based solvers (MiniCP [MSV21], MaxiCP [Sch+24] and Choco-Solver [PF22]).

Example 2.2.2. This example is borrowed from [MSV21; SMV25]. Listing 1 presents the usage of reversible integers (`StateInt`) in the MiniCP solver. Saving and restoring the state is done by a `StateManager`, which is either a Copier or a Trailer (working by copying or trailing, respectively). The entries stored in memory are illustrated in Figure 2.3. The Copier stores the values of all reversible integers at each `saveState` call, creating a fresh copy of the full state ready to be used. In contrast, the Trailer stores in a current backup the undo operations: the assignment to perform after executing a `restoreState` operation. In either case, each `saveState` call adds entries in a stack of backup, which consumes memory.

```
StateManager sm = makeStateManager();
// sm is either a Trailer or Copier
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);

sm.saveState(); // record current state a=7, b=13
    a.setValue(6);
    a.setValue(11);
    sm.saveState(); // record current state a=11, b=13
        a.setValue(4);
        b.setValue(9);
    sm.restoreState(); // now a=11, b=13
sm.restoreState(); // now a=7, b=13
```

Listing 1: Usage of reversible integers (`StateInt`) in MiniCP.

Sparse Sets Several data structures can be used to represent an integer domain in a reversible way. One of them is the reversible sparse set from [BT93; Sai+13]. It relies on two integers arrays and only one reversible integer. It allows querying in constant time if a value is contained in a domain $\mathcal{D}$, and iterate in $O(|\mathcal{D}|)$ over the domain. Removing a single value and fixing the domain to a singleton are constant time operations.

In a reversible sparse set, one array *values* encodes the current values in the domain $\mathcal{D}$. A second array *indices* tells the index of each domain value

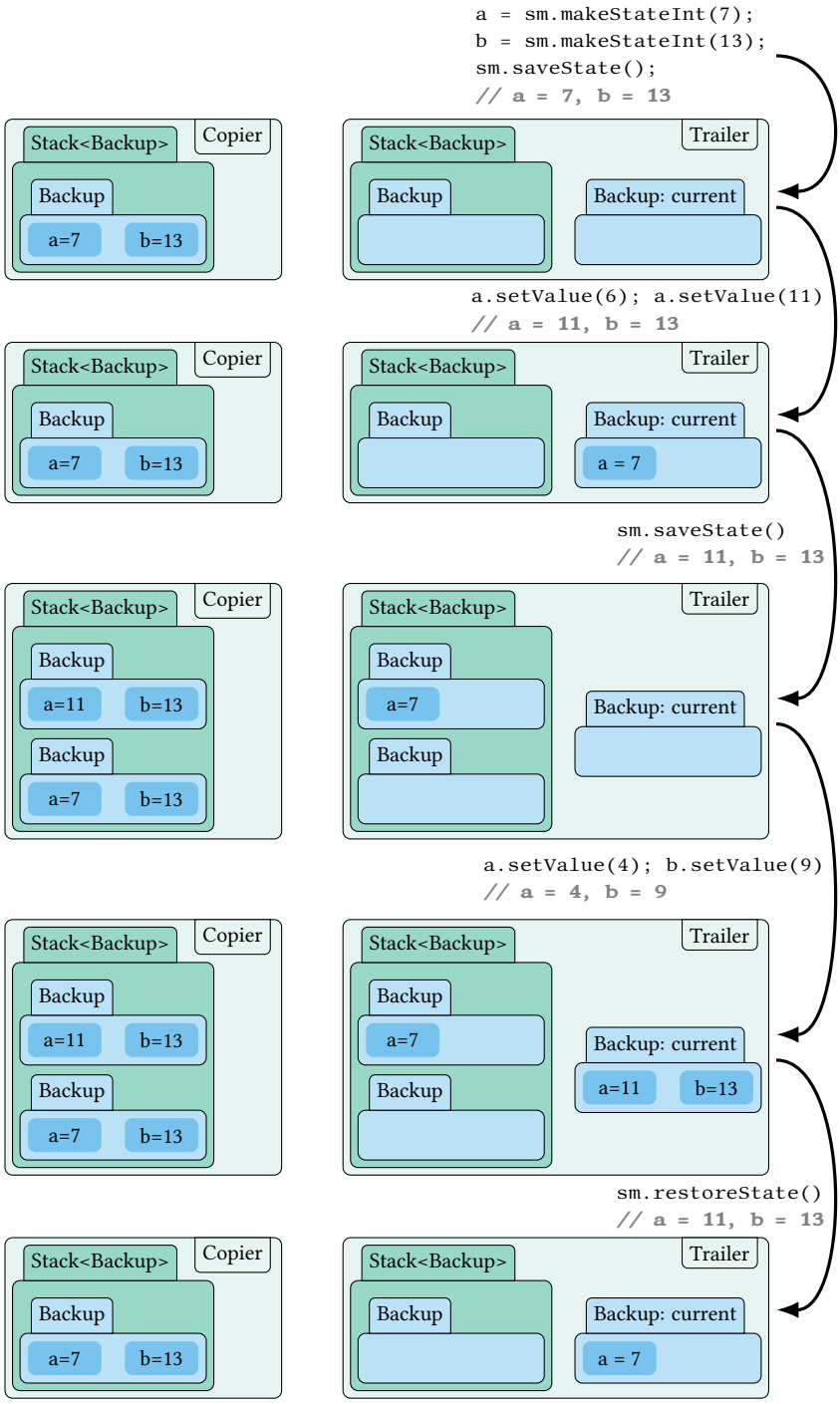


Figure 2.3: Entries stored between a Copier and a Trailer for listing 1.

within the values array. Finally, a reversible integer n tells the size of the current domain. For a domain $\mathcal{D}$, the following invariants are maintained:

$$values[indices[v]] = v \tag{2.1}$$

$$\mathcal{D} = \{values[i] \mid 0 \leq i < n\} \tag{2.2}$$

One main advantage of this data structure is that only one reversible integer (n) must be tracked, no matter the domain size. Domain changes may only occur through value removals: either by removing a single value from the domain or by fixing $\mathcal{D}$ to a singleton. The previous state of a domain is retrieved by restoring the value of the reversible integer n. It is worth pointing out that this data structure does not maintain the order of values, which may impact the interactions with algorithms iterating over the remaining values in the domain.

Example 2.2.3. A reversible sparse set and operations on it are represented in Figure 2.4.

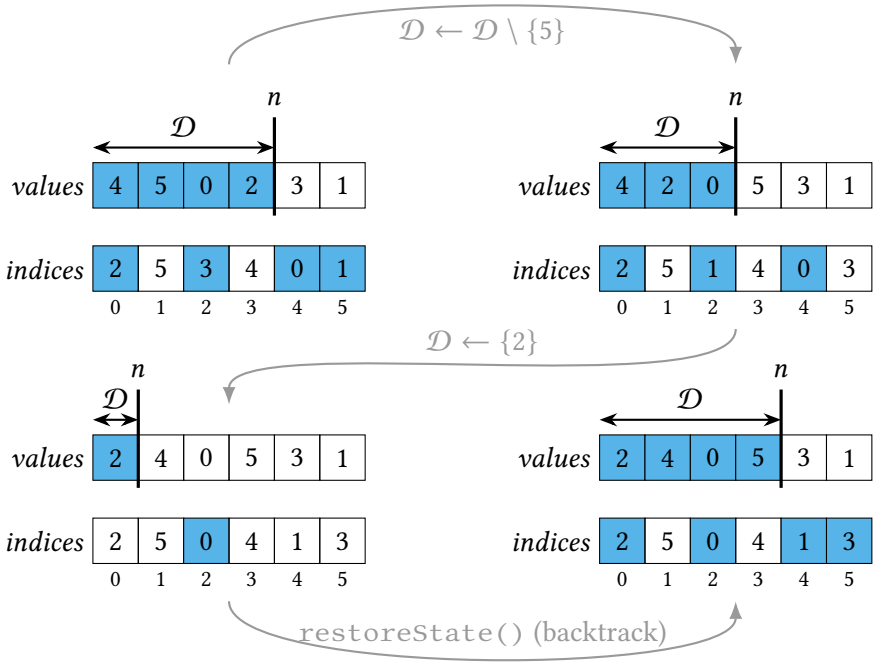


Figure 2.4: A domain $\mathcal{D}$ encoded in a reversible sparse set. From an initial domain (top left), value 5 is removed (top right). The domain is then fixed to value 2 (bottom left). It is then restored to its initial state (bottom right).

A sparse set actually encodes a bipartition, separating by the n marker (*i.e.* a reversible integer) the values currently in the domain $\mathcal{D}$ and the values that were previously in the domain. This data structure can also encode a tripartition over a set of values V, by using two markers r, p instead of a single one, as shown in [Sai+13]. The invariants maintained in this case are

$$(R \cup P \cup X = V) \wedge (R \cap P = R \cap X = P \cap X = \emptyset) \tag{2.3}$$

$$R = \{values[i] \mid 0 \leq i < r\} \tag{2.4}$$

$$P = \{values[i] \mid r \leq i < p\} \tag{2.5}$$

$$X = \{values[i] \mid p \leq i < |V|\} \tag{2.6}$$

And invariant (2.1) is kept. The set V is split into three disjoint subsets, R, P and X (2.3). The split between the set is encoded with the markers r and p (2.4), (2.5), (2.6). Initially, the sets R and X are empty, and $P = V$. The set P is shrunk over time, by moving values from P into either R or X. The set P can only decrease in size, while both R and X can only increase in size through the data structure updates. A previous state is retrieved by restoring the value of the two reversible integers r, p.

Example 2.2.4. Figure 2.5 shows a tripartition implemented with a sparse set.

Optional Task Interval Domain The domain $\mathcal{D}$ of an optional task interval variable represents when a task begins, its duration, and when it ends [LR08; Lab+09b]. Additionally, such a domain can take a special "absent" value if the task is not executed. Queries on the domain allow retrieving

- A *Start*, that indicates when the task begins.
- An *End*, indicating when the task finishes.
- A *Duration* representing the duration of the task.
- A *Presence* status, telling if the task is executed or not.

The start, end and duration can be represented by integer domains. In CP Optimizer [Lab+18b], a commercial scheduling solver, those 3 integer domains only support update of their lower and upper bound, meaning that they do not require data structures such as sparse sets for their implementation. The presence status can be represented by a boolean domain.

Example 2.2.5. Figure 2.6 shows an optional task interval domain where the presence is set to true, $Start \in \{0, 1, 2, 3\}$, $End \in \{4, 5, 6, 7, 8\}$, $Duration \in \{3, 4, 5\}$. One instantiation of the domain is shown below, where the start, end and duration are fixed.

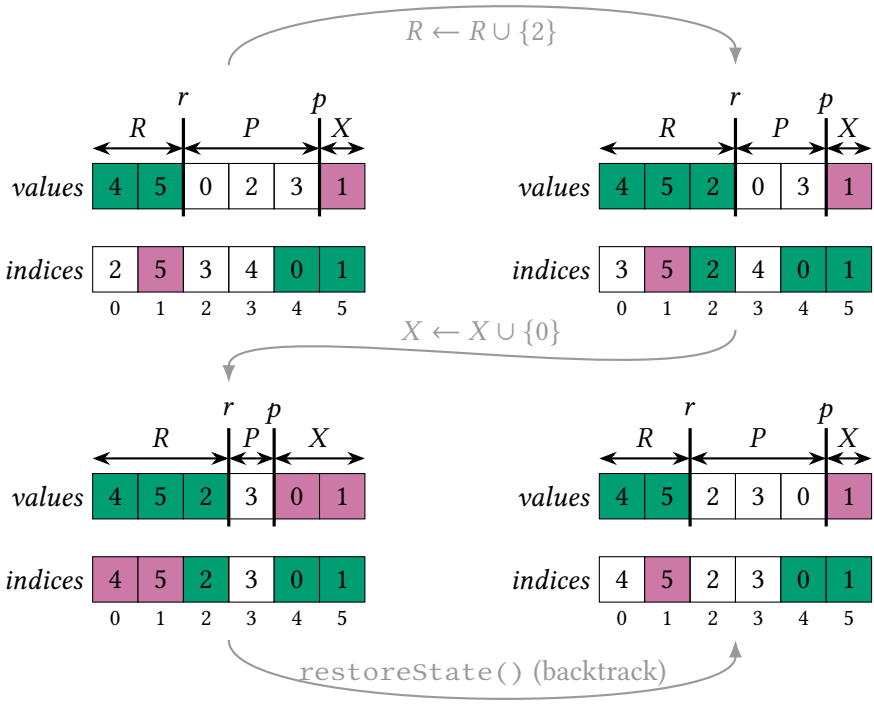


Figure 2.5: A reversible sparse set partitioning a set of values V into three disjoint sets R, P, X. From an initial partitioning (top left), value 2 is added into R (top right). Value 0 is then added into X (bottom left). The partition is afterward restored to its initial state (bottom right).

Interval Sequence Variable Domain An interval sequence variable domain represents all orderings over the optional task intervals related to it. Some or all optional tasks may be absent.

In terms of implementation, CP Optimizer represents such a domain with a *head-tail structure* [Lab+18b; Lab+18a], which is also the case in the OrTools solver according to their manual [PFa; PFb; 21]. The head (resp. tail) represents the beginning (resp. ending) of the sequence. Both the head and the tail can grow, by adding one optional task interval to it. A domain becomes fixed whenever the head and the tail merge together, therefore representing only one ordering. At the beginning, when no task is included, the head and the tail are represented by *sinks*, special values indicating that no task is currently the first or the last one in the sequence.

Example 2.2.6. Figure 2.7 shows the domain of an interval sequence variable over a set of optional task intervals $\{t_1, \ldots, t_{18}\}$. The sinks are represented by α and ω for the head and the tail, respectively.

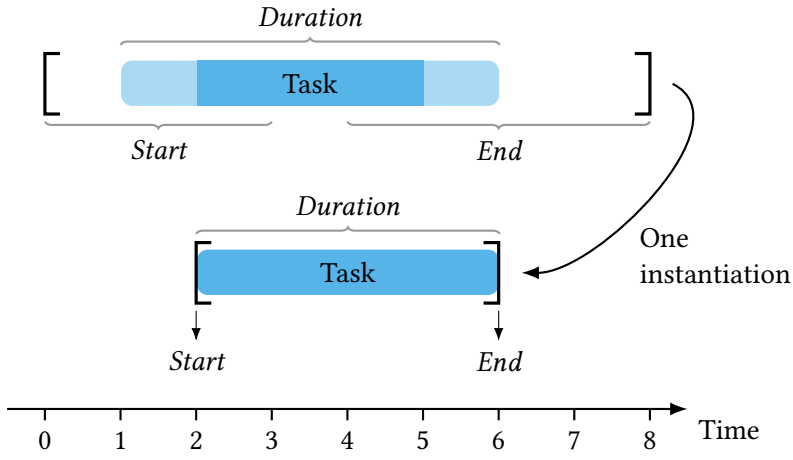


Figure 2.6: Domain of an optional task interval (top) and a possible instantiation for it (bottom). The rectangle represents the processing time of the optional task interval.

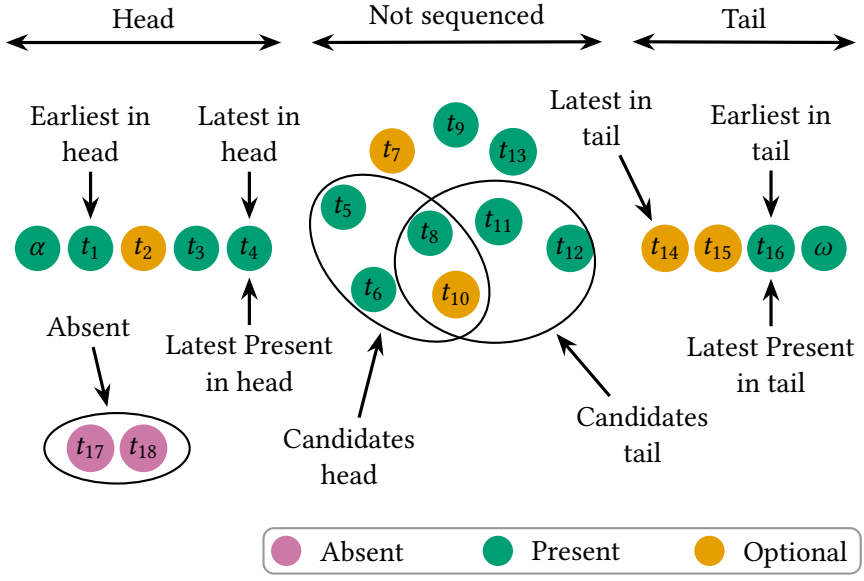


Figure 2.7: Domain of an interval sequence variable.

2.2.1.3 Constraint

A constraint $c \in C$ is used to restrict the domain(s) of one or several variables through a relation. The variables affected by a constraint c are said to be in the *scope* of c. In the context of a TSP, given that each city must be visited exactly once, one constraint may restrict the domain of the successor variable

x_i of a city i to be different from i: no self-loops are allowed.

The valuation function $c(\sigma)$ of a constraint c with respect to a candidate solution σ is the 0-1 value evaluating to 1 if and only if the constraint holds. For instance, assuming a constraint $c(x, y) \leftrightarrow x = y + 1$, with $\mathcal{D}(x) = \{0, 1\}$ and $\mathcal{D}(y) = \{1, 2\}$, $c(0, 1) = 1$ and $c(0, 2) = 0$. A solution to a CSP $\langle \mathcal{X}, \mathcal{D}, C \rangle$ is an assignment satisfying all constraints in C. The set of all solutions is written by $\mathcal{S}(\langle \mathcal{X}, \mathcal{D}, C \rangle)$. A CSP without any solution is said to be unsatisfiable (UNSAT) and satisfiable (SAT) otherwise.

Given a constraint c, we can introduce a boolean variable b indicating whether c is satisfied. *Reification* is then defined by the equivalence

$$\text{reify}(c, b) \iff \big(c \leftrightarrow b = 1\big).$$

That is, $b = 1$ if and only if c holds.

A filtering algorithm $\mathcal{F}_c$, also called *propagator*, is a function associated with a constraint $c \in C$, which takes as input the domains of the variables, and outputs shrunk domains, without the removal of any solution to the constraint c. Different filtering algorithm may exist for the same constraint.

A constraint enforces a given *consistency* level, which describes the strength of its filtering. Two of the most commonly encountered consistency levels are

Bound consistency that assumes that all domains of variables in the scope of the constraint c are dense, and enforces that the bounds of each domain of every variable participate in a solution. Assume that $scope(c) = \{x_0, \dots x_{n-1}\}$ is a function retrieving the n variables in the scope of c. Let us define $\mathcal{N} = \{0, \dots, n-1\}$, the set of indices of the n variables in the scope of c. We can define by the predicate

$$\begin{aligned} g(i, v) = \forall j \in (\mathcal{N} \setminus \{i\}) : \exists w_j \in \mathcal{D}(x_j) \\ \text{s.t. } c(w_0 \dots w_{i-1}, v, w_{i+1} \dots w_{n-1}) = 1 \end{aligned} \tag{2.7}$$

that there exists a solution to the constraint such that $x_i = v$. Bound consistency is defined as:

$$\forall i \in \mathcal{N} : (g(i, \lfloor \mathcal{D}(x_i) \rfloor) \wedge g(i, \lceil \mathcal{D}(x_i) \rceil)) \tag{2.8}$$

Domain consistency that enforces that each value in each domain of the variables in the scope of c participates in a solution.

$$\forall i \in \mathcal{N} \, \forall v \in \mathcal{D}(x_i) \, g(i, v) \tag{2.9}$$

Although domain consistency filters more inconsistent values than bound-consistency, enforcing it might be more time-consuming and complex than enforcing bound consistency.

The *fixpoint* of a CSP is obtained after executing the filtering of all constraints included in the CSP, so that no further filtering of any domain occurs.

Even when all constraints composing the CSP enforce domain consistency and a fixpoint has been reached, some values within the domain of variables may not belong to any solution. A value that belongs to a solution of a CSP is said to be *globally consistent*, while a value belonging to a domain after the fixpoint has been reached are said to be *locally consistent*.

Example 2.2.7. Let us consider a CSP with $\mathcal{X} = \{x_0, x_1, x_2\}$, $\mathcal{D} = \{\{0, 1, 2\}, \{0, 1\}, \{0, 1\}\}$ and $C = \{c_0, c_1, c_2\}$, with $c_0 \leftrightarrow x_0 \neq x_1, c_1 \leftrightarrow x_1 \neq x_2, c_2 \leftrightarrow x_2 \neq x_0$. The values within all domains are locally consistent: the fixpoint has been reached and any filtering from c_0, c_1 and c_2 may not provoke further changes. However, in $\mathcal{D}(x_0) = \{0, 1, 2\}$, only value 2 is globally consistent: there does not exist any solution to this CSP where $\mathcal{D}(x_0) = \{0\} \vee \mathcal{D}(x_0) = \{1\}$.

It is worth noting that c_0, c_1, c_2 actually imply that all variables in $\mathcal{X}$ must be different. If those constraints were instead replaced by such an AllDifferent constraint, with a domain consistent propagator such as [Rég94], the domain of x_0 would have been directly reduced to $\mathcal{D}(x_0) = \{2\}$.

Works such as [BV03b] describe more properties associated with constraints. Hundreds of constraints exist (the Global Constraint Catalog website contains a list of 423 constraints at the time of writing this thesis [Dem14]). Some of the constraints encountered in this thesis are

- **Sum** which enforces that the sum of integer variables matches a given variable:

$$\text{Sum}(\boldsymbol{x}, y) \leftrightarrow \sum_{x_i \in \boldsymbol{x}} x_i = y \tag{2.10}$$

- **AllDifferent** [Rég94; Van01; ZLZ18] which enforces that n integer variables take different values:

$$\text{AllDifferent}(\boldsymbol{x}) \leftrightarrow \forall_{i \in \mathcal{N}, \forall j \in \mathcal{N} \setminus \{i\}} x_i \neq x_j \tag{2.11}$$

  Where $\mathcal{N} = \{0, \ldots, n-1\}$.

- **Circuit** is applied on an array of integer variables *succ*, and enforce that they represent a Hamiltonian circuit [Pes+98]. More specifically, each variable $succ_i$ must be instantiated to a value corresponding to

an index in the *succ* array. By following the links represented by the variables, all variables are traversed in a circuit. A CP model using such circuit constraint to model a VRP, for instance a TSP, is referred to as a *successor model*.

A variant is the subcircuit constraint available in some solvers [Net+07; Bou+16a; Van02; Lab+18b], that allows self-loops to be encoded. In this case, a variable $succ_i$ can be instantiated to value i, which represents a self-loop for this variable.

Example 2.2.8. One example of the circuit and the subcircuit constraint is shown on Figure 2.8.

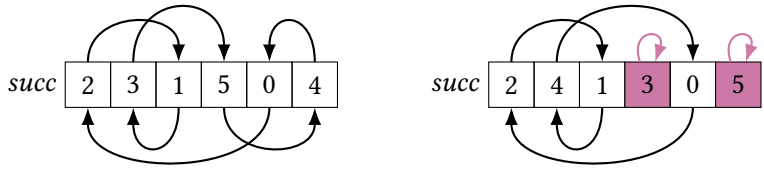


Figure 2.8: Circuit constraint (left) and subcircuit (right). On the right, variables $succ_3$ and $succ_5$ do not belong to the circuit.

Some variants also include a weighted version of this constraint. In addition to the *succ* array, the length of the circuit is captured in an integer variable. Transitions between variables can be weighted, which impacts the computed length. Given that this constraint enforces a Hamiltonian circuit, it is often used to model TSP and VRPs.

Note that this constraint implies an AllDifferent constraint over the *succ* variables.

- **Element** models array accesses [Van87; Van89; HC88]. It is defined by

$$\text{Element}(T, x, y) \leftrightarrow T[x] = y \tag{2.12}$$

The array T may be an array of integers or an array of variables.

- **Cumulative** is used with activities in scheduling contexts [GHS15; BC02; AB93]. Each activity is associated to a resource consumption amount. The cumulative constraint ensures that the cumulated load of executed activities never exceeds a given capacity at any point in time. More formally:

$$\text{Cumulative}(s, e, l, c) \leftrightarrow \forall t \sum_{i \in \{0,\dots,|s|-1\}, s_i \leq t < e_i} l_i \leq c \tag{2.13}$$

Where s is the start of the activities, e, l their corresponding end and load, and c is the capacity. Solvers typically support this constraint with integer or optional task interval variables for representing the activities. In the latter case, more expressive variants of the constraint allow defining functions related to the load of executed activities.

Example 2.2.9. Figure 2.9 shows one cumulative constraint with a capacity $c = 3$ and 4 activities, with loads 1, 2, 3 and 2.

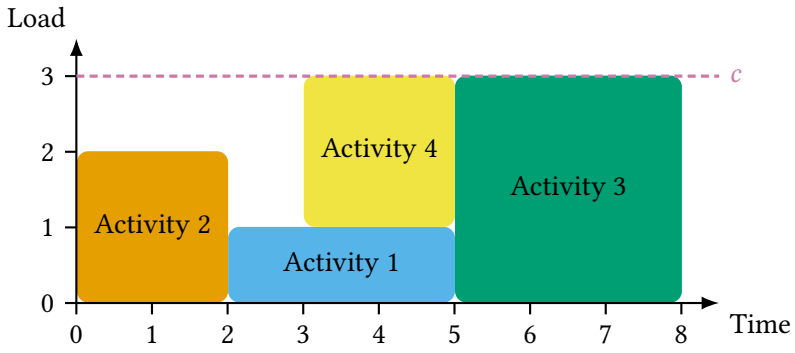


Figure 2.9: Cumulative constraint

In CP Optimizer, algebra to define cumulative constraints is provided in a richer API, allowing defining for instance a minimum load c to respect $(\forall t \sum_{i \in 0..|s|, s_i \leq t < e_i} l_i \geq c)$, and to define how the execution of an activity impacts the resource consumption (*i.e.* the resource is consumed during the full duration of the activity, is consumed from its start and stays consumed forever, etc.) [Lab09].

An important aspect of the filtering algorithms used with this constraint consists in *mandatory parts* [Lah82]. The mandatory part of an activity denotes a fragment of time when it will be executed, no matter the remaining domain operation that may occur on the variable. This information is then exploited by filtering algorithms such as [BC02; LBC12; OQ13; GHS15] to detect infeasible starting and ending times, and prune inconsistent values.

Example 2.2.10. Figure 2.10 shows the mandatory part of an optional task interval variable with its presence set to true, $Start \in \{0, 1, 2\}$, $End \in \{5, 6, 7\}$, $Duration \in \{5\}$. Its mandatory part starts at 2 and ends at 5.

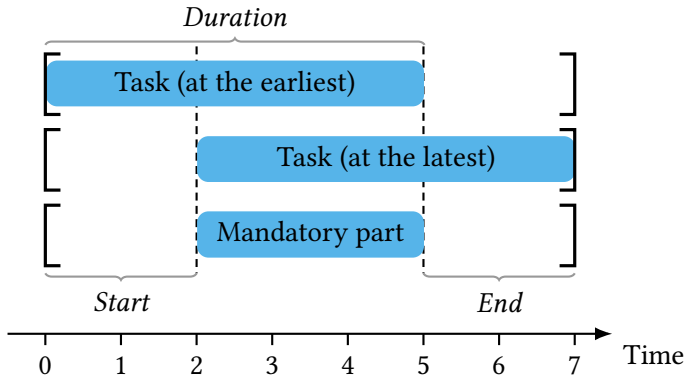


Figure 2.10: Mandatory part of an activity

2.2.1.4 Optimization Problems

A CSP may also include an objective function $f(\mathcal{X}^n)$, with $\mathcal{X}^n \subseteq \mathcal{X}$, to be optimized. In such cases, the CSP becomes a constraint optimization problem (COP), described by the tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, f \rangle$. Without loss of generality and unless explicitly stated, this thesis will assume that objectives are functions to be minimized. In the context of CP, they take variables as input and output an integer value.

A solution to a COP is a solution to the CSP it contains. The optimal solution of a COP is the solution with the best cost. Multiple optimal solutions may exist.

2.2.2 Search

In Example 2.2.7, we saw that even when enforcing domain consistency on every constraint, we do not necessarily directly find a solution to a CSP. In such cases, one needs to explore the *search space* defined by the CSP in order to find a solution, or to prove that none exists.

2.2.2.1 Branching Scheme

The branching scheme defines the strategy used to explore the search space. In CP, the search space is commonly explored, using a depth-first search (DFS). More specifically, each node considered by the DFS represents a given CSP. If the CSP contains variables that are unfixed, the corresponding node in the search tree can be expanded, partitioning the problem into several CSPs to consider. Each generated CSP adds a constraint further restricting the original CSP. Whenever a node is explored, all filtering algorithms are triggered until the fixpoint is reached. If all variables are fixed, it corresponds to a valid

assignment to the original problem. Backtracks occur whenever a domain becomes empty which corresponds to a failure: no solution exists to the considered CSP. Backtracks can also occur to jump back to a remaining CSP not yet considered.

Example 2.2.11. Continuing example 2.2.7. A possible search tree for solving the CSP is shown in Figure 2.11. The original CSP at node 0 is split into two CSPs: the first one considered (node 1) adds a constraint $x_0 = 0$ to the problem, while the second one adds a constraint $x_0 \neq 0$. In both cases, the filtering of the domains occurs until the fixpoint is reached, but this process creates an inconsistency in the CSP containing the constraint $x_0 = 0$. The search finds the two solutions to the problem after 6 nodes have been explored.

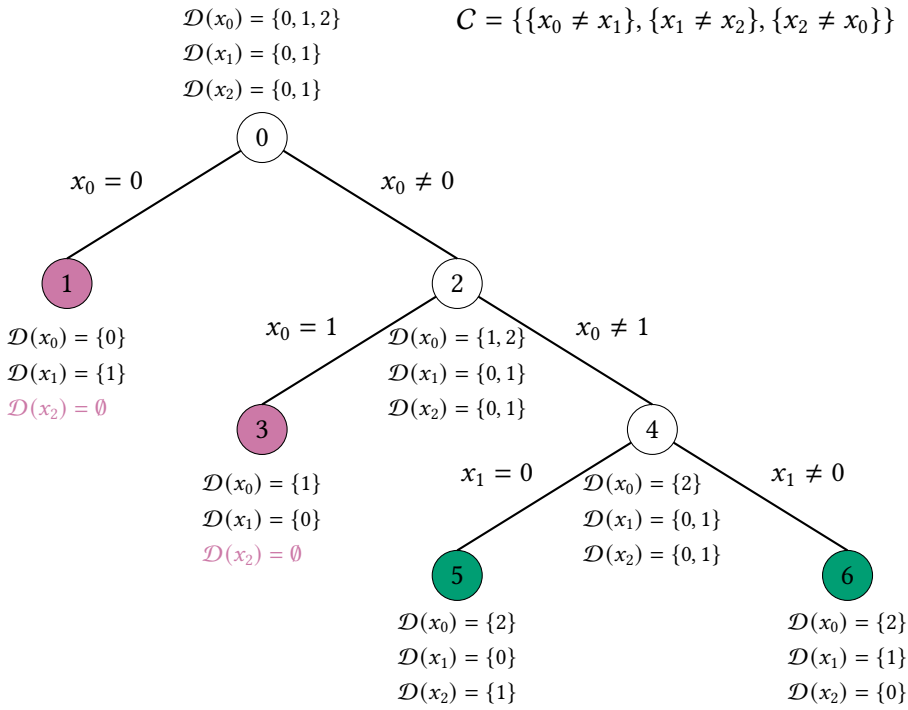


Figure 2.11: Possible search space exploration for Example 2.2.7. Each node corresponds to a CSP where the fixpoint has been reached. Numbers in the nodes correspond to their order of visit by a DFS. Nodes in green correspond to a solution, and nodes in purple to a failure (*i.e.* inconsistency in the represented CSP).

Perhaps the most common strategy used to partition the CSP, when working with integer variables, is to select an unfixed variable $x \in \mathcal{X}$ s.t. $|\mathcal{D}(x)| >$

1, select a value $v \in \mathcal{D}(x)$ within its domain, and generate two CSPs: the first one adding a constraint $c \leftrightarrow x = v$, and the second its negation $\bar{c} \leftrightarrow x \neq v$. In this setting, the strategy used to select an unfixed variable is referred to as *variable selection*, while *value selection* describes how to pick a value in the domain of the selected variable.

Not all variables need to be considered during the search to find a solution where all constraints hold. The user may identify a subset of the variables as *decision variables*, that are considered during the search procedure. Fixing the decision variables should fix the remaining variables once the fixpoint has been reached.

Solving a COP can be done by considering the CSP it contains. Assume that the objective f of a COP $\langle \mathcal{X}, \mathcal{D}, C, f \rangle$ is associated to a variable $z = f(\mathcal{X}^n)$, with $\mathcal{X}^n \subseteq \mathcal{X}$. Each time a solution σ for a CSP is found, where all decision variables are fixed, its cost $f_1 = f(\sigma^n)$, $\sigma^n = \{\sigma_i \mid \forall i \in \mathcal{X}^n\}$ is computed. Afterward, each subsequent CSP $\langle \mathcal{X}, \mathcal{D}, C \rangle$ considered by the DFS will also include a new constraint c_{f_1}, enforcing that the cost of the solution to this CSP must be better than the cost previously found: $\langle \mathcal{X}, \mathcal{D}, C \cup (c_{f_1} \leftrightarrow z < f_1) \rangle$. Once a solution with optimal cost f_n is generated in this manner, the constraint $z < f_n$ added on the remaining CSPs renders them infeasible. This proves the optimality of the solution f_n: no assignment can be produced with better cost.

The type of branching scheme considered plays a huge role in practice. Given a CSP to solve, the main goal consists in proving if it is satisfiable or not. Depending on the branching scheme performed, for the same initial CSP, the search tree might be composed of millions of nodes or a few hundreds, which impacts the solving time. Therefore, one is interested in generating a search tree which is as small as possible. In the same manner, when solving a COP with objective function f, finding a feasible solution having a low cost f_1 early in the search tree is also important. This first solution will add a constraint $f < f_1$ for all remaining CSP to consider, which may prune nodes in the search tree, preventing consideration of suboptimal solutions.

Example 2.2.12. Continuing example 2.2.11. By branching first on x_1 instead of x_0, a much smaller search tree is obtained, as shown in Figure 2.12. The DFS considered only 3 nodes compared to 6.

2.2.2.2 Search Exploration Principles

Three main principles used to reduce the size of the search tree considered are [Ref04]:

1. Given that all variables must be fixed, choosing the variable constraining the most the problem helps in generating small search space. This

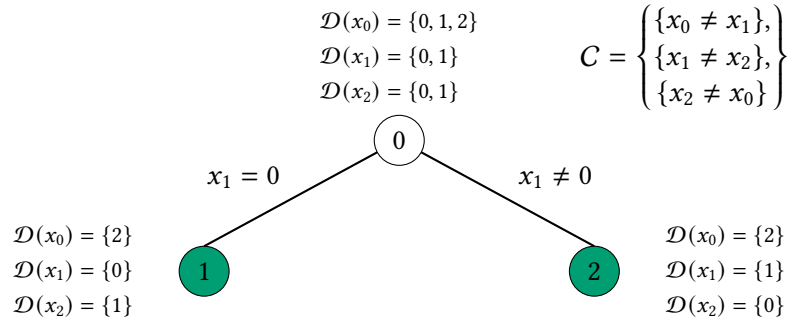


Figure 2.12: Possible search space exploration for Example 2.2.7. Numbers in the nodes correspond to their order of visit by a DFS. Nodes in green corresponds to a solution.

is also phrased in the *first-fail principle*: "To succeed, try first where you are most likely to fail" [HE80].

2. Regarding the choice of value, two situations may arise. If the problem is UNSAT, the choice of value does not matter much: all possibilities will eventually be considered. On the contrary, if the problem is SAT, a solution will be obtained faster if the chosen value maximizes the number of valid assignments for the remaining variables (in other words, if it constrains as least as possible the remaining search space to consider). In the case of a feasible COP, the value chosen should be the one most likely to appear in an optimal solution.

3. The first choices, performed near the root of the search tree, should be carefully chosen, as they are the ones having the most impact on the search tree size. Recall that the search tree is explored using DFS. If the CSP to solve is SAT, that the first left branch applied a constraint c leading to no solution without being detected, all subsequent (and UNSAT) CSP expanded after this first decision will be considered before coming back to the first decision.

The next sections present some existing variable and value selection heuristics applying those principles.

2.2.2.3 Variable Selection

Many variable selection heuristics attempting to follow the first-fail principle have been developed over the years. A complete overview of all variable selection heuristics falls outside the scope of this thesis, and there's a lack in the literature of a complete survey categorizing their features and performance. Some commonly encountered features of variable selections heuristics are

- **Domain size**, prioritizing variables with small domain size.
- **Number of failures**, selecting variables that often lead to a failure (i.e., one domain became empty when reaching the fixpoint) when branching on them.
- **Impact on search space**, selecting variables that have substantially reduced the search space when branching on them, due to filtering.

A non-exhaustive list of some popular variable heuristics using those features is presented next. The heuristics are described at a high-level; readers are referred to the corresponding paper for more details on the implementation or computational considerations.

- **MinDom** that selects the unfixed variable with the smallest domain size
- **DomWDeg** that selects the unfixed variable with the smallest ratio of domain size over its weighted degree. For a variable x, its weighted degree is defined as the sum of weights on the constraints with arity > 1 in the neighborhood of x. The weight of a constraint c itself is one plus the number of times it has emptied a domain when applying its related filtering. This heuristic was proposed in [Bou+04].
- **PickOnDom**, introduced in [ALP23] and relying on the same principle as DomWDeg, but where the weight is instead adjusted by tracking the filtering performed by the constraints.

  More specifically, a candidate weight increase with respect to a variable x is computed based on the number of values removed from its domain during the fixpoint computation. Each time a failure is encountered during the fixpoint computation, all candidate weight increases change the weight of their corresponding variable. Different types of increases are discussed in the paper, leading to different variations of this heuristic.
- **Impact-Based Search** that maintains a score for each variable, corresponding to its impact [Ref04]. The impact is defined as the contraction of the search space occurring when a variable is branched on, and is updated as branching decisions occur.

  More specifically, it relies on $\prod_{x \in \mathcal{X}} |\mathcal{D}(x)|$, the *Cartesian product of the domains*, which is an estimation of the search space size. The difference in estimated search space size before and after branching on a variable $x \in \mathcal{X}$ is used to set the impact of x. Variables having the largest impact are expected to reduce greatly the search space, and are selected first.

- **Activity-Based Search**, selecting the variable with the largest *activity* [MV12a]. The activity of a variable x corresponds to the number of times its domain was reduced during the branching and the fixpoint execution. A decay factor is also used, forgetting oldest statistics progressively.

- **Last Conflict**, proposed in [Lec+09], selects first the variable involved in the most recent failure. This heuristic is combined with a *fallback heuristic* when no selection can be made: at the beginning of the search when no failure has been encountered or when the variable in the most recent failure is currently fixed. This fallback heuristic is responsible for selecting a variable in such a case.

- **Conflict Ordering Search** that generalizes last conflict by adding a time stamp to every variable x, representing the latest time at which a failure was detected when branching on x. Variables with the highest timestamp are branched over first. It also needs to use a fallback heuristic and was proposed in [Gay+15].

Even though no survey covering those heuristics exists, [ALP23] provides more explanation on the implementation and behavior of some of them. They are also presented on a massive online open course [SMV25]. It's worth noting that no heuristic strictly dominates the other ones on all problems.

The list of variable selections heuristics listed here are *dynamic* heuristics: their choice depends on the current state of the problem (*i.e.* the current domains) and possible past information, such as the number of failures encountered. In contrast, *static* heuristics do not exploit such information. One example of a static heuristic is for instance to always iterate in a deterministic order over the variables $\mathcal{X}$, and select the first one being unfixed. Dynamic heuristics outperform static ones in the majority of cases.

2.2.2.4 Value Selection

Compared to variable selection, where many different heuristics have been developed over the years, there are fewer options regarding black-box value selection. Therefore, selecting the minimum value in the domain of a variable is often the default choice in CP solvers. Nevertheless, some alternative options are presented next.

Phase-Saving is a value heuristic that can be used for COP [DCS18]. It requires a past solution to be provided. When selecting a value to assign to a variable, it gives the one appearing in a previous solution. If no value can be selected (either because no previous solution exist, or because the value to

return does not belong to the variable's domain), a fallback heuristic is used instead. This heuristic is fast and enhances the performance when solving COP, but still requires a prior solution and a fallback heuristic.

Objective landscapes can also be used for COP [Lab18]. Assume that the objective f of a COP $\langle \mathcal{X}, \mathcal{D}, C, f \rangle$ is associated to a variable $z = f(\mathcal{X}^n)$, with $\mathcal{X}^n \subseteq \mathcal{X}$. The objective landscape of a variable $x_i \in \mathcal{X}^n$ is a function $L_i : \mathcal{D}(x_i) \to [\lfloor \mathcal{D}(z) \rfloor, +\infty)$ representing an optimistic estimation of the impact of assignment on the value of the objective z. When selecting a v value for x_i, the one associated with the smallest increase $L_i(v)$ on the objective is chosen first. An objective landscape is computed before the search begins, by observing the reductions on the bounds of the variables $x_i \in \mathcal{X}^n$ when shrinking the domain of the objective variable z. Such functions are mainly used in scheduling scenarios and are one of the components present in CP Optimizer [Lab+18b].

Example 2.2.13. Figure 2.13 shows the objective landscape of a variable x_i.

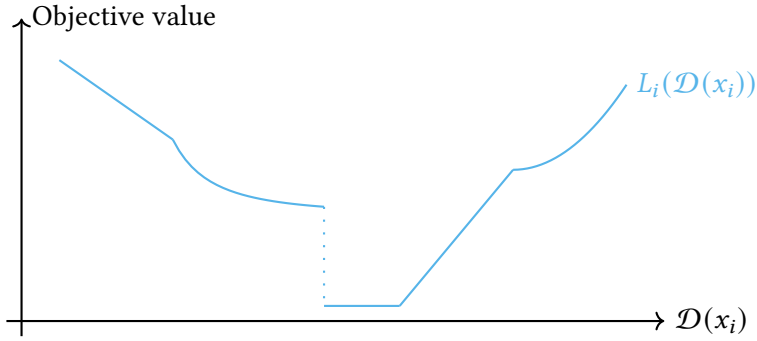


Figure 2.13: Objective landscape L_i of a variable x_i.

While effective in scheduling, three practical points limit the performance of objective landscapes on other types of problems. Firstly, they are estimated before the search begins, based on domain reductions of the objective. Domain reductions of other variables are not represented in those functions; in other words, they do not encode how values should be selected based on the remaining domain of other variables. Secondly, the related paper [LR08] only presents how to compute them for variables $\mathcal{X}^n$ directly involved in the objective function. Some consideration regarding non-objective variables $\mathcal{X} \setminus \mathcal{X}^n$ are briefly presented, but without proposing a computation of their objective landscape. Thirdly, the construction process detailed in the paper implies

that they are *quasi-convex*[1]. On a TSP, this means that they cannot accurately represent the best values for a successor variable, as such functions are not quasi-convex.

Activity-Based Search can also be used to act as a value heuristic [MV12a]. In this case, it computes an activity corresponding to an assignment $x = v$ as the number of variables whose domain has been reduced when applying it. The value $v \in \mathcal{D}(x)$ with the smallest associated activity (*i.e.* the value expected to modify the fewest variables) is selected first. A similar process can be used with **Impact-Based Search** [Ref04], selecting values associated with the smallest impact. Two main drawbacks of this approach are the space requirements, as activity related to past assignments must be stored, and the absence of information at the root node of the search tree, when no activity has been observed yet.

2.2.2.5 Miscellaneous

The heuristics presented thus far assumed that the branching scheme proceeds in two steps: first selecting an unfixed variable, then a value to assign to it. Other strategies exist, sometimes specifically designed for particular types of problems. Some of them are presented next.

Failure Directed Search : Failure Directed Search (FDS) is a search procedure present in CP Optimizer [VLS15]. Instead of working with variable and value selection, this search procedure works with a set of *choices*. A choice can be seen as a constraint used for the search space exploration, and may be negated. Given a variable x in the problem and a value v in its domain, a choice could be $x \leq v$ and its negation $x > v$. At a high level, FDS maintains a rating for each choice, increased based on the search space reduction and on the failures provoked when applying the choice. Choices with the highest rating are applied first during the branching, with the aim to close the search space as soon as possible and prove that the problem is UNSAT.

Given that FDS is used in scheduling context, the set of choices is mainly related to interval variables. In particular, the initial set of choices generated ensure that, if all performed, each optional task interval variable present in the problem is either absent or contains a mandatory part (see Figure 2.10). Once no choice can be applied, either the variables are fixed and a solution has been found, a DFS can be used over the remaining problem, or more choices can be generated at that point.

[1] A function $f : \mathbb{R} \to \mathbb{R}$ is quasi-convex if for all $x, y \in \mathbb{R}$ and for all $\lambda \in [0, 1]$, we have $f(\lambda x + (1 - \lambda)y) \leq \max(f(x), f(y))$.

FDS is well suited for proving that the problem is infeasible, and therefore plays a key role in proving optimality for COP. However, it is not designed to find a solution of good quality. Therefore, CP Optimizer combines FDS with another strategy suited for finding improving solutions, Large Neighborhood Search (LNS), whose process is detailed in Section 2.3. The search procedure alternates between LNS phases to find solutions, and FDS phases to prove their optimality.

Limited Discrepancy Search : Limited Discrepancy Search [HG95] is a search procedure well suited when an informed value heuristic is provided. It assumes that the best decisions are taken on the left branches during the search tree exploration, and therefore limits the number of right branches considered. This search procedure takes as input a maximum discrepancy (*i.e.* number of right decisions) to tolerate and explore the search space using DFS, except that search nodes exceeding the provided discrepancy are not explored. This process is highlighted in Figure 2.14.

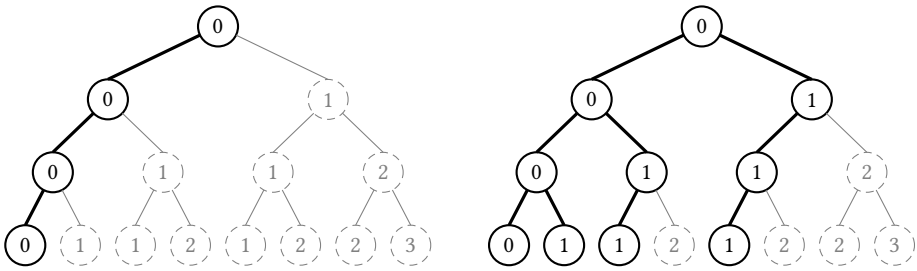


Figure 2.14: Limited Discrepancy Search, with a limited discrepancy of 0 (left) and 1 (right). The discrepancy is indicated at each node. Nodes in gray are ignored by LDS as their discrepancy is too high.

2.3 Large Neighborhood Search

On large problems, exploring the whole search space is impractical as it takes too much time. In such a situation, a compelling alternative to DFS consists in exploring diversified regions of the search space, covering only a subset of the whole search space. Large Neighborhood Search is a meta heuristic with this behavior [Sha98]. At a high level, it behaves as follows. It takes as input a feasible solution, and creates a relaxed solution from it. This relaxed solution is obtained by keeping a fragment of the assignments (or structure) from the solution, and leaving the remaining variables unfixed. From this relaxed solution, the remaining problem is then solved using a search strategy such

as DFS to fix the remaining variables. The process thus alternates between relaxation phases and optimization phases.

In the context of VRPs, the original paper [Sha98] relaxes a solution by removing some nodes from the obtained paths. The removed nodes are then reinserted into the vehicles paths in order to optimize the routing plan. This process is illustrated in Figure 2.15. In terms of search space exploration, LNS can be seen as "jumping" between different parts of the search space: the CSP considered after relaxation is located in a different region of the search space. This phenomenon is illustrated in Figure 2.16 (adapted from [SMV25]): on a large problem and given a time limit, DFS may only explore a small portion of the search space. In contrast, LNS allows covering more diversified regions of the search space.

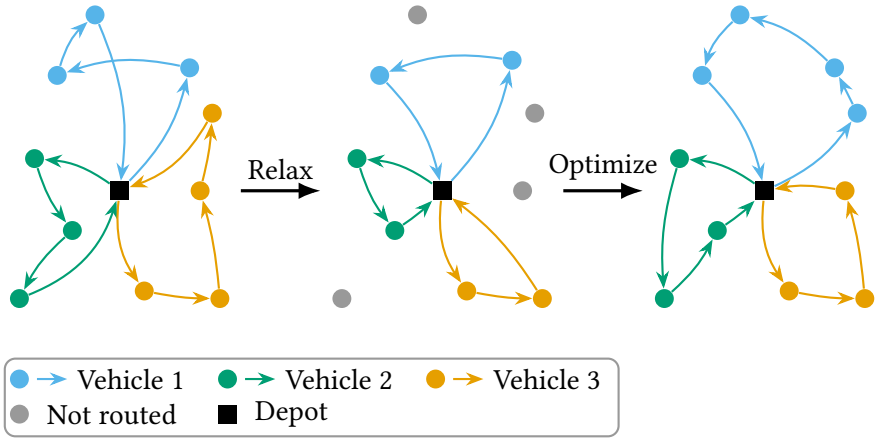


Figure 2.15: LNS behavior on VRPs. From an initial solution (left), a relaxed solution is constructed by removing some visits (middle). This relaxed solution is then optimized by inserting the removed visits (right).

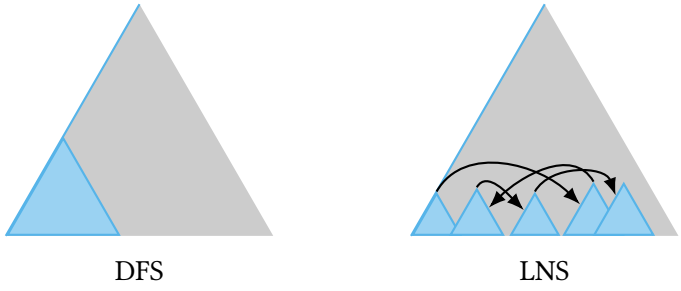


Figure 2.16: DFS vs. LNS behavior in terms of search space explored (blue).

The two components of LNS, the relaxation and optimization, play a critical role in its performance. Different relaxations are possible: relaxed vari-

ables may be chosen randomly, similarly [Sha98], by removing successive visits in one or several paths [CV20], etc. Regarding node insertion, they can be done using greedy decisions (inserting the nodes with the smallest detour first), using regret heuristics, etc. After relaxation, the optimization can also use LDS instead of DFS [HG95; Sha98].

An extension of LNS is the Adaptive Large Neighborhood Search (ALNS) [TS18; RP06; TSH21], where different operators for relaxation and optimization are provided. ALNS dynamically selects the relaxation and optimization operator to use in one iteration based on the past performance observed. ALNS was originally proposed in [RP06]. Additional extensions of LNS are discussed in [PR18].

Another line of research on LNS studies how to automatically choose a suitable set of variables to relax, attempting to exploit the structure of the problem. The core idea is to relax variables that are identified as related. Works such as [GLN05] identify related tasks to relax together based on their ordering in scheduling contexts and relax successive tasks, [PSF04] chose variables to relax based on domain reductions, identifying links between variables due to propagation, and works such as [LS14] relax variables that are the most expected to have an impact on the objective value, also exploiting reductions of domains.

LNS is particularly well suited for complex and large VRPs, where a declarative model can be written using CP, but where the performance of the CP solver does not allow exploring the full search space.

2.4 Challenges

With the background now laid out, some challenges are worth highlighting when attempting to solve VRPs using CP. They are briefly presented in this section, and the corresponding research questions tackled by this thesis are stated afterwards.

2.4.1 Black-Box Search Heuristics

When solving VRPs, the value selection heuristics contribute significantly to the performance. Take for instance the solving of a TSP. Some simple heuristics attempt to create a TSP tour using a nearest neighbor strategy, selecting the closest city as the successor of a node within a TSP tour. However, this simple value heuristic is not commonly found in a black-box fashion within CP solvers[2]. Most solvers still rely on choosing the minimum value in the domain of the variables, which has no regards for the impact on the distance, and provides solutions of lower quality than nearest neighbor strategies.

[2]Section 3.1 dives deeper into previous attempt to create such value selection.

2.4.2 Insertions and Large Neighborhood Search

In the LNS introduced in section 2.3, the relaxation phase removes some nodes from the routing plan, and the optimization attempts to reinsert them. This behavior with node reinsertion is not so trivial to enforce within a CP solver, as a successor model using a circuit constraint does not easily support insertion heuristics.

LNS commonly used in CP solvers keeps some variable assignments from a previous solution. In the context of VRPs, assuming successor variables and (sub)circuit constraint, this is equivalent to keeping some successor assignment from a previous solution, and the optimization process fixes the remaining successors in the problem. With this behavior, insertions such as in Figure 2.15 are not feasible: once a successor variable is fixed due to the LNS relaxation, it cannot be reassigned until another LNS iteration is performed. Instead of inserting the remaining nodes into existing paths, the solver deals with several chains of successors, and may only fix the unassigned successor variables without altering the initial chains of successors. This process is illustrated in Figure 2.17. Compared to the original LNS presented in section 2.3, less flexibility is offered to find a new solution, and the obtained objective value may be more costly than the one found with the original LNS.

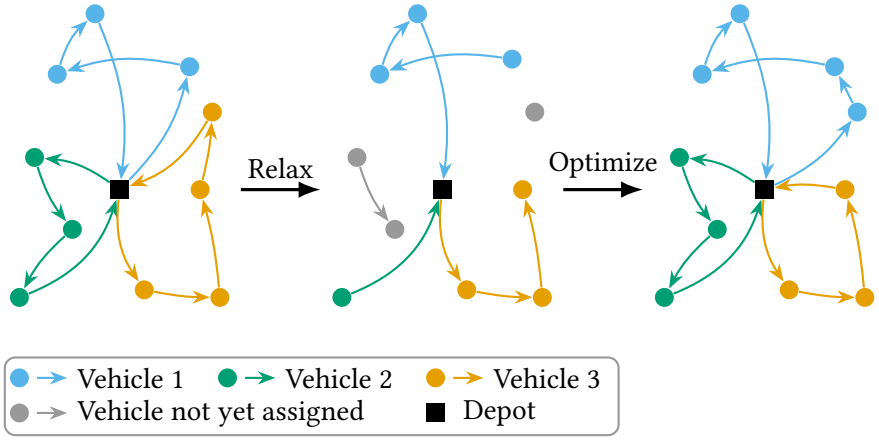
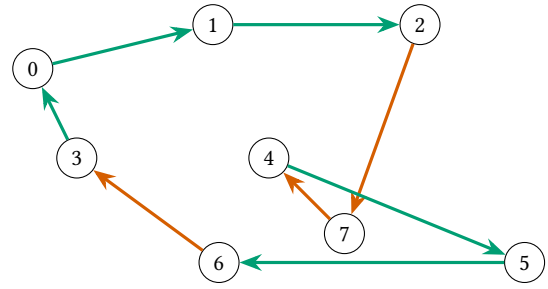


Figure 2.17: LNS behavior on a VRP with successor variables. From an initial solution (left), a relaxed solution is obtained by keeping some successor variables assignments (middle). This relaxed solution is then optimized by fixing the remaining successor variables (right).

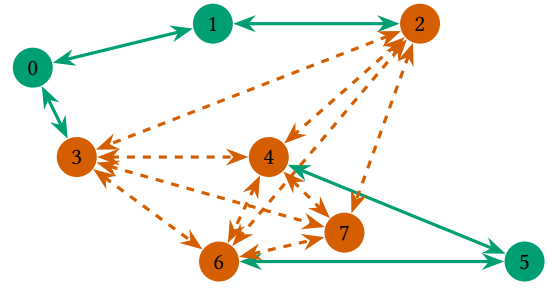
Some approaches such as [TS18] have instead proposed a relaxation specifically for the successor model, providing more flexibility than the behavior illustrated in Figure 2.17. In their work, a k-opt relaxation is proposed, which removes some edges (*i.e.* relax successor variables) and restrict the remaining successor variables to only include their current successor and predecessor

from the best solution in their domain. This behavior is shown in Figure 2.18 (adapted from [Tho23]). But even this specific relaxation does not allow for insertions at an arbitrary place within the path given that non-relaxed successor variables have their domain being restricted.



Variable	Domain
$succ_0$	$\{1\}$
$succ_1$	$\{2\}$
$succ_2$	$\{7\}$
$succ_3$	$\{0\}$
$succ_4$	$\{5\}$
$succ_5$	$\{6\}$
$succ_6$	$\{3\}$
$succ_7$	$\{4\}$

Relax



Variable	Domain
$succ_0$	$\{1, 3\}$
$succ_1$	$\{0, 2\}$
$succ_2$	$\{1, 3, 4, 6, 7\}$
$succ_3$	$\{0, 2, 4, 6, 7\}$
$succ_4$	$\{2, 3, 5, 6, 7\}$
$succ_5$	$\{4, 6\}$
$succ_6$	$\{2, 3, 4, 5, 7\}$
$succ_7$	$\{2, 3, 4, 6\}$

Figure 2.18: K-opt relaxation from [TS18], for a successor model.

If one wants to allow insertions at an arbitrary place within the path, this may be obtained by reasoning on other kinds of variables than successor variables. For instance, fixing precedences between nodes (*i.e.* working with boolean variables $p_{i,j}$ indicating if a node i precedes a node j) instead of direct successor assignments may help to provide more flexibility, and find better successor assignments in the optimized solution. However, it requires further processing of the solution and may introduce more variables.

2.4.3 Optional Visits

Dealing with optional visits is not always straightforward on complex VRPs. Even though the subcircuit constraint allows representing nodes outside a vehicle path, a difficulty may arise depending on the way some constraints are formulated, as constraints need to be aware that self-loops are permitted and designate an unvisited node.

Take for instance the case of a VRP involving V nodes, temporal constraints and optional visits. Assume that each node $v \in V$ is associated to a service duration $d_v > 0$, and that a matrix $D^{V \times V}$ defines transition times between nodes. The visit time of a node v is captured in a variable T_v, and a variable S_v defines its successor. The channeling of visit time can be expressed by

$$T_{S_v} = T_v + d_v + D_{v,S_v} \tag{2.14}$$

which constrains the time visit T_{S_v} of the successor S_v of node v. If the transition matrix D contains only positive entries, this implies $T_{S_v} > T_v$. Given that the visits are optionals, the subcircuit is used on the successor variables S. However, this provokes a failure in case nodes are not visited: setting $S_v = v$ (for a self-loop) violates the implied constraint $T_{S_v} > T_v$.

This problem can be mitigated by either (i) adapting the input, setting particular values for self-loops within the transition matrix (for instance $D_{v,v} = -d_v \ \forall v \in V$), or (ii) using boolean variables tracking if the node is visited or not (replacing (2.14) by $(S_v \neq v) \Leftrightarrow T_{S_v} = (T_v + d_v + D_{v,S_v})$). The latter case increases the computation time, as it may introduce more variables (for instance boolean variables tracking expressions of the form $S_v \neq v$), and adds more constraints.

2.4.4 Research Questions

Given the challenges introduced previously, this thesis attempts to answer the following research questions:

1. **How to efficiently automate a nearest neighbor value selection on VRPs?**
2. **How to integrate LNS with insertions inside CP solvers?**
3. **How to efficiently deal with optional visits when solving VRPs with CP?**

The first research question is studied in Section 3, by presenting corresponding value selections that are both fast and generic. The last two ques-

tions are tackled in Section 4, introducing sequence variables, specifically designed to solve VRPs while coping with insertions and optional visits.

Value Heuristics

3

This chapter attempts to answer the first research question: *How to efficiently automate a nearest neighbor value selection on VRPs?*

Although CP allows defining VRP models in a declarative way, its performance are behind that of other methods. Part of the explanation for CP inefficiency is the way values are assigned to variables during the search. Existing CP value heuristics in black-box are either fast, but do not steer the search in promising direction, or are slow, and their gain in search guidance may not outweigh the induced computational overhead. This is observed in works such as [Cap+21], where the cost of selecting an informed value through a neural network is identified as a bottleneck, and in [FP17], where the fastest value heuristics fail to deliver the optimal solution on some problems compared to slower.

This can be directly seen on the TSP. Given a TSP instance with n cities and a distance matrix $d \in \mathbb{Z}^{n \times n}$, a commonly used CP model is

$$\min TotD \tag{3.1}$$

subject to:

$$\text{circuit}(S) \tag{3.2}$$

$$D_i = \text{element}(d_{i,*}, S_i) \qquad \forall i \in \{0..n-1\} \tag{3.3}$$

$$TotD = \text{sum}(D) \tag{3.4}$$

It introduces two variables per city: S_i is the visit occurring after the city i in the tour, and D_i the distance between city i and its successor S_i. The circuit constraint enforces every city to be visited within a single tour (3.2). The distance D_i between a city i and its successor S_i corresponds to the S_i-th entry within the line i in the distance matrix d, enforced with an element constraint (3.3). The sum of traveled distance $TotD$ is the objective to minimize (3.1), (3.4).

The S variables can be branched on in order to find a solution. However, if a simple value heuristic method such as MinDom is used, the first found solution will have an objective value far from the optimum: successor of the cities are chosen based on their index, which has no regard for the length of

the TSP tour. In contrast, a hand-coded heuristic selecting the nearest neighbor as the successor of the city would find a first solution of better quality, and also prove optimality faster.

Example 3.0.1. Figure 3.1 shows the behavior between MinDom and a hand-coded heuristic selecting the nearest neighbor of a city as its successor, on a TSP instance. The first branching decision and the first solution reached are shown, both of which are obviously better when using a closest successor heuristic.

The variable selection is assumed to select the city with the smallest index first. On a TSP instance, given that no failure occurs before finding the first solution, and that the domain sizes are the same, dynamic variable selection might default to such selection.

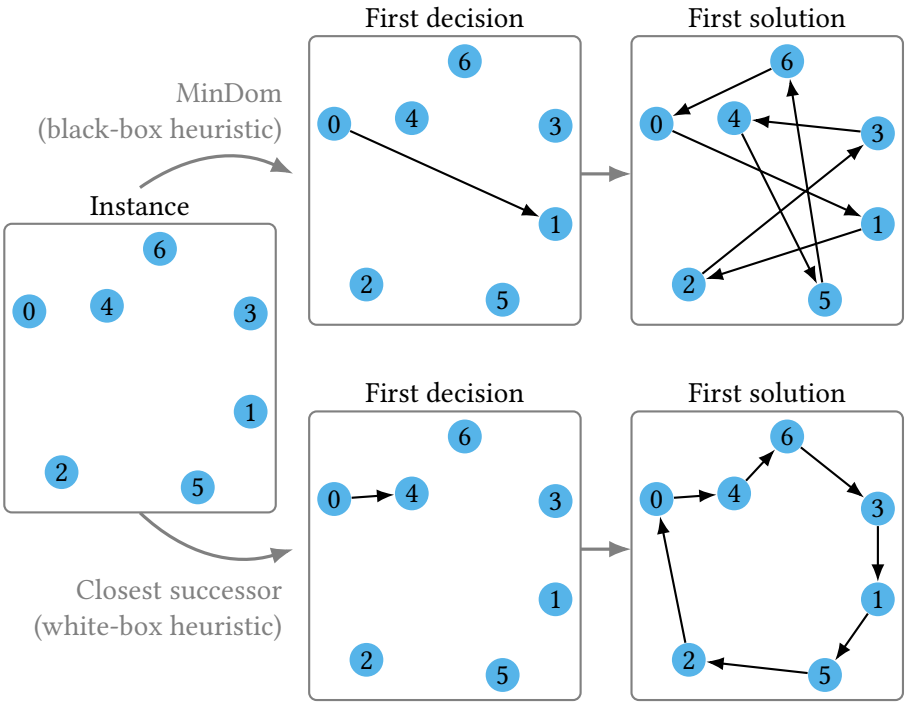


Figure 3.1: Given a TSP instance (left), the first decision taken by MinDom (top) and a closest successor value heuristic (bottom). The first solution generated by each method is shown on the right.

A natural question that arises is whether one can automate the selection of the nearest successor inside a CP solver, instead of having to manually write a search procedure. The answer is yes. This chapter presents black-box value heuristics developed in the thesis, that not only select the nearest successor on

a successor model, but also enhance performance on non-routing problems. Given that many VRPs are modeled in CP on top of a successor model, by adding more constraints on it, those heuristics also automate nearest neighbor selection on those models. One closely related approach is first described in section 3.1. Section 3.2 then covers the heuristics developed in this thesis, which were published in A. Delecluse and P. Schaus. "Black-Box Value Heuristics for Solving Optimization Problems with Constraint Programming (Short Paper)". In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024

Compared to the publication, this chapter provides more details on the algorithms involved, implementation considerations, and discuss how they may be applied on problems without an objective function.

3.1 Existing work: Bound-Impact Value Search

Bound-Impact Value Search (BIVS) [FP17] is a value heuristic for optimization problems, choosing the value with the best impact on the objective. Its behavior is detailed in Algorithm 1. BIVS examines each value v in the domain of the selected variable $x \in \mathcal{X}$ (line 3), assigning v to x (line 5) and running the fixpoint algorithm (line 6). If the assignment does not lead to a failure and increases the objective's lower bound ($\lfloor \mathcal{D}(obj) \rfloor$), the value is considered for retention (line 7). The process is encapsulated within `saveState`($\mathcal{X}, C$) and `restoreState`($\mathcal{X}, C$) operations (lines 4 and 10), ensuring that the solver's state is reset before each new trial. In the original paper, the authors show that this approach manages to automate the nearest neighbor selection on the TSP.

3.2 Reducing Bound-Impact Value Search Cost

Although BIVS is effective in steering the search towards high-quality solutions, the time complexity for selecting a value for variable x in the BIVS algorithm is $\Theta(\mathcal{F} \cdot |\mathcal{D}(x)|)$, with $\mathcal{F}$ as the fixpoint algorithm's complexity. As [ALP23] notes, controlling this high computation cost is challenging. This section proposes two methods to reduce this complexity: one by lowering the cost of the fixpoint algorithm and the other by reducing the frequency of its calls. However, these changes do not guarantee identical outcomes as the original BIVS.

Algorithm 1: Bound-Impact Value Selector, adapted from [FP17].

Input : $\mathcal{X}$: variables, C: constraints, x: branching variable, obj: objective variable
Ouput: bestV, the value to assign to the variable x.

```
1  bestV ← ⌊D(x)⌋
2  bestBound ← ∞
3  for v ∈ D(x) do
4  |   saveState()
5  |   D(x) ← {v}
6  |   success ← fixpoint(X, C)
7  |   if success and ⌊D(obj)⌋ < bestBound then
8  |   |   bestBound ← ⌊D(obj)⌋
9  |   |   bestV ← v
10 |   restoreState()
11 return bestV
```

3.2.1 Restricted Fixpoint

The Restricted Fixpoint (RF) approach assesses the impact on the objective by focusing solely on a limited set of constraints. Specifically, it considers only the constraints that are located on the shortest paths from the selected variable to the objective variable within a bipartite variables-constraints graph. In this graph, variables and constraints are nodes, connected by edges if the variable is within the scope of the constraint. For each variable, we precompute all constraints on these paths. This method reduces computation costs as it involves fewer constraints, but it may be less informative and risk missing potential failures.

Example 3.2.1. Consider the TSP model from (3.1)-(3.4). A TSP with 4 cities is shown in Figure 3.2 and its variables-constraint graph in Figure 3.4. When selecting values for S_0 using RF with BIVS, only 2 constraints are considered per iteration, compared to 6 without RF.

3.2.1.1 Implementation

The RF may be implemented in several ways inside a solver. Some considerations regarding the shortest path computation are first covered before diving into the RF itself .

Shortest Paths Regarding the shortest paths between each variable and the objective, they are computed once, at the root node of the search. Those

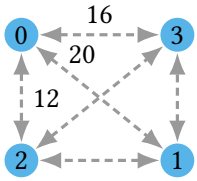


Figure 3.2: A TSP instance with 4 cities to visit. The distances are shown on the edges.

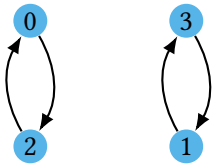


Figure 3.3: When fixing $\mathcal{D}(TotD)$ to $\{48\}$, all successors points towards their nearest city. This violates the circuit constraint.

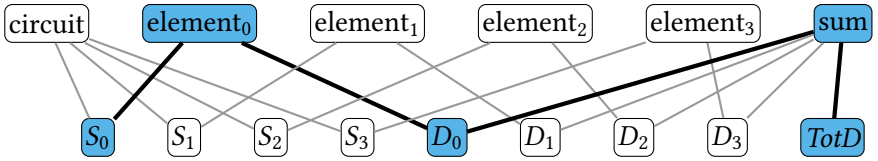


Figure 3.4: Constraints and variables of the TSP instance from Figure 3.2. Variables and constraints in blue are located on the shortest path to the objective $TotD$ when considering variable S_0.

paths do not change under the assumption that no constraint with arity[1] > 1 is added onto the problem during the search exploration, which is commonly the case in CP solvers. In other situations, with a specific branching scheme that adds constraints with arity > 1, recomputing the shortest paths may be worthwhile.

One naive way to retrieve the relevant constraints related to each variable would be to store, for each variable, a set of all constraints on the shortest path. Obviously, this approach consumes a lot of memory on problems with a large constraint network.

To reduce the memory footprint, retrieving the constraints on the shortest paths between each variable and the objective is instead implemented using a shortest path tree. Each node in the constraint network contains a list of their direct successors in the shortest paths leading to the objective. Retrieving the subset of relevant constraints can then be done by traversing the stored shortest paths. This process can be computed using a breadth-first search over the constraint network, starting from the objective to minimize. The time spent computing this shortest path tree is negligible compared to the solving time of a COP instance.

Restricted Fixpoint For the restricted fixpoint, they are two main ways to implement it: a non-intrusive method that does not require a rewrite of

[1] the arity of a constraint is the number of variables in its scope.

existing trail-based CP solvers, and an intrusive one where a new method must be added to the solver API.

1. The **non-intrusive method** relies on the temporary deactivation of constraints. For performance reasons, solvers typically support such deactivation, for instance when it can be detected that no filtering may be performed by a constraint anymore (*e.g.* a constraint enforcing $x < y$ with $\mathcal{D}(x) = \{1, 2\}, \mathcal{D}(y) = \{3, 4\}$ cannot perform any future filtering, and may be deactivated even though the variables in its scope are not fixed). Constraint deactivation is considered a reversible operation, which is the case in MiniCP and Choco-Solver [MSV21; PF22]. Using this deactivation, a restricted fixpoint may be implemented by either:

   - First storing all constraints along the shortest paths in a set, then looping over all constraints registered in the solver, and deactivating the ones outside the set.
   - Traversing the shortest paths using a Breadth-First Search, starting from the branching variable and going to the objective. Whenever the BFS processes a variable, all constraints attached to it are examined. If a constraint does not lie on the shortest path to the objective, it is deactivated. Whenever the BFS processes a fixed variable, or a constraint already deactivated, it is skipped, as no filtering may occur by considering it.

   This deactivation of irrelevant constraints w.r.t. the RF is surrounded by save and restore operations, to ensure that constraints remain active after the RF computation.

2. The **intrusive method** needs to enrich the solver API by implementing a new function. This function takes as input a list of propagators to consider, and triggers a fixpoint considering only those propagators, while ignoring the other ones registered in the solver. This can be implemented by introducing a restricted fixpoint mode in the solver. If the solver is in restricted fixpoint mode, only constraints in the provided list (the shortest paths in our case) are triggered.

The first approach is more suited when the internals of the solvers cannot be changed. The second one is more effective as it does not require the temporary deactivation of constraints, which adds a low but noticeable overhead on large constraint networks, due to the parsing of additional propagators to deactivate.

Given that the last conflict heuristic is commonly used in CP solvers [Lec+09], whenever a failure occurs, a variable is branched twice in a row. Therefore, it might be worthwhile to use caching, and store the shortest path computed

at the last call. When querying the shortest paths for a variable x, they may be retrieved in constant time in such situations.

Given that this approach modifies successively small portions of the domains, and on a small set of variables, it is more suited in solvers relying on trailing. With copying, the cost of repeatedly creating many CSPs at each search node would prevent the usage of this heuristic, which is also the case with the original BIVS.

3.2.2 Reverse Look-Ahead

The Reverse Look-Ahead (RLA) strategy reduces the number of calls to the fixpoint computation by restricting optimistically the domain of the objective and observing the effects on the variable domain $\mathcal{D}(x)$, rather than fixing x directly. It is similar to the Destructive Lower Bound used in scheduling [KS99] and can also tighten bounds on the objective.

Algorithm 2 outlines RLA. A value δ controls the domain size of the objective variable during the fixpoint computation, starting with a value of 1 to fix the objective to its minimum value. Several iterations may be performed, each increasing δ until the fixpoint computation succeeds. At this point, the minimum value from the domain of x is returned at line 11. The value of δ doubles at each iteration, resulting in an exponential evolution. Note that when the fixpoint computation fails, the lower bound of the domain of the objective variable can be safely increased (line 14).

Example 3.2.2. Consider the same situation as Example 3.2.1, shown in Figure 3.2. The initial fixpoint yields $\mathcal{D}(D_i) = \{12, 16, 20\}\ \forall i \in \{0..n-1\}$ and $\mathcal{D}(TotD) = \{48, \ldots, 80\}$. When using RLA to choose a value for S_0, the following iterations occur:

1. The fixpoint is triggered with $\mathcal{D}(TotD) = \{48\}$. This means that the successor of every city must be the closest neighbor, violating the circuit constraint (cf Figure 3.3). The iteration fails and the lower bound of the objective is now set to 49 for the subtree to consider.

2. $\delta = 2$ and $\mathcal{D}(TotD) = \{49, 50\}$. Similarly, this fails and sets the lower bound to 51.

3. $\delta = 4$ and $\mathcal{D}(TotD) = \{51, \ldots, 54\}$. The fixpoint proceeds without failure, resulting in $\mathcal{D}(S_0) = \{2, 3\}$. Value 2, the nearest neighbor, is picked and the state is restored while keeping the lower bound of the objective to 51. Finally, 2 is returned (lines 9 to 11).

The time complexity of RLA is $\Omega(\mathcal{F})$ in the best case, if only one iteration needs to be performed, and $O(\mathcal{F} \cdot \log_2 |\mathcal{D}(obj)|)$ in the worst case. Moreover,

Algorithm 2: Reverse Look-Ahead

Input : $\mathcal{X}$: variables, C: constraints, x: branching variable, *obj*: objective variable
Ouput: *success*: boolean indicating search node expandability, v: assigned value for x

```
1  δ ← 1
2  success ← true
3  while success do
4      saveState()
5      ⌈D(obj)⌉ ← min(⌈D(obj)⌉, ⌊D(obj)⌋ − 1 + δ)
6      success ← |D(obj)| > 0
7      if success then
8          if fixpoint(X, C) then
9              v ← ⌊D(x)⌋
10             restoreState()
11             return (true, v)
12         else
13             restoreState()
14             ⌊D(obj)⌋ ← ⌊D(obj)⌋ + δ
15             success ← |D(obj)| > 0
16             δ ← δ ∗ 2
17     else
18         restoreState()
19 return (false, 0)
```

RF can also be used with RLA, meaning that the complexity of the fixpoint can be lowered.

Example 3.2.3. We reuse the model from Example 3.2.2. Initially, RLA restricts $\mathcal{D}(TotD) = \{48\}$ and runs the RF. This scenario suggests all successors should be nearest neighbors, which is infeasible given the circuit constraint (see Figure 3.3). However, when considering only the shortest path constraints (the sum constraint and one element constraint, cf Figure 3.4), the failure is missed. Consequently, the domain of S_0 becomes $\{2\}$ (its closest neighbor), and 2 is returned. Compared to the scenario in Example 3.2.2, only one iteration has been performed (being less costly) but the heuristic itself could not tighten the bounds of the objective.

An attentive reader would say that the previous examples on the TSP are not 100% convincing. Indeed, it suffices to sort the city successors before the beginning of the search based on their distance, and use as value selection

the first available successor in the sorted list. This is because the theoretically best value for each variable in the TSP, according to its direct impact on the objective, does not depend on the values that other variables may take, and can be computed at the root node of the search tree. This is akin to the objective landscapes presented in [Lab18] (which were not directly applicable on the TSP, as explained in section 2.2.2.4).

However, the presented heuristics actually reconsider the best value at each node in the search tree, allowing them to perform informed decisions not only on the TSP but on other problems as well. For instance, in problems with time-dependent transitions, the best successor for a city depends on the time at which the departure occurs. Therefore, there is no absolute best successor for the successor variable: it depends on the departure time. The impact of reconsidering the best value at each node is better illustrated in the next example (which is not a VRP but was chosen for its relative simplicity).

Example 3.2.4. Let us consider a variant of the bin packing problem. It consists in placing I items within B bins, each item $i \in \{0, \dots, I-1\}$ having a given weight w_i. The objective is to minimize the maximum load occurring in any bin. A CP model for this problem can be as follows:

$$\min maxL \tag{3.5}$$

subject to:

$$maxL = \text{maximum}(L_0, L_1, \dots, L_{B-1}) \tag{3.6}$$

$$L_b = \sum_{i=0}^{I-1} w_{b,i} \qquad \forall b \in \{0..B-1\} \tag{3.7}$$

$$w_{b,i} = w_i * y_{b,i} \qquad \forall i \in \{0..I-1\} \forall b \in \{0..B-1\} \tag{3.8}$$

$$y_{b,i} \leftrightarrow (x_i = b) \qquad \forall i \in \{0..I-1\} \forall b \in \{0..B-1\} \tag{3.9}$$

The main decision variables are the integer variables x_i, each having initial domain $\{0, \dots B-1\}$, and representing in which bin item i must be placed. The presence of item i in bin b is represented by a boolean variable $y_{b,i}$ through a reified equality constraint (3.9): $y_{b,i} = 1$ if and only if $x_i = b$. Each boolean variable $y_{b,i}$ is used to capture the weight $w_{b,i}$ occupied by item i in bin b: either w_i if $y_{b,i} = 1$, and 0 otherwise (3.8). The total load L_b in a bin b is summed over the weights w_{bi} (3.7). The objective consists in minimizing the maximum load (3.5), computed based on the load L_b of each bin b (3.6).

The best value to assign to a variable x_i depends on the current load in the bins. Therefore, choosing in advance the best values for each variable at the root of the search tree is irrelevant: when all bins are empty, they are equally valuable and no value dominates the other ones. Therefore, MinDom

or an initial ranking should both favor to put items in the bin with the smallest index (assuming that ties are broken by selecting the smallest value). In contrast, BIVS(+RF) and RLA(+RF) would both favor to put items in a bin that increases the maximum load by the smallest amount. This is shown on Figure 3.5: given an instance to solve, the first decision with any heuristic is to put the first item in bin 0. With the second decision, MinDom or an initial ranking keeps putting items in bin 0, while BIVS(+RF) and RLA(+RF) will try to spread out the items across the bins.

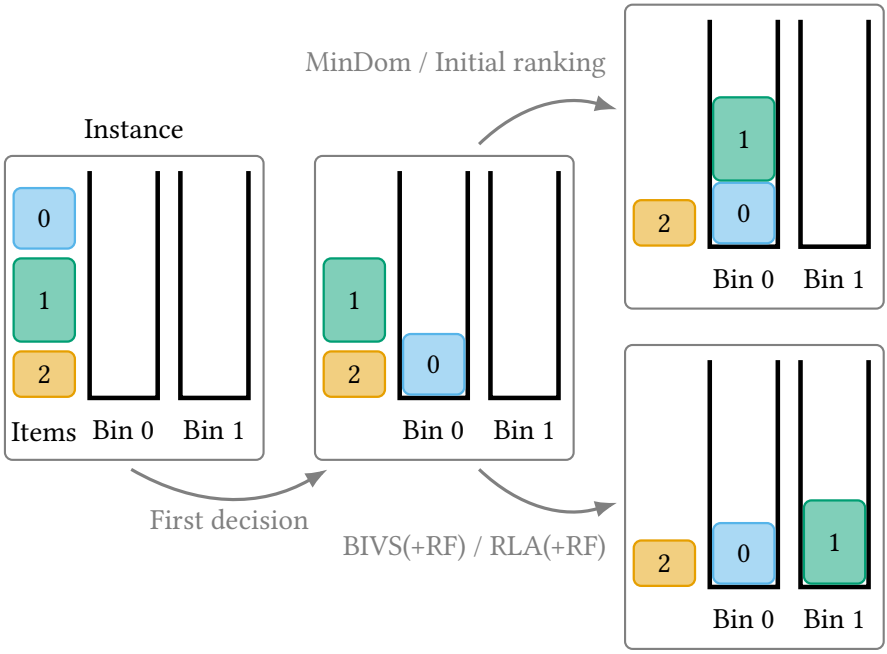


Figure 3.5: Value heuristics behavior on a bin packing instance.

Note that a global BinPacking constraint [Sha04] is commonly present in modern solvers. The behavior of BIVS(+RF) and RLA(+RF) would be the same as using this global constraint instead of the presented model.

3.3 Experiments

To assess the performance of the methods, two main settings were considered. The first one analyzes the TSP and two other classical discrete problems easily modeled with CP. The second one reports the performance on various optimization problems, using the XCSP^3 2023 competition [Bou+16b; ALL23].

The implementation was done in Java in the Choco-Solver (version 4.10.5), a state-of-the-art general purpose constraint programming solver [PF22]. In all settings, instances needing more than 16GB were discarded. All experi-

ments were conducted using two Intel(R) Xeon(R) CPU E5-2687W in single-threaded mode. DFS was used, with last conflict [Lec+09] with DomWDeg [Bou+04] as the fallback heuristic for the variable selection heuristic, a popular default selection. Ties for the value selection are broken by selecting the smallest value. Ties for the variable selection are broken randomly and the same seed for random number generation was used. The timeout was set to 30 minutes. Experiments were run in parallel using GNU Parallel [Tan21].

3.3.1 Fundamental Problems

Three fundamental problems are studied. (i) The TSP and the instances from TSPLib [Rei91]. (ii) The JobShop and the instances from [VRF15]. (iii) The Quadratic Assignment Problem and the instances from [24c]. For each model, the standard models are used. The JobShop model branches on precedences like in [GHM09]. A precedence variable b_{ij} telling if task i is executed before task j only exists if the id of task i is smaller than the id of task j (i.e., $i < j$). On the instances from [VRF15], this should not *a priori* favor a particular value for b_{ij}. For each model a custom white-box value heuristic is used called Greedy in the results, with the following behavior

TSP the value corresponding to the closest successor is selected.

QAP the facility to open is placed at the location minimizing the weighted flow with already placed facilities.

JobShop when choosing a value for a precedence variable b_{ij}, telling if a task i is executed before another task j, the precedence value selected is the one yielding the most slack, describing how much time is still available between the two tasks. This was proposed in [SMV25].

In total, the value selectors analyzed are

Min choosing the smallest value in the domain of the variable.

BIVS the original algorithm as proposed in [FP17], using the author's implementation in Choco-solver. In this implementation, when the domain size of a variable is larger than 100, only the minimum and maximum values of the domain are considered by BIVS.

BIVS+RF indicates BIVS but with the restricted fixpoint, presented in section 3.2.1.

RLA depicts the Reverse Look-Ahead.

RLA+RF depicts the Reverse Look-Ahead using the restricted fixpoint.

Regarding the RF implementation, it uses the non-intrusive method presented in Section 3.2.1.1.

One criterion used to compare the value selection heuristics is the primal gap introduced in [Ber13].

$$\gamma(obj) = \begin{cases} 0 & \text{if } obj = obj^* \\ 1 & \text{if no solution has been found} \\ \frac{|obj - obj^*|}{\max(\{|obj|, |obj^*|\})} & \text{otherwise} \end{cases} \tag{3.10}$$

The primal gap γ gives a value between 0 and 1 measuring the gap between the value of a solution found obj and the best found solution obj^* during the experiment. A value close to 0 means that the solution found is the best one found, while a value of 1 indicates that no solution was found.

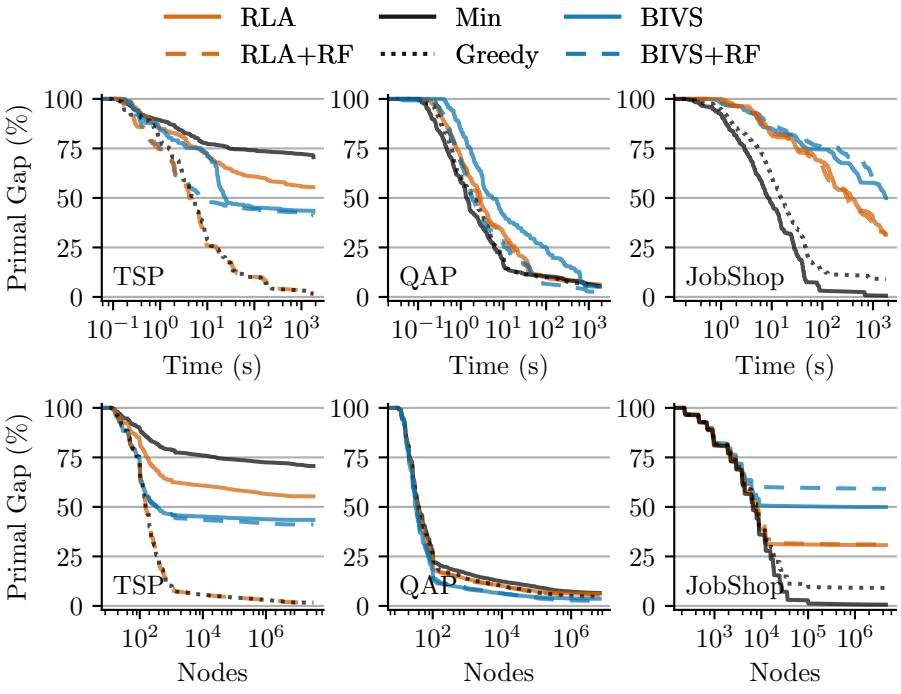


Figure 3.6: Primal gaps in percentage over time (top) and nodes in the search tree (bottom) averaged over all instances.

Heuristic performance varies by problem (Figure 3.6). For TSP, RLA+RF matches the greedy heuristic, while BIVS+RF lags slightly due to considering bounds for domain sizes over 100; without this restriction, BIVS+RF performs comparably to RLA+RF. In QAP, BIVS and its RF variant outperform others, with RF significantly speeding up BIVS. For JobShop, RLA surpasses BIVS but is less effective than MIN due to the cost of fixpoint calls.

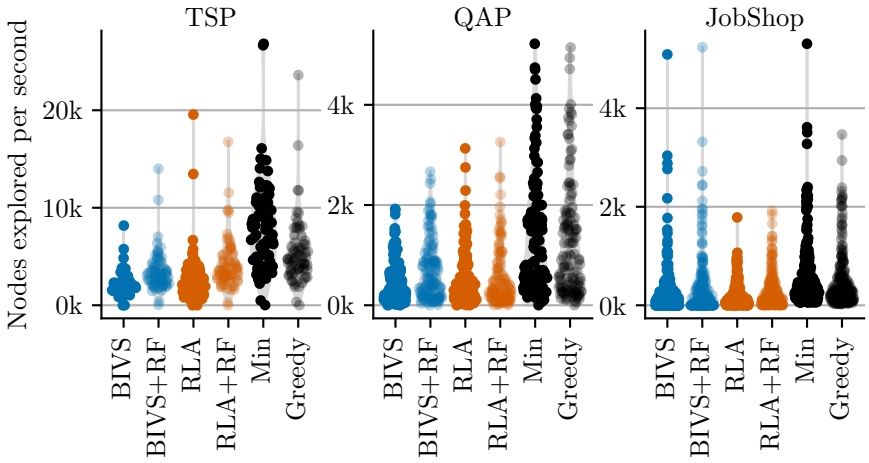


Figure 3.7: Rate of nodes explored over time as a SinaPlot [Sid+18], representing individual observations over the instances as dots, as well as density estimates.

The addition of RF to BIVS results in speedups for the TSP and the QAP, with more nodes in the search tree explored over time, as confirmed by Figure 3.7). The average solution quality across instance sizes is maintainedas shown in Figure 3.8 and supported by Figure 3.6 where BIVS falls behind BIVS+RF. Conversely, it slows performance on JobShop. On this problem, the additional cost of scanning all constraints and potentially deactivating some, in the hope of reducing cost on the two iterations performed by BIVS (as the precedence variables have a domain of size 2) does not offer benefits compared to considering all constraints.

3.3.2 XCSP^3

We consider instances from the XCSP^3 COP 2023 competition [Bou+16b; ALL23]. The instances requiring more than 32GB were discarded, leaving 18 problems and 232 instances.

Table 3.1 displays gaps and number of solutions found per problem, with the average gap over time shown in Figure 3.9. Performance varies greatly across problems, with no single heuristic outperforming others universally. However, adding RF to BIVS reduces the average gap and aids in finding solutions missed otherwise. Similar benefits, though smaller, are observed with RLA. On average, RLA outperforms BIVS, with RF enhancing both methods. Min excels in finding feasible solutions, mostly attributed to its constant time complexity. Such fast selection allows exploring more nodes in the same amount of time, as shown in Table 3.2 presenting the rate of nodes explored

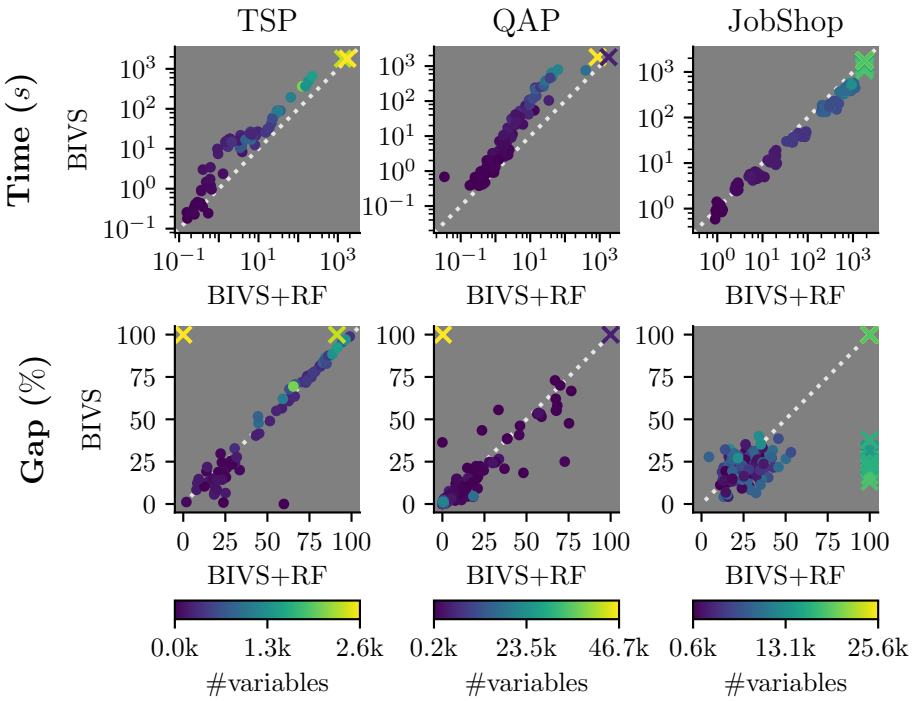


Figure 3.8: Comparison of BIVS and BIVS+RF regarding the time to find the first feasible solution (top, in seconds) and its corresponding gap (bottom, in percentage). Each dot represents an instance across problems: TSP (left), QAP (center), and JobShop (right). Dots on the diagonal indicate equal performance between methods. Dots above the diagonal show that BIVS was slower or found poorer solutions. Crosses denote timeouts by at least one method, resulting in a 100% gap.

per second. Having a faster value selection heuristic gives more opportunities to the variable selection heuristic, in the same amount of time, to set its weights related to encountered failures, allowing for better learning.

In Figure 3.9 two Virtual Best Solver entries are presented: "VBS (old)" computed from previous selectors available in Choco (BIVS, Min, middle, Max and random domain selection) and "VBS (new)" adding RLA and RF-based methods. The 3.87% decrease in the final gap demonstrates that RLA and RF explore the search space differently than traditional heuristics, enhancing portfolio efficiency. The figure also compares BIVS(+RF) with BIVS*(+RF), where the latter considers all domain values — not just the bounds when the domain size exceeds 100 — showing improved performance with RF, suggesting the removal of the domain size consideration. Both BIVS+RF and BIVS*+RF outperform their non-RF versions, indicating their superior efficiency. Notably, the average gap by BIVS+RF at 100.0s is matched by BIVS

Prob. (#inst.)	Min	BIVS	BIVS+RF	RLA	RLA+RF
AAL (20)	95.00 (1)	**90.00 (2)**	100.00 (0)	95.00 (1)	95.00 (1)
CC (20)	61.29 (9)	57.17 (9)	65.48 (8)	62.67 (8)	**54.48 (11)**
GBACP (20)	78.86 **(11)**	**61.43** (8)	79.69 (10)	69.54 (9)	85.51 (9)
GMKP (15)	49.74 **(15)**	**6.78** (14)	6.81 (14)	26.57 (12)	20.26 (13)
HCPizza (10)	**33.56** (10)	34.91 (10)	34.43 (10)	34.67 (10)	34.62 (10)
Hsp (18)	**0.00 (18)**	5.56 (17)	5.56 (17)	5.56 (17)	**0.00 (18)**
KM (15)	50.84 (8)	57.32 (7)	**43.96 (9)**	56.01 (7)	50.84 (8)
KE (14)	44.63 **(14)**	85.71 (3)	51.11 (10)	**43.68** (13)	52.79 (10)
LSS (9)	66.76 **(6)**	66.83 **(6)**	66.83 **(6)**	45.56 (5)	**34.44 (6)**
PSP1 (8)	100.00 (0)	100.00 (0)	**87.50 (1)**	100.00 (0)	100.00 (0)
PSP2 (8)	**87.50** (1)	**87.50** (1)	87.62 (1)	**87.50** (1)	87.66 (1)
PP (7)	57.14 (3)	57.14 (3)	57.14 (3)	57.14 (3)	57.14 (3)
RIP (12)	5.06 (12)	6.31 (12)	4.59 (12)	**3.24** (12)	4.78 (12)
RM (9)	100.00 (0)	100.00 (0)	100.00 (0)	100.00 (0)	100.00 (0)
SREFLP (15)	7.27 (15)	**3.08** (15)	7.44 (15)	8.78 (15)	7.72 (15)
Sonet (16)	**1.40** (16)	2.28 (16)	3.42 (16)	2.42 (16)	3.15 (16)
TSPTW1 (8)	87.81 **(1)**	100.00 (0)	87.84 **(1)**	87.81 **(1)**	**87.50 (1)**
TSPTW1 (8)	**75.19 (2)**	87.50 (1)	**75.19 (2)**	75.42 **(2)**	75.24 **(2)**
All (232)	52.02 **(142)**	51.04 (124)	50.23 (135)	49.85 (132)	**49.52** (136)

Table 3.1: Performance between the methods for each problem. Each column shows the average primal gap over all instances, in percentage, and the instances where at least one feasible solution was found, in parentheses. Best results are highlighted in bold if at least one heuristic was outperformed.

only at 938.32s, demonstrating its significant speed advantage.

3.4 Extending to Satisfaction Problems

The previous section presented BIVS+RF and RLA(+RF), which lead to the same decisions as nearest neighbor value heuristics on VRPs. They are also applicable directly within CP solvers for other kinds of problems than VRPs.

A natural question that arises is whether those techniques can be applied on CSPs instead of COPs. While this slightly falls outside the scope of this chapter (the goal would now be to design general-purpose value heuristics for CSPs), this section still presents some considerations if one attempted to tackle CSPs with those techniques.

Minimizing Constraint Violation A naive way to render the techniques compatible with CSPs is through constraints reifications and penalties. One can transform a CSP into a COP, by reifying some constraints. Every boolean variable introduced during the reification can be mapped to a penalty, set to 1 if the constraint is violated, and to 0 if it holds. The objective of the

Problem (#instances)	Min	BIVS	BIVS+RF	RLA	RLA+RF
AAL (20)	**1962.41**	579.70	1634.72	1700.13	1543.85
CC (20)	**8602.77**	1795.75	5395.77	3619.87	7815.56
GBACP (20)	9574.80	4506.79	8721.60	3660.39	**9949.38**
GMKP (15)	**1738.40**	1105.62	1299.89	987.88	497.97
HCPizza (10)	**16842.26**	7639.15	7455.64	8807.99	8413.84
Hsp (18)	312.64	172.56	330.47	108.16	**384.77**
KM (15)	**372.96**	20.08	268.26	135.81	226.35
KE (14)	2418.85	794.51	2323.79	**2674.80**	1520.95
LSS (9)	976.83	632.34	**1107.52**	304.01	989.66
PSP1 (8)	465.32	317.26	**588.58**	124.02	142.69
PSP2 (8)	**2361.15**	935.21	1178.51	701.76	769.81
PP (7)	1036.62	681.08	968.96	1020.07	**1084.41**
RIP (12)	**4179.38**	893.40	2506.77	2742.59	3192.95
RM (9)	300.94	132.32	207.19	137.24	**302.84**
SREFLP (15)	**719.83**	328.75	559.33	444.72	260.16
Sonet (16)	**13842.64**	7481.46	9189.45	9037.57	8501.42
TSPTW1 (8)	**2033.02**	383.63	1393.55	649.65	732.01
TSPTW2 (8)	**2008.05**	660.96	1535.98	1005.32	1045.61
All (232)	**4303.93**	1769.54	2988.33	2323.61	3139.58

Table 3.2: Average rate of nodes explored over time (per second) between the methods for each problem.

newly formed COP is then the minimization of the sum of those penalty variables. During the search, the value chosen for a variable x with RLA(+RF) and BIVS(+RF) would be the one that violates the penalties less.

However, this procedure is not efficient for several reasons. First, a choice must be made regarding the constraints to reify, as reifying all of them might remove the majority of the filtering strength, as highlighted in [Bjö+20]. Second, for RF, the introduced objective is likely connected to a large portion of the constraints. Therefore, the shortest paths between a variable and the objective are likely to be of very small length, and may involve only the constraints directly linked to the variable. This is for instance the case when all constraints are reified, and the shortest paths are of length 2 (one constraint for going to the introduced penalty, and one for the sum of penalties). Last, in this setting, we are in reality interested in two cases: generating a feasible solution (*i.e.* with a penalty sum of 0) or proving that none exists (in which case reifying the constraints would hinder the propagation strength). In other words, there are only two valuable cases in this setting: a zero penalty sum or a non-zero penalty sum, and such reification might be too costly to encode this setting.

Reifing constraints and reducing their penalties shares similarities to the

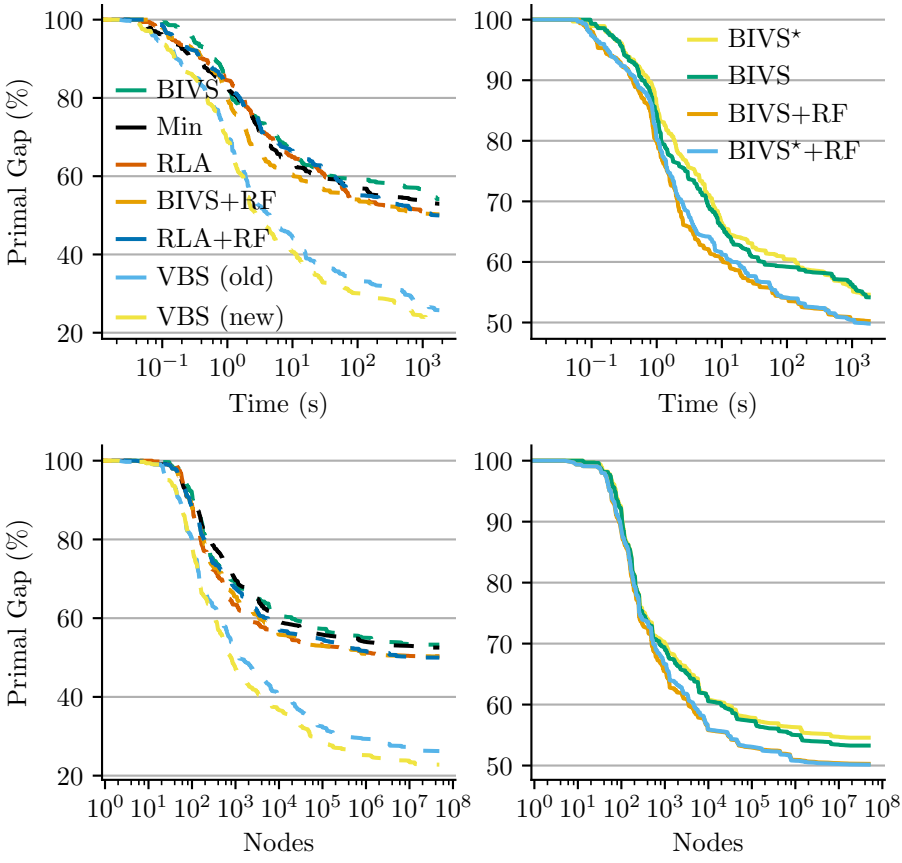


Figure 3.9: Average primal gap over time (top) and over number of nodes (bottom) on the XCSP^3 instances, in percentage. The right part shows only 4 selectors compared to the left part, and their y-scale differs. Legends are sorted by the final gap observed.

procedure proposed in [Bjö+20]. Their work describes a more involved strategy, where some hard constraints were *softened*. A softened constraint can be regarded as a reified constraint, where the penalty introduced is an integer variable instead of a boolean variable, whose domain gives more information on the constraint violation (see [MRS06] for more detailed explanations). When solving a problem using their approach, the solver considers both the hard and softened constraints, and deactivates the hard constraint whenever a failure occurs. This procedure renders the application of LNS possible as we are working with a COP, and allows benefiting from both branch-and-bound and from the propagation of the hard constraint. The work from [Sch13] also proposes to use LNS to minimize the sum of penalty variables. Part of their approach proposes to target one individual penalty at each LNS iteration, in-

stead of directly minimizing the penalty sum. In either case, the mentioned methods are mainly meant to be used in an LNS setting, therefore giving up on exact methods, and the value heuristic in their constructed COP is still MinDom.

Dedicated Approach to CSP Instead of working with penalties, it might be worthwhile to opt for a strategy specifically dedicated to CSPs for value selection to integrate the RF. Compared to RF on COPs:

- **The objective** consists in maximizing the remaining search space, estimated as the Cartesian product of the domain size of the variables.

- **The shortest paths** are computed between a variable and the "strongest" identified constraint. Strongest might have different interpretations, such as the constraint responsible for the largest number of failures, or the one that provoked the majority of filtering on the domains.

Given the absence of variable corresponding to the objective, RLA cannot be used in this setting. With BIVS+RF, the value chosen for a variable x would be the one that maximizes the remaining search space, estimated after filtering of the strongest constraint and a few other ones lying between it and x and on the constraint network.

There are some similarities between this procedure and Belief Propagation, where values are selected as the ones maximizing the probability to appear in a solution [Pes19]. The key difference is the absence of marginal probability to estimate for the propagators here. While less precise, this method does not suffer from the damping effect present in BP, where probability estimates may fluctuate significantly between iterations. In addition, BP requires computing and maintaining probabilities for each value-variable pair, making it costly on large problems. In comparison, the proposed method does not need to record values for all such pairs. Lastly, it does not require implementing dedicated algorithms per constraint, needed for approximating the distributions in BP.

There is a stronger similarity with impact-based search [Ref04] and activity-based search [MV12a], which also selects values having the smallest domain reduction on other variables, or values having modified the fewest number of variables. One difference is that those searches, when used for value selection, rely on past computation at previous search nodes to select values. Moreover, they need to store the impact of each value, whereas such storage is not needed with this approach.

3.5 Conclusion

Deriving effective, generic value heuristics that balance speed and informativeness remains challenging. BIVS stands out among these approaches, yet its cost limits its applicability in certain scenarios. By incorporating a restricted fixpoint in the look-ahead process and employing a reverse look-ahead strategy, costs are significantly reduced, making previous restrictions on BIVS usage less relevant. On problems such as the TSP, the proposed methods offer performance on par with the white-box value selection that chooses nearest neighbors. The proposed methods do not require any training, are well suited for black-box settings, and substantially improve performance. When utilized alone or within a portfolio approach, these strategies continue to enhance the efficiency of solving COPs.

Some considerations for generalizing those techniques to CSPs were presented in section 3.4. Given that they are slightly outside the initial scope of this chapter, which was to design value heuristics for VRPs, they were not experimentally validated. Future work may be conducted to assess whether those techniques are worth considering.

Sequence Variables 4

This chapter attempts to answer the research questions:

- *How to integrate LNS with insertions inside CP solvers?*
- *How to efficiently deal with optional visits when solving VRPs with CP?*

Both of those questions will be tackled by studying insertion sequence variables.

Insertion sequence variables were first introduced in [TKS20] to tackle complex VRPs. I continued the work from Thomas, Kameugne, and Schaus by changing the domain computed by those variables, and improved the obtained solutions on several VRPs. This led to the publication A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables for Routing Problems". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022

Since then, sequence variables have changed quite a lot. This chapter is heavily based on a journal paper to be published, introducing the newest version of those variables, in A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables: A Constraint Programming Computational Domain for Routing and Sequencing". Manuscript in preparation. 2025

Sequence variables are first presented from a practical point of view on a challenging VRP in section 4.1. Afterward, the domain of a sequence variable is introduced in Section 4.2, followed by sequence-specific constraints in Section 4.3 and consideration on the search in Section 4.4. Once all those concepts are introduced, differences with previous versions are highlighted in section 4.5. Section 4.6 evaluates those variables on several optimization problems. Lastly, Section 4.7 discusses limitations and future work.

4.1 Introduction

Existing CP approaches struggle to handle optional visits and do not support insertion-based search strategies, which are crucial for quickly obtaining high-quality solutions. To address this, we introduce a sequence-based computational domain that enables both optional visits and insertion-based searches. Let us first demonstrate its usage for modeling the Dial-A-Ride Problem (DARP) and searching for solutions with an insertion-based search.

The DARP is defined over a graph $G = (V, E)$, the nodes being transportation points. A distance matrix $\boldsymbol{d}$ indicates the distance between nodes, and the objective is to minimize the total routing cost. The set of K available vehicles departs from a common depot, fulfill a subset of the R transportation requests in the problem, and return to the common depot. Each request $\boldsymbol{r}_i \in R$ is composed of a pickup $\boldsymbol{r}_i^+ \in V$ and its corresponding drop off location $\boldsymbol{r}_i^- \in V$. Several constraints restrict the types of travel that can be performed. Each node $v_i \in V$ to be visited (*i.e.* the depot, each pickup and drop location) has a service duration $\boldsymbol{s}_i$ for visiting it, and a given time window: the visit must happen within $[a_i, b_i]$. Additionally, the ride time a customer $\boldsymbol{r}_i \in R$ spends in its vehicle is limited, ensuring the time from pickup $\boldsymbol{r}_i^+$ to drop-off $\boldsymbol{r}_i^-$ does not exceed a predefined time limit $\boldsymbol{t}_i > 0$. In addition to restricting the ride time, the route duration is also constrained: the total time between departing and returning to the depot cannot exceed a set limit $\boldsymbol{t}^d > 0$. Finally, processing a transportation request $\boldsymbol{r}_i \in R$ consumes a load of $\boldsymbol{q}_i > 0$ in a vehicle, whose limited capacity c cannot be exceeded.

The problem is depicted in Figure 4.1, which only presents the location of nodes. Time windows are not represented to improve readability.

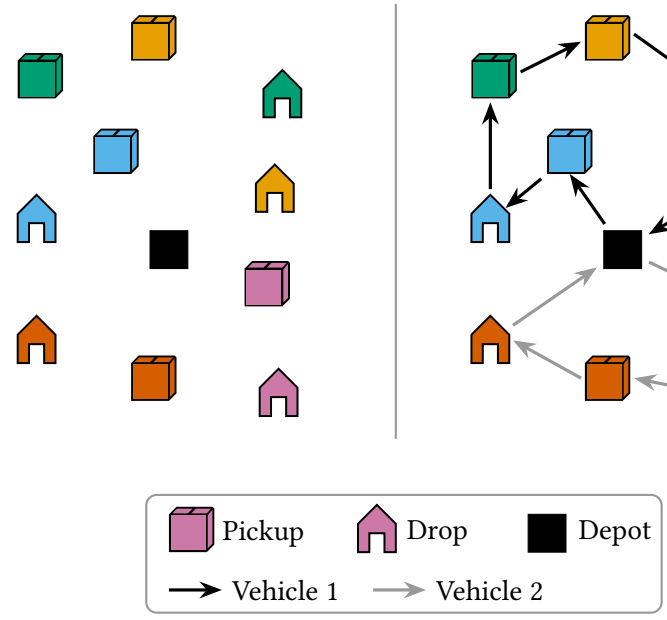


Figure 4.1: Example of a DARP instance (left) and a possible solution (right).

The full model and the search are given next, illustrating the *CP = model + search* paradigm. The precise semantics of each constraint does not need to be understood in detail at this stage.

The Model

$$\min \sum_{k \in K} \boldsymbol{Dist}_k \tag{4.1}$$

subject to:

$$\text{Distance}(\boldsymbol{Route}_k, \boldsymbol{d}, \boldsymbol{Dist}_k) \qquad \forall k \in K \tag{4.2}$$

$$\text{TransitionTimes}(\boldsymbol{Route}_k, (\boldsymbol{Time}), \boldsymbol{s}, \boldsymbol{d}) \qquad \forall k \in K \tag{4.3}$$

$$\text{Cumulative}(\boldsymbol{Route}_k, \boldsymbol{r}^+, \boldsymbol{r}^-, \boldsymbol{q}, c) \qquad \forall k \in K \tag{4.4}$$

$$\sum_{k \in K} \mathcal{R}_v(\boldsymbol{Route}_k) = 1 \qquad \forall v \in V \tag{4.5}$$

$$\boldsymbol{Time}_{\boldsymbol{r}_i^-} - \boldsymbol{Time}_{\boldsymbol{r}_i^+} - \boldsymbol{s}_{\boldsymbol{r}_i^+} \leq \boldsymbol{t}_i \qquad \forall r_i \in R \tag{4.6}$$

$$\boldsymbol{Time}_{\omega_k} - \boldsymbol{Time}_{\alpha_k} \leq \boldsymbol{t}^d \qquad \forall k \in K \tag{4.7}$$

This model relies on an insertion-based sequence variable $Route_k$ for each vehicle $k \in K$. It represents the sequence of nodes visited by vehicle k, starting at the depot and ending at the depot. The objective consists in minimizing the sum of traveled distance (4.1). The travel length of each vehicle $k \in K$ is linked in (4.2) to an integer variable $\boldsymbol{Dist}_k$, while constraint (4.3) enforces the visit of each node $v \in V$ during its time window $(\boldsymbol{Time})_v$, an integer variable. The capacity available within a vehicle is constrained through (4.4). Constraint (4.5) ensures that every node $v \in V$ is visited exactly once. Lastly, (4.6) and (4.7) enforce the maximum ride time and the maximum duration of the route, respectively.

The Search In vehicle routing, two main search strategies are commonly used. The nearest neighbor heuristic sequentially appends the closest unvisited node to the current sequence. The insertion-based heuristic selects a node and inserts it at the position that minimizes the cost increase. As shown in [RSL77], insertion-based construction heuristics are particularly effective in avoiding the long edge effect, where a very long final connection is required to close a tour, a common issue with nearest neighbor strategies. An insertion-based branching procedure for the DARP is presented in Algorithm 3.

The branching strategy prioritizes the unvisited request $\boldsymbol{r_i}$ with the fewest remaining feasible insertions (line 3), using the method nInsert(v) returning the number of insertions for node v. This is in line with the *first-fail* principle, which aims to create the shallowest possible search tree. Given this request, all insertion points I^+ for its pickup $\boldsymbol{r}_i^+$ in a given sequence variable $\boldsymbol{Route}_k$ are retrieved (line 6). Each insertion point corresponds to a node predecessor after which the pickup $\boldsymbol{r}_i^+$ may be inserted. Suitable insertions I^- for its delivery $\boldsymbol{r}_i^-$ in the current path are also retrieved, including the pickup $\boldsymbol{r}_i^+$ as

Algorithm 3: Creation of the branching points for the DARP.

```
1  if ⋀_{k∈K} Route_k.isFixed() then
2  |  return solution
3  r_i ← argmin_{r_j ∈ R | r_j not inserted} Σ_{k∈K} Route_k.nInsert(r_j^+) · Route_k.nInsert(r_j^-)
4  branches ← {}
5  for k ∈ K do
6  |  I^+ ← Route_k.getInsert(r_i^+)
7  |  for p^+ ∈ I^+ do
8  |  |  I^- ← Route_k.getInsertAfter(r_i^-, p^+) ∪ {r_i^+}
9  |  |  for p^- ∈ I^- do
10 |  |  |  branches ← branches ∪
   |  |  |    {(Route_k.insert(p^+, r_i^+) ∧ Route_k.insert(p^-, r_i^-))}
11 sort branches by increasing order of heuristic cost
12 return branches
```

a predecessor candidate (line 8). All valid combinations of predecessors are considered as an alternative branching decision to consider (line 10). Since DFS is used in CP, the most promising insertions should be explored first on leftmost branches. This can for instance be achieved by sorting all candidate insertions based on their impact on tour length, prioritizing those that minimize the increase in distance (line 11). A solution is reached when no further insertions are possible in any vehicle: all paths are fixed (line 2).

Example 4.1.1. The example refers to Figure 4.2 with only one vehicle. A route serves both visits of r_1. The request r_2 is selected, resulting in six possible sequences for inserting its two visits, each corresponding to a possible branching decision in the search tree. These insertion options are sorted by their cost, corresponding to the increase in distance.

A DFS using the branching of Algorithm 3 can be used to find an initial solution by stopping at the first feasible solution encountered. A limited discrepancy DFS search, keeping only the leftmost few branches, as was done in [JV11] can also be used to quickly discover good solutions. The search can also be used as part of an LNS strategy [Sha98], whose main iteration is depicted in Algorithm 4. A set $\mathcal{R}$ of requests to relax from a previous solution S is first selected (line 1). The paths represented in the previous solution are then enforced, except for nodes $\mathcal{V}^{\mathcal{R}}$ belonging to relaxed requests, which are omitted (line 7). A search is finally performed (line 8) to insert those remaining relaxed requests, leading to a new solution $\hat{S}$. This process is repeated until a given stopping criterion is met.

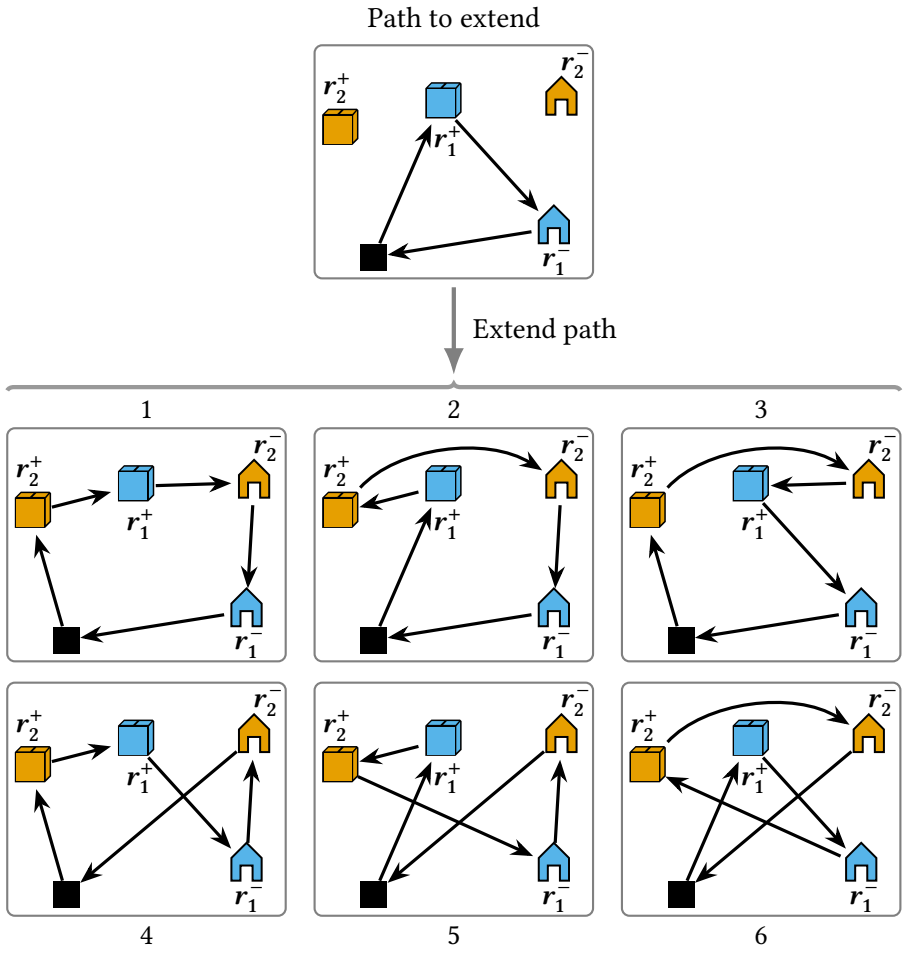


Figure 4.2: Paths generated by Algorithm 3 given an initial one (top). Numbers indicate which are considered first.

4.1.1 Limitations of Existing CP Approaches

The model and search strategy introduced for the DARP in the previous section rely on insertion-based sequence variables. However, the most common approach for modeling VRPs in CP is based on the successor model, which does not support insertion-based search strategies. This kind of model has two main limitations. First, representing the optional nature of visits with the successor model is not straightforward[1]. Second, at the search level, the first

[1]Even though the subcircuit version allows a node to be excluded from the circuit by assigning it a self-loop ($succ_i = i$), it complicates the model semantics, as constraints need to be aware that self-loops are permitted and designate an unvisited node

Algorithm 4: An LNS iteration for DARP

Input: $S = [S_1, \ldots, S_K]$: a sequence of visits for each vehicle

```
1 R ← relaxedRequests(S)
2 V^R ← ⋃_{r_i ∈ R} {r_i^+, r_i^-}
3 for k ∈ K do
4     Route_k ← empty sequence variable // empty path
      // iterate in order over previous path
5     for v ∈ S_k do
6         if v ∉ V^R then
              // append the visit in same order as in S
7             Route_k.insertAtEnd(v)
8 Ŝ ← Find best solution by optimizing Route
```

solution constructed along the leftmost branch of the search tree typically relies on a *nearest neighbor* heuristic. These nearest neighbor heuristics tend to add small edges near the root of the search tree, but as decisions progress, they tend to add very long edges at the end, making it difficult to quickly find good solutions.

An advanced CP alternative to the successor model, which enables dealing with optional visits more naturally, is to use the *head-tail sequence variables* implemented in IBM CP Optimizer [Lab+18a; Lab+09a]. These variables handle optionality by design since not every node needs to be added in the sequence. However, similarly to the successor model, a heuristic on those variables would also rely on a nearest neighbor strategy for extending the head or the tail of the sequence.

Since both the successor and the head-tail sequence models do not easily support insertion heuristics, a new type of variable, the insertion-based sequence variable, was recently introduced in [TKS20; DSV22; Tho23] to address these limitations. The insertion-based sequence variables have a domain composed of the possible insertions for each node within a partial sequence, being a cycle over a subset of nodes. These insertion-based sequences thus consume more space than the head-tail sequence variables. A summary of the main properties of the modeling variables for VRPs in CP is given in Table 4.1.

In the rest of the chapter, insertion-based sequence variables are simply denoted *sequence variables*. This work significantly extends the previous work on sequence variables from [TKS20; DSV22; Tho23] by:

- **Formalizing the computational model for sequence variables**, providing a robust theoretical foundation.
- **Introducing consistency levels**, including the novel concept of *insert consistency*.

Feature	Type of variable		
	Successor	Head-tail sequence	Insertion-based sequence
Nearest neighbor heuristics	✓	✓	✓
Insertion based heuristics			✓
Optional visits		✓	✓
Memory complexity on VRPs with n nodes and k vehicles	$O(n^2)$	$O(k \cdot n)$	$O(k \cdot n^2)$

Table 4.1: CP Variables characteristics.

- **Proposing an implementation** along with the underlying data structures to integrate sequence variables into existing trail-based CP solvers.
- **Developing global constraints** for modeling VRPs with sequence variables.
- **Demonstrating the effectiveness of sequence variables** on several problems.

4.2 Sequence Domain

We first introduce some notations before defining the domain of sequence variables.

Notations A sequence $\overrightarrow{S}$ is defined as an ordered set of nodes belonging to a graph, without repetition. Let $\overrightarrow{S}$ be a sequence with the form $\overrightarrow{S} = \overrightarrow{S}_1 \cdot v_1 \cdot v_3 \cdot \overrightarrow{S}_2$ ($\overrightarrow{S}_1$ and $\overrightarrow{S}_2$ being sequences, possibly empty). An insertion operation defined by the triplet of nodes (v_1, v_2, v_3) produces a new sequence $\overrightarrow{S}' = \overrightarrow{S}_1 \cdot v_1 \cdot v_2 \cdot v_3 \cdot \overrightarrow{S}_2$. We denote $v_i \xrightarrow{\overrightarrow{S}} v_j$ to indicate that v_i directly precedes v_j in the sequence $\overrightarrow{S}$ and $v_i \overset{\overrightarrow{S}}{\prec} v_j$ when v_i precedes (not necessarily directly) v_j in $\overrightarrow{S}$. Those relations are simply written $v_i \rightarrow v_j$ and $v_i \prec v_j$ when clear from the context. If the nodes can be the same, the relation is written $v_i \preceq v_j$. Given a sequence $\overrightarrow{S} = v_1 \dots v_i \dots v_n$, $\text{prefix}(\overrightarrow{S}, v_i) = v_1 \dots v_i$ and $\text{suffix}(\overrightarrow{S}, v_i) = v_i \dots v_n$. Lastly, given a node set V, a start node $\alpha \in V$ and an end node $\omega \in V$, $\mathcal{P}(V)$ denotes all sequences $\overrightarrow{S}$ over a subset of nodes V, starting at node $\alpha \in V$ and ending at node $\omega \in V$.

Domain A sequence domain denoted $\mathcal{D} \subseteq \mathcal{P}(V)$ contains a set of sequences over a subset of the nodes V without repetition, starting at node $\alpha \in V$ and ending at node $\omega \in V$. Four elementary domain updates are exposed, to restrict any sequence represented in the domain:

1. **Require** a node to be visited, without explicitly stating the position of the node in the sequence.

2. **Exclude** a node from any sequence in the domain.

3. **Enforce a subsequence** to be included within any sequence from the domain. This is useful when an initial path has been identified (the subsequence), and every sequence from the domain must be constructed by extending it through insertions.

4. Forbid the occurrence of a node v_2 between two other nodes v_1 and v_3 within the sequence, regardless of any additional nodes that may be present between them. This operation is useful to represent that the visit of v_3 may be performed after v_1 provided that v_2 is omitted, for instance if passing through v_2 would exceed an allowed distance. This corresponds to **forbid subsequences** $v_1 \cdot v_2 \cdot v_3$ of length 3, with potential nodes between them.

To define the sequence domain $\mathcal{D}$, we can define one subdomain per operation, tailored to represent the remaining sequences after applying it. Those four subdomains are as follows.

1. $\mathcal{D}^{\textcircled{R}}(R)$ is specified by a set of required nodes $R \subseteq V$. It denotes all sequences including all nodes in R. More formally,

$$\mathcal{D}^{\textcircled{R}}(R) = \left\{ \overrightarrow{S} \mid \forall v \in R : v \in \overrightarrow{S} \right\} \tag{4.8}$$

2. $\mathcal{D}^{\textcircled{X}}(X)$ is specified by a set of excluded nodes $X \subseteq V$. It denotes all sequences where no node in X is included. More formally,

$$\mathcal{D}^{\textcircled{X}}(X) = \left\{ \overrightarrow{S} \mid \forall v \in X : v \notin \overrightarrow{S} \right\} \tag{4.9}$$

3. $\mathcal{D}^{\textcircled{S}}(\overrightarrow{s})$ is specified by a partial sequence of nodes $\overrightarrow{s}$. It denotes all sequences $\overrightarrow{S}$ such that $\overrightarrow{s}$ is a subsequence of $\overrightarrow{S}$. More formally,

$$\mathcal{D}^{\textcircled{S}}(\overrightarrow{s}) = \left\{ \overrightarrow{S} \mid \forall v_1, v_2 \in \overrightarrow{s} : v_1 \overset{\overrightarrow{s}}{\prec} v_2 \implies v_1 \overset{\overrightarrow{S}}{\prec} v_2 \right\} \tag{4.10}$$

4. $\mathcal{D}^{\text{Ⓕ}}(F)$ is specified by a set of subsequences $F \subseteq V \times V \times V$ of length 3, which is the minimum length to represent the occurrence of a node between two other nodes. $\mathcal{D}^{\text{Ⓕ}}(F)$ denotes all sequences where no subsequence in F appears. More formally,

$$\mathcal{D}^{\text{Ⓕ}}(F) = \left\{ \overrightarrow{S} \mid \forall (v_i \cdot v_j \cdot v_k) \in F : \neg (v_i \overset{\overrightarrow{S}}{\prec} v_j \wedge v_j \overset{\overrightarrow{S}}{\prec} v_k) \right\} \tag{4.11}$$

Those four subdomains define a sequence domain as follows.

Definition 4.2.1. A sequence domain is defined as

$$\mathcal{D} = \langle R, X, \overrightarrow{s}, F \rangle = \mathcal{D}^{\text{Ⓡ}}(R) \cap \mathcal{D}^{\text{Ⓧ}}(X) \cap \mathcal{D}^{\text{Ⓢ}}(\overrightarrow{s}) \cap \mathcal{D}^{\text{Ⓕ}}(F) \tag{4.12}$$

Definition 4.2.2. $\mathcal{D}$ is said to be fixed when $|\mathcal{D}| = 1$.

Example 4.2.1. Let

- $V = \{\alpha, v_1, v_2, v_3, \omega\}$
- $R = \{\alpha, \omega, v_2\}$
- $X = \{v_3\}$
- $\overrightarrow{s} = \alpha \cdot v_1 \cdot \omega$
- $F = \{(\alpha \cdot v_2 \cdot v_1), (v_3 \cdot v_1 \cdot v_2)\}$

The sequence domain is $\mathcal{D} = \mathcal{D}^{\text{Ⓡ}}(R) \cap \mathcal{D}^{\text{Ⓧ}}(X) \cap \mathcal{D}^{\text{Ⓢ}}(\overrightarrow{s}) \cap \mathcal{D}^{\text{Ⓕ}}(F) = \{(\alpha \cdot v_1 \cdot v_2 \cdot \omega)\}$. The compositions of the subdomains are represented in Table 4.2. Given that $|\mathcal{D}| = 1$, the domain is fixed.

Domain updates over $\mathcal{D}$ are defined by growing the sets R, X and F, adding more elements to them, or by inserting a node within the partial sequence $\overrightarrow{s}$. One can also enforce more complex constraints through several subdomain modifications. For instance, forcing a node v_2 to be visited between nodes v_1, v_3 can be achieved by adding $(\alpha \cdot v_2 \cdot v_1), (v_3 \cdot v_2 \cdot \omega)$ to the forbidden subsequences F given that any sequence $\overrightarrow{S} \in \mathcal{D}$ begins at α and ends at ω. Moreover, if the node v_2 must be included in the sequences of the domain, v_2 can be added into the required nodes R.

A sequence domain is used by a sequence variable $\overrightarrow{S}$, which represents an unknown sequence. Such a variable is particularly convenient for modeling a route from an origin node α to a destination node ω in a VRP, where the nodes visited and their ordering represents the path performed by a vehicle.

The main challenge for representing a domain is memory. The set of forbidden subsequences $F \subseteq V \times V \times V$ requires a cubic amount of space in

$\mathcal{P}(V)$	$\mathcal{D}^{\textcircled{r}}(R)$	$\mathcal{D}^{\textcircled{x}}(X)$	$\mathcal{D}^{\textcircled{s}}(\overrightarrow{s})$	$\mathcal{D}^{\textcircled{f}}(F)$
$\alpha \cdot \omega$		✓		✓
$\alpha \cdot v_1 \cdot \omega$		✓	✓	✓
$\alpha \cdot v_2 \cdot \omega$	✓	✓		✓
$\alpha \cdot v_3 \cdot \omega$				✓
$\boldsymbol{\alpha \cdot v_1 \cdot v_2 \cdot \omega}$	✓	✓	✓	✓
$\alpha \cdot v_1 \cdot v_3 \cdot \omega$			✓	✓
$\alpha \cdot v_2 \cdot v_1 \cdot \omega$	✓	✓	✓	
$\alpha \cdot v_2 \cdot v_3 \cdot \omega$	✓			✓
$\alpha \cdot v_3 \cdot v_1 \cdot \omega$			✓	✓
$\alpha \cdot v_3 \cdot v_2 \cdot \omega$	✓			✓
$\alpha \cdot v_1 \cdot v_2 \cdot v_3 \cdot \omega$	✓		✓	✓
$\alpha \cdot v_1 \cdot v_3 \cdot v_2 \cdot \omega$	✓		✓	✓
$\alpha \cdot v_2 \cdot v_1 \cdot v_3 \cdot \omega$	✓		✓	
$\alpha \cdot v_2 \cdot v_3 \cdot v_1 \cdot \omega$	✓		✓	
$\alpha \cdot v_3 \cdot v_1 \cdot v_2 \cdot \omega$	✓		✓	
$\alpha \cdot v_3 \cdot v_2 \cdot v_1 \cdot \omega$	✓		✓	

Table 4.2: Subdomains composition in Example 4.2.1. A check mark in one of the last 4 columns indicates that the corresponding sequence in the first column is included within the subdomain.

the size of the set V, and is thus impractical to encode in problems with a large set of nodes. Moreover, computing the intersection of the subdomains to yield the domain $\mathcal{D}$ might be time-consuming. We instead introduce next a compact encoding of this domain using a directed graph, restraining the forbidden subsequences that can be encoded in F while consuming $O(|V|^2)$ memory.

4.2.1 Encoding Forbidden Subsequences in a Graph

It is worth noting that a partial sequence $\overrightarrow{s}$ and set of forbidden subsequences F define a subset of a sequence domain: $\mathcal{D} \subseteq \mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(F)$. Given that we are interested in the intersection of the subdomains, we will restrict how the partial sequence $\overrightarrow{s}$ can grow given the set F, and how forbidden subsequences can be added to F given the partial sequence $\overrightarrow{s}$. For instance, it would be useless to add a forbidden subsequence $(v_1 \cdot v_2 \cdot v_3)$ to F if $\overrightarrow{s} = \alpha \cdot v_3 \cdot v_2 \cdot v_1 \cdot \omega$, as $\mathcal{D}^{\textcircled{s}}(\overrightarrow{s})$ already prevents such subsequence.

4.2.1.1 Forbidden Subsequences Restriction

To allow a compact encoding of the forbidden subsequences F, we restrict them as follows:

$$F \subseteq \{(v_1 \cdot v_2 \cdot v_3) \mid v_1 \overset{\overrightarrow{s}}{\prec} v_3\} \tag{4.13}$$

Where $\overrightarrow{s}$ is the partial sequence from the domain $\langle R, X, \overrightarrow{s}, F\rangle$. In other words, condition (4.13) restricts F so that it only contains subsequences of length 3, where the extremities of each subsequence belong to the partial sequence $\overrightarrow{s}$ and are consecutive. To emphasize this restriction, F is written $\hat{F}$ in the following, to clearly show that condition (4.13) holds. Forbidden subsequences from F that do not match condition (4.13) need to be delayed before being added to $\hat{F}$ until their endpoints are consecutive in the partial sequence. In case $v_1 \prec v_2 \prec v_3 \in \hat{F}$, the partial sequence $\overrightarrow{s}$ contains a forbidden subsequence. This corresponds to a domain wipe-out ($\mathcal{D} = \emptyset$) and will be discussed in section 4.2.1.6.

Given $\hat{F}$ and $\overrightarrow{s}$, the set of subsequences implicitly forbidden is:

$$\mathcal{F}^I(\hat{F}, \overrightarrow{s}) = \{(v_i \cdot v_2 \cdot v_j) \mid (v_1 \cdot v_2 \cdot v_3) \in \hat{F} \wedge v_1 \preceq v_i \overset{\overrightarrow{s}}{\longrightarrow} v_j \preceq v_3\} \tag{4.14}$$

In (4.14), $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ defines all forbidden subsequences, based on $\hat{F}$, where the extremities of each subsequence are directly preceding each other in the partial sequence $\overrightarrow{s}$. Given that extremities must only be preceding each other, not necessarily directly, in subsequences from $\hat{F}$ ($v_i \prec v_j$), we do not necessarily have $\hat{F} \subseteq \mathcal{F}^I(\hat{F}, \overrightarrow{s})$.

Example 4.2.2. Let $\overrightarrow{s} = \alpha \cdot v_1 \cdot v_2 \cdot \omega$ and $\hat{F} = \{(v_1 \cdot v_3 \cdot v_2), (\alpha \cdot v_4 \cdot v_2)\}$. We get $\mathcal{F}^I(\hat{F}, \overrightarrow{s}) = \{(v_1 \cdot v_3 \cdot v_2), (\alpha \cdot v_4 \cdot v_1), (v_1 \cdot v_4 \cdot v_2)\}$: compared to $\hat{F}$, the subsequence $(\alpha \cdot v_4 \cdot v_2)$ has been decomposed into two subsequences: $(\alpha \cdot v_4 \cdot v_1)$ and $(v_1 \cdot v_4 \cdot v_2)$

By allowing the restriction (4.13) and exploiting the fact that the partial sequence $\overrightarrow{s}$ can only grow, the set $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ can be represented using a directed graph $G(V, E)$. In this graph representation, *detour edges* limit the allowed insertions in $\overrightarrow{s}$.

Definition 4.2.3. A *pair of detour edges* is a pair of edges $(v_1, v_2), (v_2, v_3)$ that can be used to insert a node v_2 outside of the partial sequence $\overrightarrow{s}$ between two nodes v_1 and v_3 with v_1 directly preceding v_3 in the partial sequence $\overrightarrow{s}$:

$$(v_2 \in V \setminus \overrightarrow{s}) \wedge v_1 \overset{\overrightarrow{s}}{\longrightarrow} v_3 \wedge (v_1, v_2) \in E \wedge (v_2, v_3) \in E \tag{4.15}$$

We equivalently say that a pair of detour edges $(v_1, v_2), (v_2, v_3)$ defines an *insertion triplet* (v_1, v_2, v_3).

Definition 4.2.4. A node $v_2 \in V$ is *insertable* if a pair of detour edges passes through it:

$$(v_2 \in V \setminus \overrightarrow{s}) \wedge \left(\exists v_1 \xrightarrow{\overrightarrow{s}} v_3 : (v_1, v_2) \in E \wedge (v_2, v_3) \in E\right) \tag{4.16}$$

The forbidden subsequences from $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ use those detour edges to restrict the insertions that may be performed over $\overrightarrow{s}$:

$$\forall (v_2 \in V \setminus \overrightarrow{s}) : (v_1 \cdot v_2 \cdot v_3) \in \mathcal{F}^I(\hat{F}, \overrightarrow{s}) \implies (v_1, v_2) \notin E \wedge (v_2, v_3) \notin E \tag{4.17}$$

Where a pair of detour edges $(v_1, v_2), (v_2, v_3)$ is deleted whenever the subsequence $(v_1 \cdot v_2 \cdot v_3)$ is forbidden by $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$. In other words, with rules (4.15), (4.17), the partial sequence $\overrightarrow{s}$ may only grow through an insertion (v_1, v_2, v_3) if the related subsequence is not forbidden: $(v_1 \cdot v_2 \cdot v_3) \notin \mathcal{F}^I(\hat{F}, \overrightarrow{s})$.

Condition (4.17) does not restrict all nodes and edges in the graph $G(V, E)$, given that it holds for nodes $v_2 \in V \setminus \overrightarrow{s}$. We next define the *insert consistency* property that the graph $G(V, E)$ and the partial sequence $\overrightarrow{s}$ must maintain to fully encode $\hat{F}$.

4.2.1.2 Insert Consistency

Definition 4.2.5. A pair $\langle \overrightarrow{s}, G(V, E) \rangle$ comprising $\overrightarrow{s}$ and graph $G(V, E)$ is *insert consistent* iff

1. The partial sequence $\overrightarrow{s}$ is a simple path of a subset of nodes in V, starting at $\alpha \in V$ and ending at $\omega \in V$.

2. The partial sequence $\overrightarrow{s}$ uses edges from E. Formally:

$$\forall v_1, v_2 \in \overrightarrow{s} : (v_1 \rightarrow v_2) \iff (v_1, v_2) \in E \tag{4.18}$$

3. Nodes outside of the partial sequence are fully connected. Formally:

$$\forall v_1, v_2 \in V \setminus \overrightarrow{s} : v_1 \neq v_2 \iff (v_1, v_2) \in E \tag{4.19}$$

4. For every two consecutive nodes in the partial sequence, and every node outside of it, the pair of *detour edges* through this node is either fully present or fully absent. Formally:

$$\forall v_1 \rightarrow v_3, \forall v_2 \in V \setminus \overrightarrow{s} : (v_1, v_2) \in E \iff (v_2, v_3) \in E \tag{4.20}$$

Condition (4.19) is useful to ensure that the set of edges E in the graph may only decrease over time, a property used to represent the edges efficiently in the implementation later on. Forbidden precedences encoded in $\hat{F}$ only remove detour edges, leaving edges between insertable nodes untouched.

Example 4.2.3. Figure 4.3 shows an insert consistent pair $\langle \overrightarrow{s}, G(V, E) \rangle$.

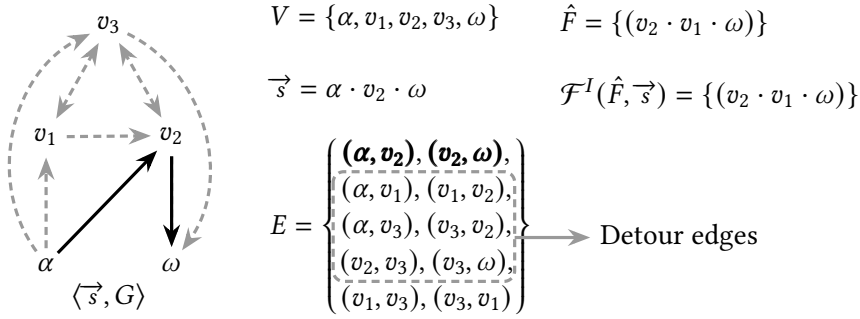


Figure 4.3: An insert consistent pair $\langle \overrightarrow{s}, G(V, E) \rangle$. Edges from the graphs are represented using dashed arrows, while the edges belonging to the sequence are continuous and shown in bold. The subsequence $(v_2 \cdot v_1 \cdot \omega)$ is forbidden, which is why the pair of detour edges $(v_2, v_1), (v_1, \omega)$ is absent.

4.2.1.3 Insertion

We next define how modifications of the partial sequence $\overrightarrow{s}$ from an insert consistent pair $\langle \overrightarrow{s}, G(V, E) \rangle$ may be performed, and their effect on the graph.

Definition 4.2.6. Let $\langle \overrightarrow{s}, G(V, E) \rangle$ be an insert consistent pair. The insert operation denoted $\longmapsto$ takes as input one insertion triplet (v_1, v_2, v_3) (*i.e.* (v_1, v_2), (v_2, v_3) is a pair of detour edges) and yields $\langle \overrightarrow{s}', G(V, E') \rangle$. This operation is formally defined as:

$$\langle \overrightarrow{s}, G(V, E) \rangle \underset{(v_1, v_2, v_3)}{\longmapsto} \langle \overrightarrow{s}', G(V, E') \rangle \tag{4.21}$$

where

- $\overrightarrow{s}' = \text{prefix}(\overrightarrow{s}, v_1) \cdot v_2 \cdot \text{suffix}(\overrightarrow{s}, v_3)$, and
- $E' = E \setminus (\{(v_1, v_3)\} \cup E_A \cup E_B)$ with

$$E_A = \{(v, v_2) \mid v \in \overrightarrow{s} \wedge v \neq v_1\} \cup \{(v_2, v) \mid v \in \overrightarrow{s} \wedge v \neq v_3\} \tag{4.22}$$

$$E_B = \{(v_2, v) \mid v \in V \setminus \overrightarrow{s} \wedge (v_1, v) \notin E\} \cup \{(v, v_2) \mid v \in V \setminus \overrightarrow{s} \wedge (v, v_3) \notin E\} \tag{4.23}$$

Lemma 4.2.1. The insert operation preserves the insert consistency.

Proof. E_A is the minimum set of edges to be removed to maintain condition (4.18) satisfied. Those are the edges linking the inserted node v_2 to other nodes than v_1 and v_3 in the partial sequence $\overrightarrow{s}$. E_B is the minimum set of edges to be removed to maintain the condition (4.20) satisfied. Indeed, prior

to the insertion, v_2 was fully connected to all other insertable nodes $V \setminus \overrightarrow{s}$. However, edges from E_B must be removed to ensure that (i) detour edges always appear in pairs in $(\overrightarrow{s}', G(V, E'))$ and (ii) any node $v \in V \setminus \overrightarrow{s}$ that could not be inserted between v_1 and v_3 before, cannot now be inserted between v_1 and v_2 or v_2 and v_3. □

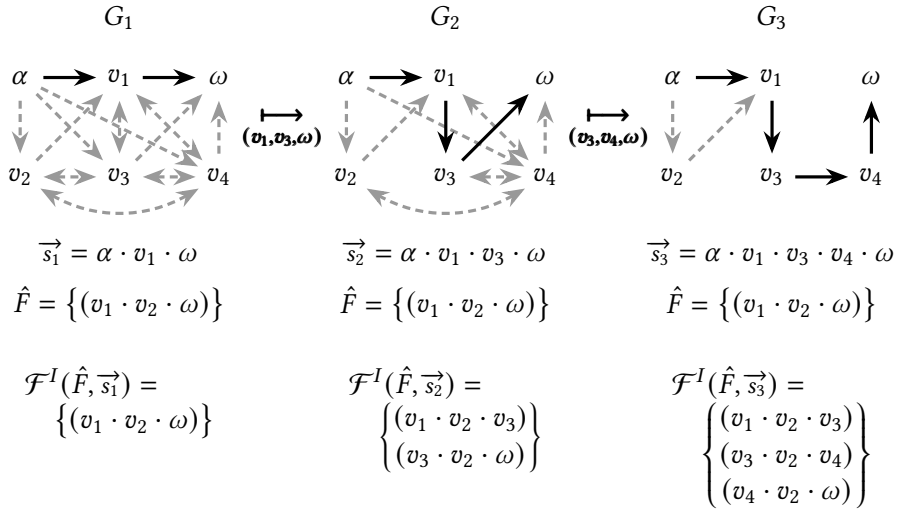


Figure 4.4: Evolution of the partial sequence $\overrightarrow{s}$, the graph $G(V, E)$ and the set $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ over multiple insertions.

Example 4.2.4. Figure 4.4 shows the evolution of the graph and the partial sequence over multiple insertions. Inserting (v_1, v_3, ω) in the first state (left) deletes the edge (v_1, ω), edges (α, v_3), (v_3, v_1) using rule (4.22) and the edges (v_2, v_3), (v_3, v_2) using rule (4.23).

This operation $\langle \overrightarrow{s}, G(V, E) \rangle \underset{(v_1, v_2, v_3)}{\longmapsto} \langle \overrightarrow{s}', G(V, E') \rangle$ has a fundamental side effect on the forbidden subsequences $\hat{F}$. When performing it, the implicit set of forbidden subsequences $\mathcal{F}^I(\hat{F}, \overrightarrow{s}')$ may be defined by means of the previous implicit set $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ only, as follows:

$$\mathcal{F}^I(\hat{F}, \overrightarrow{s}') = \bigcup_{(v_i \cdot v_j \cdot v_k) \in \mathcal{F}^I(\hat{F}, \overrightarrow{s})} \varphi(v_i \cdot v_j \cdot v_k) \tag{4.24}$$

$$\varphi(v_i \cdot v_j \cdot v_k) = \begin{cases} \{(v_i \cdot v_j \cdot v_k)\} & \text{if } v_i \overset{\overrightarrow{s}'}{\longrightarrow} v_k \\ \{(v_i \cdot v_j \cdot v_2),\ (v_2 \cdot v_j \cdot v_k)\} & \text{otherwise} \end{cases} \tag{4.25}$$

The function φ defines the evolution of a forbidden subsequence $(v_i \cdot v_j \cdot v_k)$ after the derivation $\underset{(v_1,v_2,v_3)}{\longmapsto}$ (4.25). Either the endpoints of the forbidden subsequence are still directly consecutive in the new partial sequence $\overrightarrow{s}'$ ($v_i \xrightarrow{\overrightarrow{s}'} v_k$), and the subsequence is kept. Otherwise, the endpoints are not directly consecutive anymore, which happens whenever the inserted node has been added between them: $v_i \xrightarrow{\overrightarrow{s}'} v_2 \xrightarrow{\overrightarrow{s}'} v_k$ (or equivalently $v_1 = v_i \wedge v_3 = v_k$). In this case, the subsequence $(v_i \cdot v_j \cdot v_k)$ is decomposed into two subsequences: $(v_i \cdot v_j \cdot v_2)$, $(v_2 \cdot v_j \cdot v_k)$ where the endpoints are now directly consecutive. Applying this rule on every initial forbidden subsequence defines the newly forbidden subsequences (4.24).

Example 4.2.5. Continuing Example 4.2.4, using Figure 4.4. When performing insertion $\langle \overrightarrow{s_2}, G(V,E)\rangle \underset{(v_3,v_4,\omega)}{\longmapsto} \langle \overrightarrow{s_3}, G(V,E')\rangle$ (middle to right), the forbidden subsequence $(v_1 \cdot v_2 \cdot v_3)$ remains in $\mathcal{F}^I(\hat{F}, \overrightarrow{s_3})$, while $(v_3 \cdot v_2 \cdot \omega)$ is decomposed into $(v_3 \cdot v_2 \cdot v_4)$, $(v_4 \cdot v_2 \cdot \omega)$.

Equation (4.24) is fundamental for efficiently maintaining the forbidden subsequences from $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ in the graph. It means that a forbidden subsequence $(v_1 \cdot v_2 \cdot v_3)$ is *implicitly* maintained over insertions: it suffices to follow the rule (4.24). This is translated, through rule (4.23), by an *explicit* deletion of detour edges in the graph.

4.2.1.4 NotBetween

It remains to define how adding new subsequences to $\hat{F}$ impact the insert consistent pair. This is achieved by a notBetween operation, defined next.

Definition 4.2.7. Let $\langle \overrightarrow{s}, G(V,E)\rangle$ be an insert consistent pair. The notBetween operation denoted $\underset{(v_1,v_2,v_3)}{\not\longmapsto}$ takes as input one triplet of nodes (v_1, v_2, v_3) where $v_1 \prec v_3$ and yields $\langle \overrightarrow{s}, G(V,E')\rangle$, where the subsequence $(v_1 \cdot v_2 \cdot v_3)$ has been forbidden. This operation is formally defined as

$$\langle \overrightarrow{s}, G(V,E)\rangle \underset{(v_1,v_2,v_3)}{\not\longmapsto} \langle \overrightarrow{s}, G(V,E')\rangle \tag{4.26}$$

where

$$E' = E \setminus \{(v_i, v_2), (v_2, v_j) \mid v_i, v_j \in \overrightarrow{s} \wedge v_1 \preceq v_i \rightarrow v_j \preceq v_3\} \tag{4.27}$$

The edges removed in (4.27) use the same enumeration rule as $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ in (4.14).

Lemma 4.2.2. The *notBetween* operation preserves the insert consistency.

Proof. The set of edges removed come by pairs, so that each pair deletion preserves (4.20). Those removals prevent future insertions of v_2 between v_1 and v_3 in $\overrightarrow{s}$. No edge between two nodes outside of the partial sequence is deleted, maintaining (4.19). Unless $v_1 \prec v_2 \prec v_3$, a special case corresponding to a domain wipe-out and discussed in section 4.2.1.6, the partial sequence remains a simple path using edges from E (4.18). □

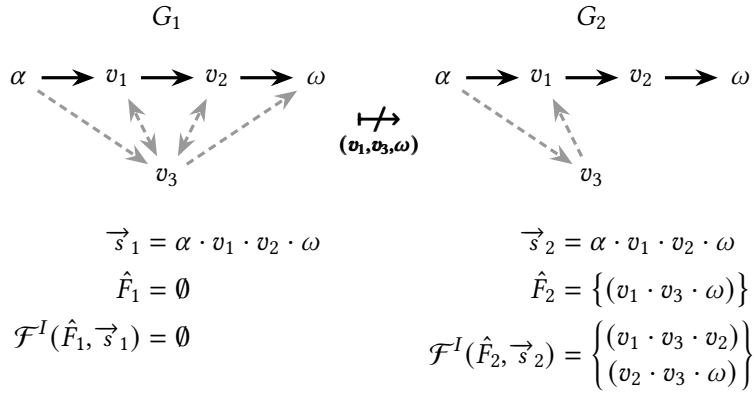


Figure 4.5: Evolution of the partial sequence $\overrightarrow{s}_1$, the graph G_1 and the set $\mathcal{F}^I(\hat{F}_1, \overrightarrow{s}_1)$ after a notBetween.

Example 4.2.6. Figure 4.5 shows the evolution of the graph and the partial sequence after a notBetween operation.

4.2.1.5 Domain Mapping

An insert consistent pair $\langle \overrightarrow{s}, G(V, E) \rangle$ implicitly contains all sequences from the domain $\mathcal{D}^{(\mathrm{s})}(\overrightarrow{s}) \cap \mathcal{D}^{(\mathrm{f})}(\hat{F})$. Those sequences can be enumerated with the help of insert and notBetween operations. To define this mapping, the derivation definition is needed.

Definition 4.2.8. The *zero or more derivation* symbol $\overset{*}{\longmapsto}$ from $\langle \overrightarrow{s}, G \rangle$ describes the application of multiple consecutive insert or notBetween operations, or to a transition to $\langle \overrightarrow{s}, G \rangle$ itself.

Example 4.2.7. In Figure 4.4, $\langle \overrightarrow{s}_3, G_3 \rangle$ can be reached from $\langle \overrightarrow{s}_1, G_1 \rangle$ through several consecutive insertions, hence $\langle \overrightarrow{s}_1, G_1 \rangle \overset{*}{\longmapsto} \langle \overrightarrow{s}_3, G_3 \rangle$.

The mapping from an insert consistent pair $\langle \overrightarrow{s}, G(V, E) \rangle$ to the domain $\mathcal{D}^{(\mathrm{s})}(\overrightarrow{s}) \cap \mathcal{D}^{(\mathrm{f})}(\hat{F})$ it represents can now be given.

Definition 4.2.9. Given a domain $\mathcal{D}^{(\mathrm{s})}(\overrightarrow{s}) \cap \mathcal{D}^{(\mathrm{f})}(\hat{F})$ represented by an insert consistent pair $\langle \overrightarrow{s}, G \rangle$. The function $\mathcal{S}(\langle \overrightarrow{s}, G \rangle)$ enumerates all sequences

that can be derived from the partial tour $\overrightarrow{s}$ through insert and notBetween operations. Formally $\mathcal{S}(\langle\overrightarrow{s}, G\rangle) = \left\{\overrightarrow{s}' \;\middle|\; \langle\overrightarrow{s}, G\rangle \overset{*}{\longmapsto} \langle\overrightarrow{s}', G'\rangle\right\}$. An insert consistent pair for $\mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(\hat{F})$ implicitly encodes this set of sequences: $\mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(\hat{F}) \equiv \mathcal{S}(\langle\overrightarrow{s}, G\rangle)$.

Example 4.2.8. Referring to $\langle\overrightarrow{s}_3, G_3\rangle$ from Figure 4.4, $\mathcal{S}(\langle\overrightarrow{s}_3, G_3\rangle) = \{\alpha \cdot v_1 \cdot v_3 \cdot v_4 \cdot \omega,\ \alpha \cdot v_2 \cdot v_1 \cdot v_3 \cdot v_4 \cdot \omega\}$.

Similarly, in Figure 4.5, $\mathcal{S}(\langle\overrightarrow{s}_2, G_2\rangle) = \{\alpha \cdot v_1 \cdot v_2 \cdot \omega,\ \alpha \cdot v_3 \cdot v_1 \cdot v_2 \cdot \omega\}$.

Definition 4.2.10. A domain $\mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(\hat{F})$ represented by $\langle\overrightarrow{s}, G\rangle$ is *fixed* when edges E from G solely contains the edges appearing in the partial sequence $\overrightarrow{s}$. Formally:

$$\text{isFixed}(\langle\overrightarrow{s}, G\rangle) \Leftrightarrow E = \left\{(u, v) \;\middle|\; u \overset{\overrightarrow{s}}{\longrightarrow} v\right\} \Leftrightarrow |\mathcal{S}(\langle\overrightarrow{s}, G\rangle)| = 1 \tag{4.28}$$

Example 4.2.9. In Figure 4.5, isFixed($\langle\overrightarrow{s}_2, G_2\rangle$) = *false*. In a pair $\langle\overrightarrow{s}_3, G_3\rangle$ obtained from $\langle\overrightarrow{s}_2, G_2\rangle$ by applying either $\underset{(\alpha, v_3, v_1)}{\not\longmapsto}$ or $\underset{(\alpha, v_3, v_1)}{\longmapsto}$, isFixed($\langle\overrightarrow{s}_3, G_3\rangle$) = *true*.

4.2.1.6 Domain Wipe-Out

An attentive reader may have seen that a forbidden sequence $(v_1 \cdot v_2 \cdot v_3) \in \hat{F}$ is not included in $\mathcal{F}^I(\hat{F}, \overrightarrow{s})$ if its nodes are all consecutive in the partial sequence: $v_1 \overset{\overrightarrow{s}}{\prec} v_2 \overset{\overrightarrow{s}}{\prec} v_3$ (4.14). This case is special as it corresponds to a domain wipe-out: the subsequence $(v_1 \cdot v_2 \cdot v_3)$ directly appears in $\overrightarrow{s}$, meaning $\mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(\hat{F}) = \emptyset$. We consider that a domain wipe-out can be represented by a flag, meaning that the domain is an empty set. No derivation (*i.e.* insertion and notBetween) can occur on a domain being empty. A domain wipe-out corresponds to a failure and triggers a backtrack.

We have so far shown how $\langle\overrightarrow{s}, G(V, E)\rangle$ can represent $\mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(\hat{F})$. To encode a complete sequence domain, we must also be able to represent its intersection with $\mathcal{D}^{\textcircled{r}}(R)$ and $\mathcal{D}^{\textcircled{x}}(X)$, which is discussed next.

4.2.2 Excluded Nodes

It is worth pointing out that when a node does not belong to the partial sequence $\overrightarrow{s}$ but has no pair of detour edges passing through it, it cannot be included in any sequence in the domain.

$$\forall v_2 \in V \setminus \overrightarrow{s} : \left(\nexists v_1 \rightarrow v_3 \text{ s.t. } (v_1, v_2), (v_2, v_3) \in E\right) \implies \left(\nexists \overrightarrow{s}' \in \mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(\hat{F}) \text{ s.t. } v_2 \in \overrightarrow{s}'\right) \tag{4.29}$$

This property can be used to represent a set of excluded nodes X in the graph representation, that may not be part of a sequence. In particular, it is used to compute the intersection $\mathcal{D} \subseteq \mathcal{D}^{\textcircled{S}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{T}}(\hat{F}) \cap \mathcal{D}^{\textcircled{X}}(X)$:

$$v_2 \in X \implies \nexists v_1 \xrightarrow{\overrightarrow{s}} v_3 \text{ s.t. } (v_1, v_2), (v_2, v_3) \in E \tag{4.30}$$

Equation (4.30) prevents from generating any sequence where an excluded node $v \in X$ is visited, by removing all detour edges passing through it (4.29).

Furthermore, to avoid maintaining edges that may never be used for insertions (*i.e.* that may never become detour edges), invariant (4.19) is modified as follows.

$$\forall v_1, v_2 \in V \setminus \overrightarrow{s} \setminus X : v_1 \neq v_2 \iff (v_1, v_2) \in E \tag{4.31}$$

$$\forall v_x \in X, \forall v_i \in V : (v_i, v_x) \notin E \wedge (v_x, v_i) \notin E \tag{4.32}$$

Condition (4.31) replaces (4.19): instead of forcing nodes outside the partial sequence to be fully connected, only the insertable nodes are fully connected. Nodes being excluded have no edge passing through them (4.32).

4.2.3 Mandatory Nodes

To obtain the complete sequence domain, we must finally compute the intersection $\mathcal{D}^{\textcircled{S}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{T}}(\hat{F}) \cap \mathcal{D}^{\textcircled{X}}(X) \cap \mathcal{D}^{\textcircled{I}}(R) = \mathcal{D}$, where a set of required nodes R is specified.

As expressed by (4.29), a node $v_2 \in V \setminus \overrightarrow{s}$ that does not belong to the partial sequence cannot be included in any sequence in the domain if it has no detour edge passing through it. Therefore, a domain wipe-out would occur if a node $v \in R \setminus \overrightarrow{s}$ required but not member of the partial sequence $\overrightarrow{s}$ has no detour edge passing through it. To prevent this situation, such required nodes are inserted when only one pair of detour edge remains for them. More specifically, we introduce a counter nI_j for every node $v_j \in V$, tracking how many pairs of detour edges passing through v_j exist.

$$nI_j = \begin{cases} \left|\left\{(v_i, v_j) \mid (v_i, v_j) \in E \wedge v_i \in \overrightarrow{s}\right\}\right| & \text{if } v_j \notin \overrightarrow{s} \\ 0 & \text{otherwise} \end{cases} \tag{4.33}$$

Those counters are incrementally updated whenever $\hat{F}$ and $\overrightarrow{s}$ change:

1. Every time a pair of detour edge $(v_i, v_j), (v_j, v_k)$ is deleted from the set of edges E, the counter for the node v_j is decremented: $nI_j \leftarrow nI_j - 1$.

2. Every time an insertion (v_i, v_j, v_k) is performed, all nodes $v_l \in V \setminus \overrightarrow{s}$ are scanned. If a node v_l could previously be inserted between v_i, v_k (because $(v_i, v_l), (v_l, v_k) \in E$), it may now be inserted between both v_i, v_j and v_j, v_k, which increases its counter: $nI_l \leftarrow nI_l + 1$.

   Furthermore, as $\overrightarrow{s}$ is a simple path, the node v_j being inserted cannot be inserted anymore, and its counter nI_j is thus set to zero.

If a counter nI_j takes value 1 and the node is required ($v_j \in R$), then it is inserted using its only remaining insertion. This means that the partial sequence $\overrightarrow{s}$ may grow automatically, depending on the nodes that are required. Given that we are interested in computing the sequence domain $\mathcal{D}$, being the intersection of the four subdomains, such automated insertions are worthwhile to perform. To prevent situations where more than one node is both required and has only one insertion, automated insertions are performed as soon as this condition holds, by adding nodes into R one at a time, and inserting nodes into $\overrightarrow{s}$ one at a time.

4.2.4 Compact Domain Implementation

The last sections showed how to represent forbidden subsequences by a graph $G(V, E)$ and a partial sequence $\overrightarrow{s}$, and how to cope with a set of required nodes R and excluded nodes X. This section now proposes an implementation of those elements, called *compact domain implementation*.

The compact implementation of a sequence domain $\mathcal{D} = \langle R, X, \overrightarrow{s}, F\rangle = \mathcal{D}^{\textcircled{r}}(R) \cap \mathcal{D}^{\textcircled{x}}(X) \cap \mathcal{D}^{\textcircled{s}}(\overrightarrow{s}) \cap \mathcal{D}^{\textcircled{f}}(F)$ is written $\hat{\mathcal{D}}$. The following assumes that all sequence domains use the compact domain implementation, and we thus assume $\hat{\mathcal{D}} = \langle R, X, \overrightarrow{s}, F\rangle$.

To be integrated in a standard trailed-based CP solver, a compact domain implementation $\hat{\mathcal{D}}$ need to be reversible, for instance through trailing. It relies on the following reversible data structures, most of which were already present in [TKS20; Tho23; DSV22]:

- The partial sequence $\overrightarrow{s}$ is encoded using a successor array of reversible integers $\mathbf{s}^+$. It stores a pointer to the current successor of each node belonging to the partial sequence $\overrightarrow{s}$, and an element without successor (*i.e.* a node $v \in V \setminus \overrightarrow{s}$) points towards itself (self-loop). Similarly, an additional array of reversible integers $\mathbf{s}^-$ tracks the current predecessors of each node.

- The set of edges E is maintained by two adjacency sets per node $v \in V$: E_v^- (incoming) and E_v^+ (outgoing) edges: $E_v^- = \{(v_i, v) \mid v_i \in V, (v_i, v) \in E\}, E_v^+ = \{(v, v_i) \mid v_i \in V, (v, v_i) \in E\}$. Those sets are implemented using the reversible sparse-sets introduced in [Sai+13], allowing deletion and state restoration in constant time in a trail-based solver.

- A set I, tracking the insertable nodes, is also maintained using a reversible sparse-set.
- The size of the partial sequence $\overrightarrow{s}$ is tracked in a reversible integer nS.
- The counters nI_j of detour edges (4.33) are maintained with reversible integers, tracking how many insertions exist for every node $v_j \in V$.
- Given that R and X are necessarily disjoint and subsets of V, we use a reversible sparse-set with two size markers [Sai+13], ensuring removal of nodes and state restoration in constant time, and enabling iteration over R, X in $O(|R|), O(|X|)$, respectively.

  The nodes not required nor excluded are denoted $P = V \setminus X \setminus R$, and described as *possible* nodes.

The data structures used for implementing a compact domain $\hat{\mathcal{D}}$ are depicted in Figure 4.6.

4.2.4.1 Initialization

When initialized, a sequence variable is defined over the set of nodes V belonging to the graph $G(V, E)$, and its successor array $\mathbf{s}^+$ contains one entry per node $v \in V$. The two starting and ending nodes, α and ω, are identified upon the initialization of the variable. These nodes constitute the initial sequence of nodes represented by the variable, forming a cycle in the successor array.

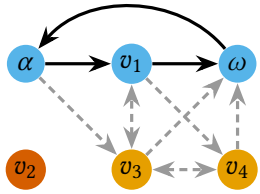
$$\mathbf{s}^+_\alpha = \omega \wedge \mathbf{s}^-_\omega = \alpha \wedge \mathbf{s}^+_\omega = \alpha \wedge \mathbf{s}^-_\alpha = \omega \tag{4.34}$$

Note that a link from the end node ω to the first node α is encoded, thus representing a cycle instead of a path, to ease invariant encoding in further equations. The other nodes point to themselves as self-loops. At the initialization, the edge set forms a complete graph, except edges originating at ω and ending at α: $E = \{(v_1, v_2) \mid v_1, v_2 \in V : v_1 \neq \omega \wedge v_2 \neq \alpha \wedge v_1 \neq v_2\} \cup \{(\omega, \alpha)\}$. The set of nodes that may be inserted is defined as $I = V \setminus \{\alpha, \omega\}$, and each of those nodes has one insertion at first: $nI_i = 1 \forall v_i \in I$.

4.2.4.2 Invariants

First, let us introduce one predicate that will be instrumental in defining the consistency invariants maintained on the domain. It determines (in constant time) if a node v_i belongs to the partial sequence (*i.e.* if $v_i \in \overrightarrow{s}$ holds) by verifying if it is not a self-loop:

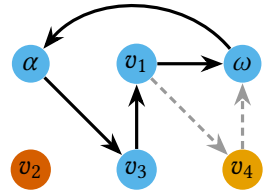
$$\text{isMember}(v_i) \equiv \mathbf{s}^+_i \neq v_i. \tag{4.35}$$



	E^-	E^+	nI
α	$\{\omega\}$	$\{v_1, v_3\}$	0
v_1	$\{\alpha, v_3\}$	$\{v_3, v_4, \omega\}$	0
v_2	$\emptyset$	$\emptyset$	0
v_3	$\{\alpha, v_1, v_4\}$	$\{v_1, v_4, \omega\}$	2
v_4	$\{v_1, v_3\}$	$\{v_3, \omega\}$	1
ω	$\{v_1, v_3, v_4\}$	$\{\alpha\}$	0

$\mathbf{s}^+$	ω	v_2	v_3	v_4	v_1	α
	v_1	v_2	v_3	v_4	α	ω
$\mathbf{s}^-$	α	v_2	v_3	v_4	ω	v_1
	v_1	v_2	v_3	v_4	α	ω

$nS = 3, I = \{v_3, v_4\}$
$R = \{\alpha, \omega, v_1\}, X = \{v_2\}$



	E^-	E^+	nI
α	$\{\omega\}$	$\{v_3\}$	0
v_1	$\{v_3\}$	$\{v_4, \omega\}$	0
v_2	$\emptyset$	$\emptyset$	0
v_3	$\{\alpha\}$	$\{v_1\}$	0
v_4	$\{v_1\}$	$\{\omega\}$	1
ω	$\{v_1, v_4\}$	$\{\alpha\}$	0

$\mathbf{s}^+$	ω	v_2	v_1	v_4	v_3	α
	v_1	v_2	v_3	v_4	α	ω
$\mathbf{s}^-$	v_3	v_2	α	v_4	ω	v_1
	v_1	v_2	v_3	v_4	α	ω

$nS = 4, I = \{v_4\}$
$R = \{\alpha, \omega, v_1, v_3\}, X = \{v_2\}$

Figure 4.6: Compact domain implementation. On the left, the partial sequence $\overrightarrow{s}$ and the graphs $G(V, E)$ are shown. Below them is a table showing the edges E^-, E^+ for the nodes and the counters of insertions nI. Below the table, the successors $\mathbf{s}^+$ and the predecessors $\mathbf{s}^-$ of the nodes (only relevant for highlighted nodes $v \in \overrightarrow{s}$) are shown. The right part shows the domain after performing an insertion with (α, v_3, v_1), extending the partial sequence and removing some edges due to (4.22), (4.23).

The lower-level consistency invariant expressed on the data structures are given next. We first identify the invariants describing the channeling between the two data structures.

$$\forall v_i, v_j \in V : v_i \in E_j^- \iff v_j \in E_i^+ \tag{4.36}$$

$$\forall v_i, v_j \in V : \mathbf{s}_i^+ = v_j \iff \mathbf{s}_j^- = v_i \tag{4.37}$$

Invariant (4.36) ensures that each edge (v_i, v_j) appears twice in the data structures: one for the adjacency set of the incoming node, and one for the outgoing node. Invariant (4.37) ensures that for any two nodes v_i and v_j in the graph, v_i is the predecessor of v_j if and only if v_j is the successor of v_i. It

guarantees a consistent and mutual relationship between successor and predecessor links for every node in the graph. In addition to the channeling, specific invariants are used to maintain the counters previously introduced.

$$nS = |\{v_i \mid v_i \in V \wedge \text{isMember}(v_i)\}| \tag{4.38}$$

$$\forall v_j \in I : nI_j = |\{v_i \mid v_i \in E_j^- \wedge \text{isMember}(v_i)\}| \tag{4.39}$$

$$\forall v_i \in V : (nI_i \geq 1) \leftrightarrow (v_i \in I) \tag{4.40}$$

The length of the partial sequence is tracked in (4.38). Invariant (4.39) tracks how many insertions are feasible for a given node $v_j \in I$, and is equivalent to (4.33). This counter is used to ensure that every node $v_j \in I$ has at least one insertion possible (4.40): otherwise the node is not insertable and thus has 0 insertions.

Given the array of successors $\mathbf{s}^+$, one can define the set of nodes reachable from a circuit containing node $v_i \in V$:

$$\text{circuit}(v_i) = \text{circuit}(v_i, \emptyset) \tag{4.41}$$

$$\text{circuit}(v_i, S) = \begin{cases} S & \text{if } v_i \in S \\ \text{circuit}(\mathbf{s}_i^+, S \cup \{v\}) & \text{otherwise} \end{cases} \tag{4.42}$$

Intuitively from (4.41), $\text{circuit}(v_i)$ gives all nodes in the sequence from α to ω if v_i is in the sequence, otherwise it returns a set containing only v_i. This is done by following recursively the pointers $\mathbf{s}^+$ of the successor array. Using this definition, the implementation invariants used to enforce insert-consistency (4.18)-(4.20) are as follows.

$$\mathbf{s}_\omega^+ = \alpha \wedge \mathbf{s}_\alpha^- = \omega \tag{4.43}$$

$$\forall v_i, v_j \in V, v_i \neq v_j : \left(\mathbf{s}_i^+ = v_j \implies v_j \in E_i^+\right) \wedge \left(\mathbf{s}_j^- = v_i \implies v_i \in E_j^-\right) \tag{4.44}$$

$$\forall v_i, v_j \in V, v_i \neq v_j : \mathbf{s}_i^+ \neq \mathbf{s}_j^+ \tag{4.45}$$

$$\forall v_i \in V : \text{isMember}(v_i) \iff \text{circuit}(v_i) = \text{circuit}(\alpha) \tag{4.46}$$

$$\forall v_i, v_j \in V, v_i \neq v_j : \neg\text{isMember}(v_i) \wedge \neg\text{isMember}(v_j) \implies v_i \in E_j^- \tag{4.47}$$

$$\forall v_i, v_j, v_k \in V, v_i \neq v_j \neq v_k : \mathbf{s}_i^+ = v_k \implies \left(v_i \in E_j^- \iff v_k \in E_j^+\right) \tag{4.48}$$

Invariant (4.43) ensures that the successor of the last node ω being visited always points toward the first node α. Invariant (4.44) enforces that the current successor of a node $v_i \in \overrightarrow{s}$ exists within the outgoing edges of the node v_i, so that (4.18) holds. Invariants (4.45) and (4.46) ensure that only one sub-circuit is encoded within the successor array. All successors must be different, and if the successor of a node $v_i \in V$ is set (*i.e.* v_i belongs to the partial sequence) then v_i belongs to the circuit of the first node α (and the last node

ω given that circuit(α) = circuit(ω)). Invariant (4.47) enforces that all insertable nodes form a clique, so that (4.19) holds. Finally, (4.48) ensures that the two detour edges $(v_i, v_j), (v_j, v_k)$ needed to insert a node $v_j \in I$ between consecutive nodes v_i and v_k are either both absent or both present, enforcing (4.20).

Lastly, some invariants interact specifically with the required R and excluded nodes X. Moreover, the implementation may modify itself the set of required nodes R, so that it always contains nodes from the partial sequence: $\overrightarrow{s} \subseteq R$, given that nodes who are part of the partial sequence are always visited in all sequences from the domain. The invariants are:

$$v_i \in X \iff E_i^- = \emptyset \iff E_i^+ = \emptyset \tag{4.49}$$

$$\forall v_i \in V : \text{isMember}(v_i) \implies v_i \in R \tag{4.50}$$

$$\forall v_i \in R \cap I : nI_i > 1 \tag{4.51}$$

Invariant (4.49) ensures that an excluded node has no edge attached to it. Invariant (4.50) captures the fact that nodes who are part of the partial sequence $\overrightarrow{s}$ are always visited, and thus considered as mandatory. Finally, invariant (4.51) guarantees that required nodes not part of the sequence have at least two insertions remaining. Otherwise, if only one pair of detour edges remained for a node, it would directly be used to add the node to the partial sequence $\overrightarrow{s}$.

4.2.4.3 API and Time Complexity

A sequence domain is fixed whenever no insertions remain (Definition 4.2.10). The worst-case time complexity for constructing a sequence is thus $O(|E|)$. In the worst case, all edges are removed except those forming the sequence.

Table 4.3 lists all query operations over a sequence variable domain, along with their associated time complexities. The domain updates are described in Table 4.4. Given that a compact domain $\hat{\mathcal{D}}$ includes an insert consistent pair $\langle \overrightarrow{s}, G(V, E) \rangle$, it inherits its derivations $\underset{(v_1,v_2,v_3)}{\longmapsto}$ (insertion) and $\underset{(v_1,v_2,v_3)}{\not\longmapsto}$ (notBetween) from Definitions 4.2.6 and 4.2.7.

4.2.4.4 Domain Updates

The implementation of the domain updates are described next. Some of them may trigger a domain wipe-out, depending on the provided arguments.

Insertion Algorithm 5 is used to perform an Sq.insert(v_1, v_2) operation on a compact sequence domain Sq, provided that (v_1, v_2, v_3) with $v_1 \rightarrow v_3$ define an insertion triplet. The inserted node v_2 is first marked as required, removed

Operation	Description	Complexity
isFixed()	Returns true if no more insertions remain	$O(1)$
isMember(v_i)	Returns true if $v_i \in \overrightarrow{s}$	$O(1)$
isRequired(v_i)	Returns true if the node v_i is required	$O(1)$
isExcluded(v_i)	Returns true if the node v_i is excluded	$O(1)$
isPossible(v_i)	Returns true if the node v_i is possible	$O(1)$
isInsertable(v_i)	Returns true if the node v_i is insertable	$O(1)$
getNext(v_i)	Returns the successor $\mathbf{s}_i^+$ of node v_i	$O(1)$
getPrev(v_i)	Returns the predecessor $\mathbf{s}_i^-$ of node v_i	$O(1)$
nInsert(v_i)	Returns the number of insertions for v_i	$O(1)$
nMember()	Returns the size nS of the partial sequence	$O(1)$
getMember()	Enumerates nodes in the partial sequence $\overrightarrow{s}$	$\Theta(\lvert\overrightarrow{s}\rvert)$
getRequired()	Enumerates the required nodes R	$\Theta(\lvert R\rvert)$
getExcluded()	Enumerates the excluded nodes X	$\Theta(\lvert X\rvert)$
getPossible()	Enumerates the possible nodes P	$\Theta(\lvert P\rvert)$
getInsertable()	Enumerates the insertable nodes I	$\Theta(\lvert I\rvert)$
getEdgesTo(v_i)	Enumerates E_i^-	$\Theta(\lvert E_i^-\rvert)$
getEdgesFrom(v_i)	Enumerates E_i^+	$\Theta(\lvert E_i^+\rvert)$
canInsert(v_i, v_j)	Returns true if (v_i, v_j, getNext(v_i)) is an insertion triplet	$O(1)$
getInsert(v_j)	Enumerates $\{ v_i \mid v_i \in E_j^- \wedge \text{canInsert}(v_i, v_j) \}$	$\Theta(\min(\lvert\overrightarrow{s}\rvert, \lvert E_j^-\rvert))$
getInsertAfter(v_j, p)	Enumerates $\{ v_i \mid p \prec v_i \wedge \text{canInsert}(v_i, v_j) \}$	$O(\lvert\overrightarrow{s}\rvert)$

Table 4.3: Queries on a compact sequence domain.

Operation	Update	Complexity
notBetween(v_i, v_j, v_k)	$\overset{\vdash\!\!\not\!\!\rightarrow}{\scriptstyle (v_i, v_j, v_k)}$	$\begin{cases} O(\lvert I\rvert) & \text{if } v_i \rightarrow v_k \\ O(\lvert V\rvert) & \text{otherwise} \end{cases}$
insert(v_i, v_j)	$\overset{\vdash\!\!\rightarrow}{\scriptstyle (v_i, v_j, \text{getNext}(v_i))}$	$\Theta(\lvert E_j^-\rvert)$
insertAtEnd(v_i)	$\overset{\vdash\!\!\rightarrow}{\scriptstyle (\text{getPrev}(\omega), v_i, \omega)}$	$\Theta(\lvert E_i^-\rvert)$
require(v_i)	$R \leftarrow R \cup \{ v_i \}$	$O(\lvert E_i^-\rvert)$
exclude(v_i)	$X \leftarrow X \cup \{ v_i \}$	$O(\lvert E_i^-\rvert)$

Table 4.4: Updates on a compact sequence domain.

from the insertable nodes and the size nS of the partial sequence $\overrightarrow{s}$ is incremented (lines 1 to 4). Then, every node v_i linked to v_2 is inspected (line 6). If a node v_i belongs to both the partial sequence $\overrightarrow{s}$ and to the ingoing edges

of v_2, it previously defined an insertion triplet $(v_i, v_2, \mathbf{s}_i^+)$ for v_2. The corresponding detour edges for v_2 are removed, according to (4.22) (lines 8 to 11). Otherwise, if v_i did not belong to $\overrightarrow{s}$, lines 13 and 14 ensure that detour edges appear together, deleting invalid links per (4.23). Finally, if the node v_i could be inserted previously between v_1 and v_3, it can now be inserted between both v_1, v_2 and v_2, v_3, increasing its counter nI_i (line 16). Lastly, the edges, successor and predecessor of v_1, v_2 and v_3 are updated to reflect the insertion (lines 17 to 19).

Algorithm 5: Sq.insert(v_1, v_2)

Input: Sq: compact sequence domain, v_1 the predecessor after which the insertion must be done, v_2 node to insert

Precondition: node v_2 is insertable after node v_1 (*i.e.* $(v_1, v_2), (v_2, v_3)$ is a pair of detour edges, with $v_1 \rightarrow v_3$)

```
1  R ← R ∪ {v_2}
2  I ← I \ {v_2}
3  nI_2 ← 0
4  nS ← nS + 1
5  v_3 ← s_1^+
6  for v_i ∈ E_2^- do
7      if Sq.isMember(v_i) then
8          if v_i ≠ v_1 then
9              E_i^+ ← E_i^+ \ {v_2}, E_2^- ← E_2^- \ {v_i}
10             v_j ← s_i^+
11             E_2^+ ← E_2^+ \ {v_j}, E_j^- ← E_j^- \ {v_2}
12     else if not Sq.canInsert(v_1, v_i) then
13         E_2^+ ← E_2^+ \ {v_i}, E_i^- ← E_i^- \ {v_2}
14         E_i^+ ← E_i^+ \ {v_2}, E_2^- ← E_2^- \ {v_i}
15     else
16         nI_i ← nI_i + 1
17 s_1^+ ← v_2, s_2^+ ← v_3
18 s_3^- ← v_2, s_2^- ← v_1
19 E_1^+ ← E_1^+ \ {v_3}, E_3^- ← E_3^- \ {v_1}
```

In the implementation, if (v_1, v_2, v_3) does not define an insertion triplet, two situations may occur:

1. If v_2 is already within the partial sequence and lies after v_1 (*i.e.* $v_1, v_2 \in \overrightarrow{s} \wedge v_1 \prec v_2$), the insertion is considered as having already been performed. Nothing happens in this case.

2. Otherwise, v_2 cannot be inserted. This corresponds to a domain wipe-

out.

NotBetween Algorithm 6 performs a Sq.notBetween(v_1, v_2, v_3) operation on a compact sequence domain Sq. It iterates over the nodes v_i between the nodes v_1 and v_3 (line 1), removes the detour edges allowing to insert v_2 after v_i, and decrements its counter of insertion nI_2 (lines 4 to 6). If the counter of insertion reaches 0, the node must be excluded due to (4.40), hence removing the node v_2 from I and marking it as excluded (lines 8 and 9). In this case, all edges passing through the node are removed. In contrast, if the counter of insertion reaches 1 and the node is required (line 14), it is automatically inserted at its only remaining insertion (lines 15 and 16).

Algorithm 6: Sq.notBetween(v_1, v_2, v_3)

Input: Sq: compact sequence domain, v_1, v_3 two nodes in the partial sequence $\overrightarrow{s}$ between which node $v_2 \in V \setminus \overrightarrow{s}$ cannot appear.

```
1  for v_i ∈ s⃗ | v_1 ⪯ v_i ≺ v_3 do
2      if Sq.canInsert(v_i, v_2) then
3          v_j ← s_i^+
4          E_2^- ← E_2^- \ {v_i}, E_i^+ ← E_i^+ \ {v_2}
5          E_j^- ← E_j^- \ {v_2}, E_2^+ ← E_2^+ \ {v_j}
6          nI_2 ← nI_2 − 1
7          if nI_2 = 0 then
8              X ← X ∪ {v_2}
9              I ← I \ {v_2}
10             for v_k ∈ I do
11                 E_k^- ← E_k^- \ {v_2}, E_k^+ ← E_k^+ \ {v_2}
12             E_2^- ← ∅, E_2^+ ← ∅
13             return
14 if nI_2 = 1 and v_2 ∈ R then
15     {v_i} ← Sq.getInsert(v_2)
16     Sq.insert(v_i, v_2)
```

Algorithm 6 is frequently called with v_3 being the direct successor of v_1: $v_1 \rightarrow v_3$. If so, only one iteration occurs at line 1, with $v_i = v_1$, and edge deletion occurs in constant time if v_2 does not become excluded or inserted.

Similarly to the insert operation, a specific situation must be checked in the implementation if (v_1, v_2, v_3) does not define a pair of detour edges. In case all nodes belong to the sequence and are consecutive ($v_1, v_2, v_3 \in \overrightarrow{s} \wedge v_1 \prec v_2 \prec v_3$), a domain wipe-out is triggered. No filtering occurs if $v_3 \preceq v_1$.

Require Requiring a node v_i is simply obtained by marking the node as required ($R \leftarrow R \cup \{v_i\}$) and inserting it in case only insertion remained for it ($nI_i = 1$).

In case the node was excluded, a domain wipe-out occurs.

Exclude Excluding a node v_i is simply obtained by marking the node as excluded ($X \leftarrow X \cup \{v_i\}$) and removing all edges connected to it. Even though edges are removed within the graph, only the insertion counter nI_i is affected. The counters related to other nodes $v_j \in I \wedge v_i \neq v_j$ are not changed by edge removals, as any edge (v_i, v_j) (or (v_j, v_i)) being removed could not be a detour edge: no endpoint of the edge is within the partial sequence $\overrightarrow{s}$.

In case the node was required, a domain wipe-out occurs.

4.2.4.5 Visit of Nodes as Boolean Variables

Given the set of mandatory nodes R, one can easily create a binary variable $\mathcal{R}_i(\overrightarrow{S})$ for any node $v_i \in V$, telling if the node is visited (value 1: $v_i \in R$) or not (value 0: $v_i \notin R$) by a sequence variable $\overrightarrow{S}$ with domain $\mathcal{D}$. In CP, this can be implemented as a *view* over the compact sequence domain, making the usage of such variables cheap once a sequence variable has been created, as in [ST13; VM14; MSV21]. Fixing a boolean variable $\mathcal{R}_i(\overrightarrow{S})$ to 1 may automatically insert it within the partial sequence.

Thanks to the usage of those binary variables, one can easily enforce logical constraints over sequences variables. For instance to force a sequence $\overrightarrow{S}$ to visit at least n nodes ($\sum_{v_i \in V} \mathcal{R}_i(\overrightarrow{S}) \geq n$), enforce two nodes $v_i, v_j \in V$ to always be visited together ($\mathcal{R}_i(Sq) = \mathcal{R}_j(Sq)$), etc. More complex constraints, with specific propagators, are presented in section 4.3.

Operation on boolean variable		**Corresponding operation on $\overrightarrow{S}$**
Queries		
$\lvert\mathcal{D}(\mathcal{R}_i(\overrightarrow{S}))\rvert = 1$	$\iff$	$\neg\overrightarrow{S}$.isPossible(v_i)
$false \in \mathcal{D}(\mathcal{R}_i(\overrightarrow{S}))$	$\iff$	$\neg\overrightarrow{S}$.isRequired(v_i)
$true \in \mathcal{D}(\mathcal{R}_i(\overrightarrow{S}))$	$\iff$	$\neg\overrightarrow{S}$.isExcluded(v_i)
Updates		
$\mathcal{D}(\mathcal{R}_i(\overrightarrow{S})) \leftarrow \{true\}$	$\iff$	$\overrightarrow{S}$.require(v_i)
$\mathcal{D}(\mathcal{R}_i(\overrightarrow{S})) \leftarrow \{false\}$	$\iff$	$\overrightarrow{S}$.exclude(v_i)

Table 4.5: Operations on a boolean variable $\mathcal{R}_i(Sq)$ defined over a node v_i in a sequence variable $\overrightarrow{S}$. The top 3 operations are queries over the domain, while the bottom 2 are domain updates.

4.2.4.6 Space Complexity

Each sparse-set of n values requires $2n$ entries (the two arrays used by the sparse-set) and 1 reversible integer. Combining all sparse-sets in the domain implementation with the other reversible integers that are maintained, this gives a total of $4n^2+4n$ entries for integer arrays, and $5n+4$ reversible integers, with n the number of nodes on which the sequence variable is defined.

In comparison, a successor model encoding only the outgoing edges for each node would require one sparse-set per node, giving $2n^2$ entries for integer arrays and n reversible integers. In both cases, the space complexity is quadratic with respect to n. However, on a VRP with k vehicles, a successor model remains in $O(n^2)$ while a model using sequence variables is in $O(k \cdot n^2)$.

4.3 Constraints

Many useful global constraints can be defined on sequence variables. This section only focuses on those required by numerous VRP applications, such as the Dial-a-Ride Problem.

Consistency of a constraint on a sequence variable. Similar to standard consistency notions for filtering algorithms over finite integer domains, one can define consistency properties for a constraint over a sequence domain.

In the next definitions, we will denote with $\mathcal{S}(c)$ all sequences being solutions to a constraint c.

Definition 4.3.1. A constraint c over a sequence domain $\hat{\mathcal{D}}$ is *insert consistent* if and only if every insertion triplet on the domain participates in a derivation compatible with the constraint. More formally, if and only if for every insertion $\hat{\mathcal{D}} \underset{(v_1,v_2,v_3)}{\longmapsto} \hat{\mathcal{D}}'$, we have that $\hat{\mathcal{D}}' \cap \mathcal{S}(c) \neq \emptyset$.

Definition 4.3.2. Given a sequence domain $\hat{\mathcal{D}} = \langle R, X, \overrightarrow{s}, F \rangle$, we define its *required-relaxed sequence domain* as $\tilde{\mathcal{D}} = \langle \{\emptyset, X, \overrightarrow{s}, F \rangle$, which considers that only nodes in the partial sequence are required. A constraint c over a sequence domain $\hat{\mathcal{D}}$ is *relaxed-insert consistent* if and only if every insertion triplet on its required-relaxed sequence domain participates in a derivation compatible with the constraint. More formally, if and only if for every insertion $\tilde{\mathcal{D}} \underset{(v_1,v_2,v_3)}{\longmapsto} \tilde{\mathcal{D}}'$, we have that $\tilde{\mathcal{D}}' \cap \mathcal{S}(c) \neq \emptyset$.

Given that $\emptyset \subseteq R$, the *relaxed-insert consistency* property for a constraint c is thus a relaxed form of *insert consistency*. The filtering algorithms that we introduce are relatively basic and aim to reach *relaxed-insert consistency* rather than *insert consistency*, which may be NP-hard to reach for some constraints.

Those definitions are quite different from, for instance, domain consistency for integer variables, which require all values in the domain to belong to a solution. This is due to the fact that an arbitrary sequence from the domain cannot be removed: the domain may only be modified through the updates described in Table 4.4. Therefore, we can only restrict the domain updates that are allowed, which translates to limiting the insertions that may be performed with those definitions.

4.3.1 Distance

The Distance constraint is used to represent the travel length in a sequence of nodes. It links an integer variable *Dist* to the traveled distance between nodes in a sequence variable, through transitions defined in a matrix $\boldsymbol{d} \in \mathbb{Z}^{|V|\times|V|}$ satisfying the triangular inequality[2].

$$\text{Distance}(\overrightarrow{S}, \boldsymbol{d}, \mathit{Dist}) \leftrightarrow \sum_{v_i \xrightarrow{\vec{s}} v_j} \boldsymbol{d}_{i,j} = \mathit{Dist} \tag{4.52}$$

Filtering The filtering is presented in Algorithm 7. First, the traveled distance over the partial sequence $\overrightarrow{s}$ is computed (line 1). If the sequence variable is fixed, this fixes the value of the integer variable *Dist* (line 5). Otherwise, this is used to set its partial sequence (line 5) and compute the length of the longest detour still possible (line 6). All insertions for every insertable node $v_j \in I$ are then looked, and if the cost of the detour for inserting a node v_j between two consecutive nodes v_i and v_k is too high, it is removed (lines 10 to 12). In the worst case, the filtering empties E, safe for the edges belonging to the current sequence $\overrightarrow{s}$, giving a time complexity of $O(|E|)$.

Algorithm 7 follows a structure used in most filtering algorithms over sequence variables. The partial sequence $\overrightarrow{s}$ is first traversed and potentially used to filter other variables. Then, insertion triplets are inspected and possibly removed if their related detour is invalid.

This filtering is not idempotent: after applying a filtering on a domain, another application of the same filtering algorithm may further change the domain. Indeed, removing a detour for node v_j at line 12 may insert v_j due to a past require operation, hence requiring to recompute the length and trigger again the filtering, possibly causing further changes. Algorithm 7 needs thus to be triggered at every insertion in order to reach a fixpoint. More complex filtering may be considered, for instance, updating the distance upper bound

[2]The triangular inequality over a matrix $\boldsymbol{d} \in \mathbb{Z}^{|V|\times|V|}$ holds iff $\boldsymbol{d}_{i,j} + \boldsymbol{d}_{j,k} \geq \boldsymbol{d}_{i,k}\ \forall i, j, k \in \{0, \ldots, |V| - 1\}$.

Algorithm 7: Distance($\overrightarrow{S}, \boldsymbol{d}, Dist$) constraint filtering.

```
1  length ← Σ_{v_i --s--> v_j} d_{i,j}
2  if S.isFixed() then
3  |  Dist ← length
4  else
5  |  ⌊Dist⌋ ← max(length, ⌊Dist⌋)
6  |  maxDetour ← ⌈Dist⌉ − length
7  |  for v_j ∈ S.getInsertable() do
8  |  |  for v_i ∈ S.getInsert(v_j) do
9  |  |  |  v_k ← S.getNext(v_i)
10 |  |  |  cost ← d_{i,j} + d_{j,k} − d_{i,k}
11 |  |  |  if cost > maxDetour then
12 |  |  |  |  S.notBetween(v_i, v_j, v_k)
```

$\lceil Dist \rceil$ by a minimum spanning tree computation over the remaining edges of the graph.

4.3.2 TransitionTimes

The TransitionTimes constraint is used for problems where node visits involve a service time and are restricted by time window constraints, with a transition time required to move from one node to the next, as specified by a transition time matrix. More formally, each node $v_i \in V$ is attached to an integer variable $\mathbf{Start}_i$ representing the start of the service at that node and a service duration value $\boldsymbol{s}_i$. A matrix $\boldsymbol{d} \in \mathbb{Z}^{|V|\times|V|}$ defines the transition times between elements and satisfies the triangular inequality. The definition of the constraint is then:

$$\text{TransitionTimes}(\overrightarrow{S}, \mathbf{Start}, \boldsymbol{s}, \boldsymbol{d}) \leftrightarrow \forall v_i \overset{\overrightarrow{S}}{\prec} v_j : \mathbf{Start}_i + \boldsymbol{s}_i + \boldsymbol{d}_{i,j} \leq \mathbf{Start}_j \quad (4.53)$$

We consider that waiting at a given node (*i.e.* reaching it before its time window without beginning the task related to it) is possible, which is why (4.53) uses inequalities. Moreover, the start variable $\mathbf{Start}_v$ of an excluded node $v \notin \overrightarrow{S}$ is not constrained.

Filtering The filtering occurs in two steps. First, the bounds of the time windows related to visited nodes are updated by iterating over the partial sequence $\overrightarrow{s}$, in order:

$$\lfloor \mathbf{Start}_j \rfloor \leftarrow \max\left(\lfloor \mathbf{Start}_j \rfloor, \lfloor \mathbf{Start}_i \rfloor + s_i + d_{i,j}\right) \qquad \forall v_i \rightarrow v_j \qquad (4.54)$$

$$\lceil \mathbf{Start}_i \rceil \leftarrow \min\left(\lceil \mathbf{Start}_i \rceil, \lceil \mathbf{Start}_j \rceil - s_i - d_{i,j}\right) \qquad \forall v_i \rightarrow v_j \qquad (4.55)$$

This step can be implemented in $O(|\overrightarrow{s}|)$. Next, all insertion triplets (v_i, v_j, v_k) are inspected, similarly to line 8 of Algorithm 7. Each triplet can be used to define the earliest and latest arrival at node v_j:

$$ea = \lfloor \mathbf{Start}_i \rfloor + s_i + d_{i,j} \tag{4.56}$$

$$la = \lceil \mathbf{Start}_k \rceil - s_j - d_{jk} \tag{4.57}$$

Triplets corresponding to time window violations are removed:

$$\begin{aligned} &(ea > \lceil \mathbf{Start}_j \rceil) \vee (la < \lfloor \mathbf{Start}_j \rfloor) \vee (ea > la) \\ &\implies \overrightarrow{S}.\text{notBetween}(v_i, v_j, v_k) \end{aligned} \tag{4.58}$$

In (4.58), if either reaching node v_j cannot be done within its time window, or if doing a detour through it would violate the time window of v_k, the insertion is removed. This shares some similarities with the work from [Sav85], where time windows are used for checking the validity of moves in local search. Finally, a time window update is performed for nodes $v_j \in R \setminus \overrightarrow{s}$ that are required but not yet inserted:

$$\lfloor \mathbf{Start}_j \rfloor \leftarrow \max\left(\lfloor \mathbf{Start}_j \rfloor, \min_{v_i \in \overrightarrow{S}.\text{getInsert}(v_j)} \lfloor \mathbf{Start}_i \rfloor + s_i + d_{i,j} \right) \tag{4.59}$$

$$\lceil \mathbf{Start}_j \rceil \leftarrow \min\left(\lceil \mathbf{Start}_j \rceil, \max_{v_i \in \overrightarrow{S}.\text{getInsert}(v_j), v_i \rightarrow v_k} \lceil \mathbf{Start}_k \rceil - s_j - d_{j,k} \right) \tag{4.60}$$

The visit time of such nodes are updated based on their earliest predecessor and latest successor in (4.59), (4.60) respectively. The time complexity of the filtering is the same as in the Distance constraint: $O(|E|)$. Although we reason over a set of required nodes as in [TKS20], we do not ensure that a valid transition exists among all required nodes: this problem is NP-complete and would be too computationally expensive to perform at every filtering.

4.3.3 Precedence

For some applications, visiting a set of nodes in a specific order is important, such as the visit of a pickup that must be done before visiting the corresponding drop-off. The Precedence constraint can be used in such scenarios, ensuring that an ordered set of nodes $\overrightarrow{o}$ appears in the same order in a sequence variable ($\overrightarrow{o}$ being a fixed sequence, not a variable). It is formally defined as

$$\text{Precedence}(\overrightarrow{S}, \overrightarrow{o}) \leftrightarrow \forall v_i \overset{\overrightarrow{o}}{\prec} v_j : v_i, v_j \in \overrightarrow{S} \implies v_i \overset{\overrightarrow{S}}{\prec} v_j \tag{4.61}$$

Note that some or all nodes in $\overrightarrow{o}$ may be absent from the sequence variable. If the nodes from the set $\overrightarrow{o}$ must be all present or all absent, one can easily enforce this with $O(|\overrightarrow{o}|^2)$ equality constraints: $\forall v_i, v_j \in \overrightarrow{o} : \mathcal{R}_i(\overrightarrow{S}) = \mathcal{R}_j(\overrightarrow{S})$.

Filtering The filtering consists of two main steps.

1. First, it considers the partial sequence $\overrightarrow{s}$ and ensures that nodes currently belonging to $\overrightarrow{o} \cap \overrightarrow{s}$ appears in the same order in both $\overrightarrow{o}$ and $\overrightarrow{s}$. This step can be implemented in $O(\max(|\overrightarrow{o}|, |\overrightarrow{s}|))$ and results in a failure if the ordering in $\overrightarrow{s}$ violates the one from $\overrightarrow{o}$.

2. Next, it considers the insertable nodes from the ordering to respect $(\overrightarrow{o})$, removing detours edges that would violate the ordering from $\overrightarrow{o}$ if used for insertions. This is described in Algorithm 8. A set Q tracks the nodes whose detours will be filtered. The main loop iterates over each node $v_k \in \overrightarrow{o}$, as well as the end of the sequence ω. If v_k is insertable (*i.e.*, it is not yet in $\overrightarrow{s}$), it is added to Q for potential detour filtering (line 5). On the contrary, if v_k is already in $\overrightarrow{s}$, it serves as a boundary for the nodes in Q. For each node v_j in Q, we enforce that it can only be inserted between v_i and v_k, where:

   - v_i is the previous node in $\overrightarrow{o}$ that also belongs to $\overrightarrow{s}$, found in a previous iteration.
   - v_k (currently being iterated over) is the next node after v_j in $\overrightarrow{o}$ that belongs to $\overrightarrow{S}$.

   To enforce this consistency, we remove all detour edges for v_j that would insert it before v_i or after v_k(line 7), preserving only detour edges consistent with the required precedence.

Algorithm 8: Precedence$(\overrightarrow{S}, \overrightarrow{o})$ constraint filtering for invalid detours.

```
1  Q ← ∅
2  v_i ← α
3  for v_k ∈ o⃗ · ω do
4      if S⃗.isInsertable(v_k) then
5          Q ← Q ∪ {v_k}
6      else if S⃗.isMember(v_k) then
7          for v_j ∈ Q do
               // v_j can only be inserted between v_i and v_k
8              S⃗.notBetween(α, v_j, v_i)
9              S⃗.notBetween(v_k, v_j, ω)
10         Q ← ∅
11         v_i ← v_k
```

The time complexity of the filtering is dominated by Algorithm 8, running in $O(|\overrightarrow{o}| \cdot |V|)$.

Example 4.3.1. An example of the filtering is illustrated in Figure 4.7. The current partial sequence is $\overrightarrow{S} = \alpha \cdot v_1 \cdot v_3 \cdot v_5 \cdot \omega$. The ordering to enforce is $\overrightarrow{o} = v_2 \cdot v_3 \cdot v_4$. The first step of the filtering, checking the ordering, does not trigger any failure. Algorithm 8 is then used for the second step. The four iterations done at line 3 are as follows:

1. $v_k = v_2$. Given that $v_2 \in I$, Q becomes $\{v_2\}$.
2. $v_k = v_3$, which belong to $\overrightarrow{s}$. The nodes in the queue Q must be placed between v_i and v_k (here forcing v_2 to be placed between α and v_3). This is done by two calls: $\overrightarrow{S}$.notBetween(α, v_2, α) (doing nothing) and $\overrightarrow{S}$.notBetween(v_3, v_2, ω). Finally, v_i becomes v_3 and Q is emptied.
3. $v_k = v_4$. Given that $v_4 \in I$, Q becomes $\{v_4\}$.
4. $v_k = \omega$, enforcing nodes in Q to be placed between v_3 and ω. The two calls at lines 8, 9 are $\overrightarrow{S}$.notBetween(α, v_4, v_3) and $\overrightarrow{S}$.notBetween(ω, v_4, ω) (the latter doing nothing).

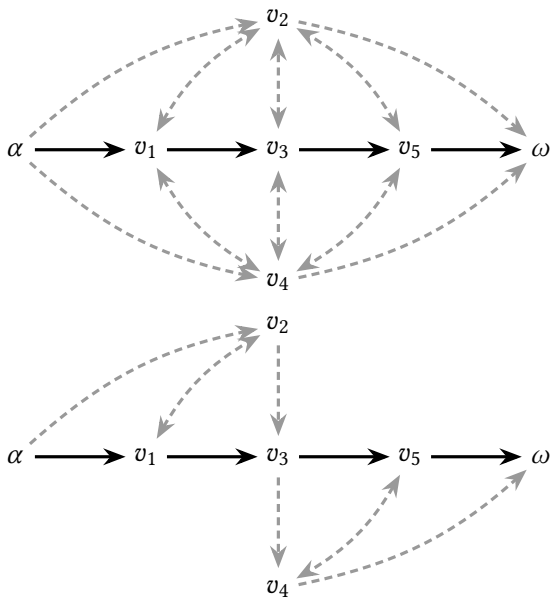


Figure 4.7: Precedence constraint with $\overrightarrow{o} = v_2 \cdot v_3 \cdot v_4$, before filtering (left) and after filtering (right). Edges (v_2, v_4) and (v_4, v_2) are present but not drawn for clarity.

4.3.4 Cumulative

Some variations of VRPs involve pickup and deliveries, transporting goods or people. The Cumulative constraint can be used to represent those scenarios. It ensures that going through all pickups and deliveries visited in a sequence respects an assigned capacity.

More specifically, let us define an *activity* i as a pair of nodes $(\boldsymbol{s}_i, \boldsymbol{e}_i)$ corresponding to its start (pickup) and end (delivery), respectively. The set of all activities is written A. An activity $i \in A$ consumes a certain load $\boldsymbol{l}_i$ during its execution and can be in one of three states with respect to the current sequence $\overrightarrow{S}$: *fully inserted* if $\boldsymbol{s}_i \in S \wedge \boldsymbol{e}_i \in S$, *non-inserted* if $\boldsymbol{s}_i \notin S \wedge \boldsymbol{e}_i \notin S$, and *partially inserted* otherwise (the start or the end is inserted but not both). The Cumulative constraint with a maximum capacity c, with starts $\boldsymbol{s}$, corresponding ends $\boldsymbol{e}$ and loads $\boldsymbol{l}$ is defined as:

$$\text{Cumulative}(\overrightarrow{S}, \boldsymbol{s}, \boldsymbol{e}, \boldsymbol{l}, c) \leftrightarrow \begin{cases} \left(\forall v \in \overrightarrow{S} : \sum_{i \in A | \boldsymbol{s}_i \leq v < \boldsymbol{e}_i} \boldsymbol{l}_i \leq c\right) \wedge \\ \left(\forall i \in A : \boldsymbol{s}_i \in \overrightarrow{S} \iff \boldsymbol{e}_i \in \overrightarrow{S}\right) \wedge \\ \left(\forall i \in A : \text{Precedence}(\overrightarrow{S}, (\boldsymbol{s}_i, \boldsymbol{e}_i))\right) \end{cases} \tag{4.62}$$

This constraint implies that the start $\boldsymbol{s}_i$ of an activity $i \in A$ is visited before its end $\boldsymbol{s}_i$ ($\forall i \in A : \text{Precedence}(\overrightarrow{S}, (\boldsymbol{s}_i, \boldsymbol{e}_i))$), and that its nodes are either both present or both absent ($\forall i \in A : \mathcal{R}_{\boldsymbol{s}_i}(\overrightarrow{S}) = \mathcal{R}_{\boldsymbol{e}_i}(\overrightarrow{S})$). An example of sequence over which the constraint holds is presented in Figure 4.8.

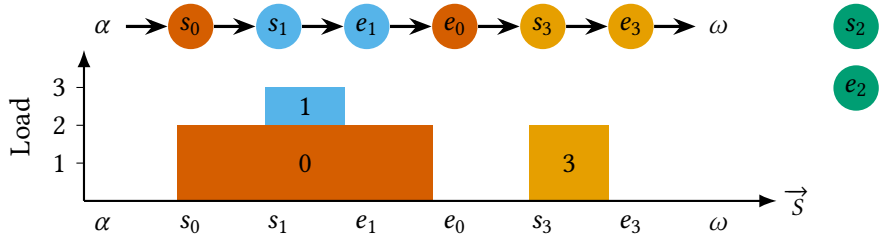


Figure 4.8: Cumulative$(\overrightarrow{S}, (s_0, s_1, s_2, s_3), (e_0, e_1, e_2, e_3), (2, 1, 1, 2), 3)$ **over a fixed sequence variable** $\overrightarrow{S} = \alpha \cdot s_0 \cdot s_1 \cdot e_1 \cdot e_0 \cdot s_3 \cdot e_3 \cdot \omega$**. The sequence is shown at the top, and the graph below shows the accumulated load over the nodes in the sequence. Nodes** s_2, e_2 **from activity 2 are not part of the sequence.**

Filtering Firstly, to ensure that the start $\boldsymbol{s}_i$ and the end $\boldsymbol{e}_i$ of an activity $i \in A$ are visited together and in a valid order, two constraints are added per request. Constraint $\mathcal{R}_{\boldsymbol{s}_i}(Sq) = \mathcal{R}_{\boldsymbol{e}_i}(Sq)$ ensures that the two nodes $\boldsymbol{s}_i, \boldsymbol{e}_i$ appear together, and Precedence$(Sq, (\boldsymbol{s}_i \cdot \boldsymbol{e}_i))$ ensures that the start $\boldsymbol{s}_i$ appears before the end $\boldsymbol{e}_i$. The remaining filtering consists of 3 steps: computing a

load profile, filtering the partially inserted activities and then the non-inserted activities.

A lower bound on the load profile is computed based on the activities that are fully and partially inserted, representing the sum of resources being consumed at each node in the sequence. It is described by three values showing the accumulated load of each inserted node $v \in \overrightarrow{s}$:

- $\boldsymbol{l}_v^-$ for the accumulated load before the visit of v ;
- $\boldsymbol{l}_v^+$ for the accumulated load at the visit of v ;
- $\boldsymbol{l}_v^{v+1}$ for the accumulated load between the visit of v and its successor.

A load profile example is shown on Figure 4.9. Using three values per node to represent the load profile may seem odd. One may think that it is not necessary to represent the load $\boldsymbol{l}_v^{v+1}$ between the visit of node v and its successor, and that only the load when entering ($\boldsymbol{l}_v^-$) and leaving ($\boldsymbol{l}_v^+$) node v are relevant. However, those values are needed to accurately represent whether activities may be put between a node and its successor, as shown in Figure 4.10.

For each fully inserted activity i, the load of i is added between the nodes: $\forall \boldsymbol{s}_i \prec v \preceq \boldsymbol{e}_i : \boldsymbol{l}_v^- \leftarrow \boldsymbol{l}_v^- + \boldsymbol{l}_i, \forall \boldsymbol{s}_i \preceq v \prec \boldsymbol{e}_i : \boldsymbol{l}_v^+ \leftarrow \boldsymbol{l}_v^+ + \boldsymbol{l}_i \wedge \boldsymbol{l}_v^{v+1} \leftarrow \boldsymbol{l}_v^{v+1} + \boldsymbol{l}_i$. When considering only the inserted activities, it follows that $\forall v \in \overrightarrow{s} : \boldsymbol{l}_v^+ = \boldsymbol{l}_v^{v+1}$. Those equalities do not hold anymore after considering the partially inserted activities.

For partially inserted activities with a start inserted, a node from the partial sequence $\overrightarrow{s}$ is used instead of the non-inserted end node to compute the load. It corresponds to the earliest node after the start node or the start node itself, after which the non-inserted end can be inserted. Partially inserted activities with the end inserted behave similarly, considering the latest node preceding the end, after which the start node can be inserted.

Example 4.3.2. On Figure 4.9, activity 0 is partially inserted. The earliest predecessor for e_0 is s_0, which only contributes to the load $\boldsymbol{l}_{s_0}^{s_0+1}$. For the partially inserted activity 1, the earliest predecessor of e_1 is e_2, contributing to $\boldsymbol{l}_{s_1}^+, \boldsymbol{l}_{s_1}^{s_1+1}, \boldsymbol{l}_{e_2}^-, \boldsymbol{l}_{e_2}^+$ (but not $\boldsymbol{l}_{e_2}^{e_2+1}$).

Setting an entry in $\boldsymbol{l}^-, \boldsymbol{l}^+$ or $\boldsymbol{l}^{+1}$ exceeding the capacity triggers a failure.

Given the load profile, the filtering then removes invalid detours for the partially inserted activities and the non-inserted activities. Algorithm 9 depicts the filtering for every non-inserted activity $i \in A$ whose start $\boldsymbol{s}_i$ is inserted but not its corresponding end $\boldsymbol{e}_i$. It first finds the closest node v after which the end $\boldsymbol{e}_i$ can be inserted (line 3), triggering a failure if no such node exists (line 6). Note that this closest node v was already used to set the load of

activity i during the load profile computation, and therefore is already marked as valid (line 7). The next nodes to inspect are thus the nodes following v, in order. As soon as the capacity occurring at a node v does not allow inserting the end $\boldsymbol{e}_i$ of the activity, all detours between this invalid node v and the end ω of the sequence are removed (line 9). A similar filtering is performed in a mirror fashion, considering the non-inserted activities whose ends are inserted but not the corresponding start.

Finally, the filtering removes invalid edges for the non-inserted activities. It uses the filtering introduced in [TKS20], that inspects every start (and end) of activities, and checks if a matching end (and start) can be found, removing detours when no match exists.

Algorithm 9: Filtering of the Cumulative($\overrightarrow{S}, \boldsymbol{s}, \boldsymbol{e}, \boldsymbol{l}, c$) constraint for partially inserted activities with start inserted.

Input: $\overrightarrow{S}$: sequence variable, $\boldsymbol{s}, \boldsymbol{e}, \boldsymbol{l}$: start, end and load of activities, c: capacity.

```
1  for i ∈ A s.t. (S.isMember(s_i) and not S.isMember(e_i)) do
2      v ← s_i
3      while not S.canInsert(v, e_i) do
4          v ← S.getNext(v)
5          if v = ω then
6              return failure
7      v ← S.getNext(v)
8      while v ≠ ω do
9          if max(l_v^-, l_v^+) + l_i > c then
10             S.notBetween(v, e_i, ω)
11             break
12         v ← S.getNext(v)
```

4.3.5 SubSequence

The SubSequence constraint links two sequence variables $\overrightarrow{S}_m, \overrightarrow{S}_s$, ensuring that a subsequence $\overrightarrow{S}_s$ is contained within a super sequence (or master sequence) $\overrightarrow{S}_m$.

$$\begin{aligned}\text{SubSequence}(\overrightarrow{S}_m, \overrightarrow{S}_s) \leftrightarrow &\left(v_i \in \overrightarrow{S}_s \implies v_i \in \overrightarrow{S}_m\right) \wedge \\ &\left(\forall v_i, v_j \in \overrightarrow{S}_s : v_i \overset{\overrightarrow{S}_s}{\prec} v_j \implies v_i \overset{\overrightarrow{S}_m}{\prec} v_j\right)\end{aligned} \tag{4.63}$$

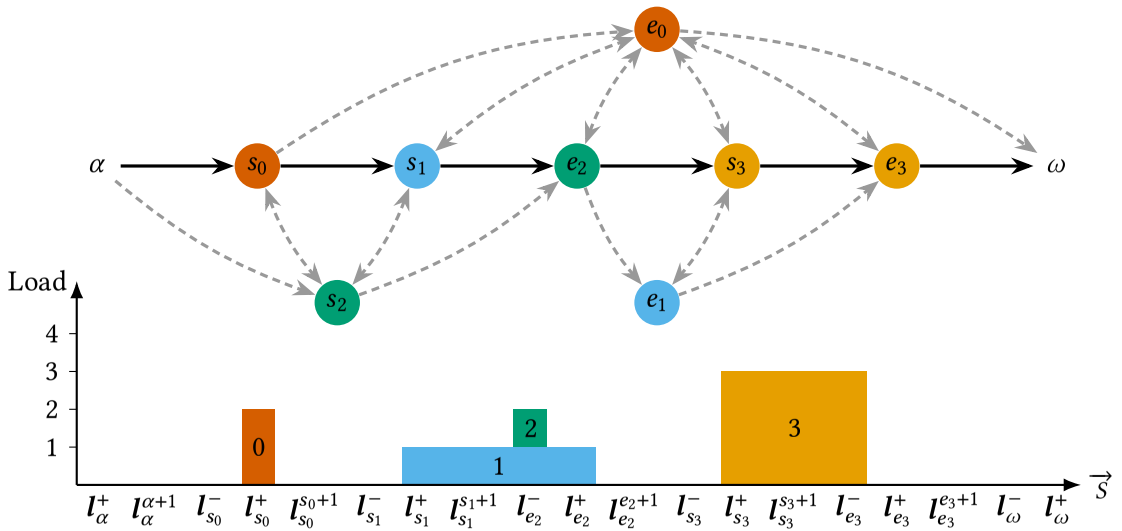


Figure 4.9: A sequence variable Sq with its partial sequence being $\vec{s} = \alpha \cdot s_0 \cdot s_1 \cdot e_2 \cdot s_3 \cdot e_3 \cdot \omega$ (top) and its corresponding load profile (bottom) when using a constraint $\text{Cumulative}(Sq, (s_0, s_1, s_2, s_3), (e_0, e_1, e_2, e_3), (2, 1, 1, 3), 4)$**. Edges between insertable nodes are not represented to improve readability. For the partially inserted activity 1, the closest node after which e_1 can be inserted is e_2, which is why its corresponding rectangle ends at $l^+_{e_2}$ instead of $l^+_{s_1}$ on the load profile.**

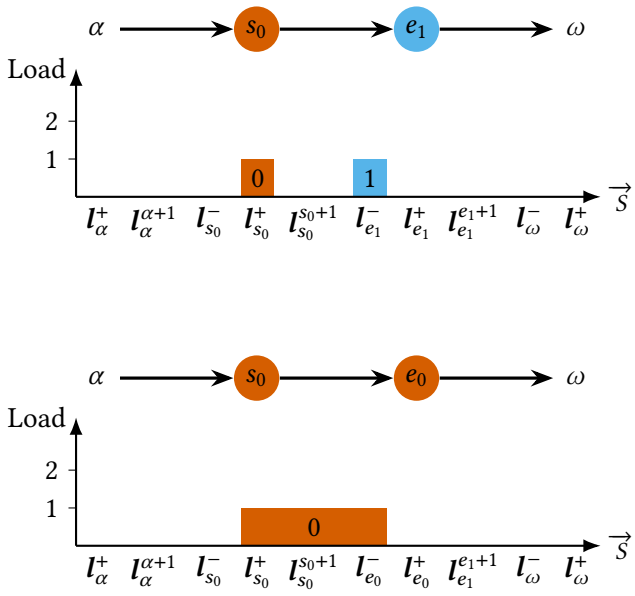


Figure 4.10: Two load profiles for different sequences, assuming a capacity of 2. At the top, with $l_{s_0}^{s_0+1} = 0$, we can detect that an activity of load 2 can be put between s_0 and e_1. At the bottom, with $l_{s_0}^{s_0+1} = 1$, we can detect that no activity of load 2 can be put between s_0 and e_0.

Filtering Firstly, to ensure that every node v_i within the subsequence $\overrightarrow{S}_s$ is visited by the master sequence $\overrightarrow{S}_m$, implications constraints are used:

$$\mathcal{R}_i(\overrightarrow{S}_s) \implies \mathcal{R}_i(\overrightarrow{S}_m) \quad \forall v_i \in V \tag{4.64}$$

Where V is the set of nodes over which the subsequence $\overrightarrow{S}_s$ is defined.

Apart from those implications, the filtering is essentially the same as the one performed by the Precedence constraint from Section 4.3.3. The partial sequences $\overrightarrow{s}_m, \overrightarrow{s}_s$ of the two sequences variables $\overrightarrow{S}_m, \overrightarrow{S}_s$ are retrieved, and the filtering ensures that nodes in common to the two partial sequences appear in the same order:

$$\forall v_i, v_j \in \overrightarrow{s}_s \cap \overrightarrow{s}_m : v_i \overset{\overrightarrow{s}_s}{\prec} v_j \iff v_i \overset{\overrightarrow{s}_m}{\prec} v_j \tag{4.65}$$

If (4.65) does not hold, a failure is triggered. Then, filtering of detour edges is performed by calling twice Algorithm 8 from the Precedence constraint. It is called once with $\overrightarrow{S} = \overrightarrow{S}_m$ and $\overrightarrow{o} = \overrightarrow{s}_s$, enforcing precedences from the partial subsequence $\overrightarrow{s}_s$ onto the master sequence $\overrightarrow{S}_m$, and a second time with $\overrightarrow{S} = \overrightarrow{S}_s$ and $\overrightarrow{o} = \overrightarrow{s}_m$, enforcing precedences from the partial master sequence $\overrightarrow{s}_m$ onto the subsequence $\overrightarrow{S}_s$.

4.4 Search

This section presents some basic search procedures and principles used in conjunction with one or more sequence variables to explore the search space.

4.4.1 Branching

Filtering of the constraints is generally not enough to terminate with fixed sequences. A search procedure is needed to explore the search space. When working with sequence variables, this corresponds to iteratively choosing an unfixed sequence and applying alternative decisions further constraining its domain, through the use of domain derivation. Once all sequences in the problem are fixed and the constraints are satisfied, a solution to the problem has been found.

As shown in the top part of Figure 4.11, different intermediate sequences can ultimately lead to the same one through different insertion steps. This symmetry, induced by branching decisions, can cause inefficiencies. Ideally, we would explore search trees corresponding to disjoint search spaces.

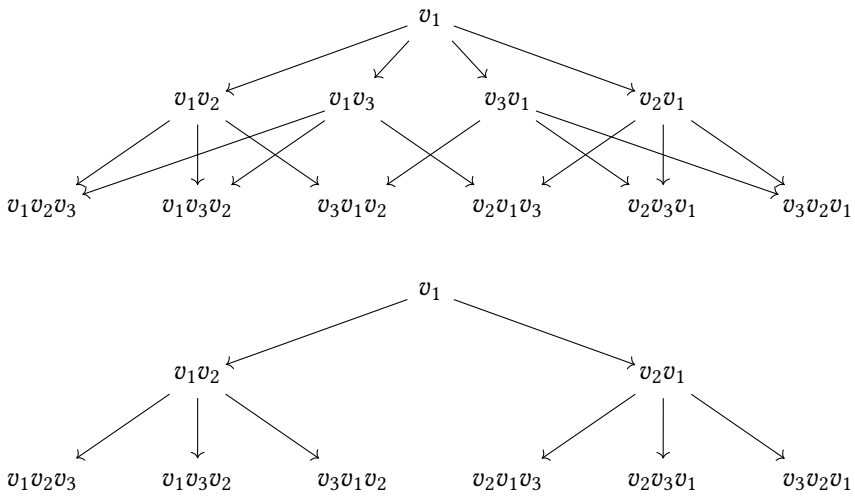


Figure 4.11: Sequences created from a fully connected graph $G(V, E)$ with $V = \{\alpha, v_1, v_2, v_3, \omega\}$, where all nodes are required ($V = R$). Nodes α and ω are implicit and not shown. Top: each sequence is extended by inserting any feasible node at each step. Bottom: a node is first selected for insertion, then all its feasible positions are considered for insertion before moving to the next node (first v_1, then v_2, then v_3).

Disjoint search spaces A simple branching strategy that guarantees the generation of disjoint search spaces is the two-step n-ary branching:

1. **Node selection**: Choose an insertable node $v_i \in I$ within a sequence variable Sq.

2. **Node branching**: For each possible insertion point for v_i in S, create a branch inserting v_i at that position.

The first step is similar to the variable selection used with integer variables. It allows the integration of first-fail strategies, such as selecting nodes with the fewest possible insertion points. The second step is conceptually similar to value selection; therefore, the most promising insertions should be attempted first. It allows the integration of insertion-based heuristics, such as selecting the insertion that results in the smallest increase in tour length. This is best suited when all nodes must be inserted in a sequence in a solution.

As can be observed in the bottom part of Figure 4.11, this strategy generates only distinct sequences.

Another simple binary branching strategy that offers the same guarantees is to replace the second step (node branching) with only two branches. An insertion point is selected, with the insertion performed on the left branch, while the corresponding notBetween derivation is enforced on the right branch.

4.4.2 Large Neighborhood Search

The usage of LNS with sequence variables was already presented in Algorithm 4. In the context of VRPs, this algorithm consists of relaxing some tours by removing nodes from sequences. This approach maintains partial tours, similar in spirit to the partial-order scheduling method introduced for scheduling in [GLN05]. It also accurately corresponds to the original LNS presented in [Sha98], where partial tours are extended through insertions.

The reconstruction phase uses CP and its search capabilities to reinsert the removed nodes into the restricted problem, possibly within a time limit, before restarting the process.

The set of nodes to relax can be selected in various ways, such as randomly or with more advanced strategies based on relatedness criteria, such as geographical proximity or time-based considerations between nodes, as in [BV04; CV20; Sha98].

4.5 Related work: Previous Insertion Sequence Variables

The preceding sections presented insertion-based sequence variables, their domain, global constraints and considerations on the search. Now that those elements have been properly introduced, some previous work may be described in depth.

Previous work has already proposed to introduce insertion-based sequence variables in constraint programming. The very first attempt was proposed by Thomas, Kameugne, and Schaus in [TKS20]. It was further described in the thesis of Thomas [Tho23]. Later on, I continued this work and proposed a second variation in [DSV22]. The differences between sequence variables in this thesis and with those previous versions are highlighted next.

4.5.1 First Iteration: The Basis

The first version of insertion-based sequence variables in CP was presented in [TKS20; Tho23]. Some notable differences with the version presented in this thesis are the following.

No outgoing edges : instead of maintaining, for each node, both its ingoing and outgoing edges, the first iteration only maintained the ingoing edges. Insertions were performed by using one directed edge (v_1, v_2), with $v_1 \in \overrightarrow{s}$ and $v_2 \in (V \setminus \overrightarrow{s} \setminus X)$ instead of detour edges. Furthermore, nodes within the partial sequence had only one ingoing edge: their immediate predecessor. While this reduces memory consumption (the number of sparse sets for the edges is halved), it prevents the implementation of some search heuristics, given the limitations in edge representations (it was for instance harder to choose to insert after a node $v \in \overrightarrow{s}$ having the fewest outgoing edges).

Removal of arbitrary edges : edge deletion in the sequence variable presented so far can only be achieved by removing detour edges. This previous version allowed to remove edges between insertable nodes. This may seems worthwhile to allow (there should not be any reason to keep an edge (v_i, v_j) if no sequence with $(v_i \prec v_j)$ is a solution) but was error-prone. Indeed, removing an edge (v_i, v_j) was not sufficient to forbid sequences where $v_i \prec v_j$: it may still be possible to insert first v_i at the beginning of the sequence and later on insert v_j near the end of the sequence. The user could thus be misled into thinking that edges encoded precedences, and implement faulty filtering algorithms.

Furthermore, although some edges may never appear in a solution (assume a constraint c that forbid edge (v_1, v_2) to appear in a sequence), they may be needed in a partial sequence to reach another sequence being a solution ($\alpha \cdot v_1 \cdot v_2 \cdot \omega$ is inconsistent w.r.t. c, but this partial sequence is needed to derive $\alpha \cdot v_1 \cdot v_3 \cdot v_2 \cdot \omega$). When implementing a filtering propagation for c, a user may think that removing edge (v_1, v_2) makes sense. However, it should actually be kept, as it allows for deriving sequences where $v_1 \prec v_2 \wedge \neg(v_1 \rightarrow v_2)$.

Non-persistent NotBetween : Currently, an insertion triplet (v_1, v_2, v_3) removal through a notBetween prevents from creating a sequence where $v_1 \prec v_2 \prec v_3$. However this previous version did not have this persisting effect: removing an edge (v_1, v_2) (which is equivalent to removing insertion $(v_1, v_2, \text{getNext}(v_1))$ in the current version) does not forbid to create a sequence including the related subsequence. This is essentially because no outgoing edges were encoded in the graph, which prevented the implementation of filtering rules such as (4.23) in the domain, which is instrumental for keeping encoding forbidden subsequences in a graph.

Indirect insertion : an interesting feature in the version proposed in this thesis is highlighted in (4.29): a node $v \notin \overrightarrow{s}$ outside the partial sequence without any feasible direct insertion cannot be part of the sequence. This first version did not ensure this, and a node $v \notin \overrightarrow{s}$ without any direct insertion at a given state of the domain may be inserted at a later point.

NP-Completeness . This first iteration also included required nodes. However, checking if a sequence visiting all required nodes could be obtained (*i.e.* checking the consistency of the domain) was NP-Complete according to [TKS20; Tho23], due to arbitrary edge removal and the need to ensure that nodes may still be connected.

No automated insertion : the current version automatically inserts required nodes that have only one insertion remaining. However, due to both indirect insertions and non-persistent NotBetween in the previous iteration, the order in which nodes were inserted mattered: some sequences could only be constructed by a particular list of insertions. One example could be that, starting from $\overrightarrow{s} = \alpha \cdot \omega$, the sequence $\overrightarrow{s}' = \alpha \cdot v_1 \cdot v_2\omega$ may only be obtained by inserting first v_2 and then v_1, and not by inserting v_1 and then v_2. This is equivalent to removing some transitions from the top of Figure 4.11, while still having all leaf nodes reachable from the root node of the search tree. This made automated insertions impractical, as this particular list of insertions was not trivial to compute on large sequences.

Stronger filtering : some filtering algorithms for the constraints enforced a stronger consistency than the *relaxed-insert consistency*. This was for instance the case on the TransitionTimes constraint, which attempted to ensure that a sequence linking all required insertable nodes could be constructed. But the time complexity needed by this heavier filtering hindered the performance on some problems.

No boolean variables for required nodes : as the name suggest, there were no boolean variables telling if a node is required. While this could have been easily implemented in this first iteration (the spars set acting as a tripartition [Sai+13], and used to create the boolean variables, was already present), its absence forced to implement custom constraints for simple logical operation, such as forcing a set D of nodes to be visited together. This corresponds to the relation $\forall v_i, v_j \in D : \mathcal{R}_{v_i}(\overrightarrow{S}) = \mathcal{R}_{v_j}(\overrightarrow{S})$ and is enforced with existing sum constraints over boolean variables in our case. However, it was enforced through a Dependence constraint, specific to sequence variables.

Clique of non-insertable nodes : in some cases, a partial sequence may not be extended further (*i.e.* no insertion remains), yet some edges outside the partial sequence subsisted (connecting nodes that were both outside of the partial sequence but not excluded). In those cases, the sequence domain was not considered as fixed due to those edges. This situation was problematic depending on the problem at hand. If the problem is required to visit all nodes, this situation corresponds to an inconsistency, and backtracking should occur. In other problems where visits may be omitted, it instead corresponds to a solution. Those situations were detected and handled by custom search procedures, which complicated the implementation of search heuristics.

4.5.2 Second Iteration: Lighter Implementation

The first version introduced a significant complexity due to the required nodes and the heavy filtering proposed. A second iteration introduced the following changes compared to the first iteration. It was published in A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables for Routing Problems". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022

No required nodes : which reduces both the expressiveness but also the complexity of the sequence domain. This is similar to working with a *required-relaxed sequence domain* introduced in Definition 4.3.2. Forcing a node to be required in a sequence could still be achieved by adding a constraint that fails whenever the node is excluded. However, nodes put as required with such constraints were not marked in the domain, and could not be retrieved through domain queries.

Lighter filterings : filtering rules proposed in the paper were much simpler as they did not take into account the required nodes. This is equivalent to only consider *relaxed-insert consistency*.

Insertion counters : this second iteration introduced the insertion counters nI_v for every node $v \in V$, absent in the first iteration, which were useful to implement search heuristics efficiently.

Although less expressive, this second iteration performed better than the first on some problems in terms of speed to reach a given solution quality. Notably, the performance on the DARP was better, and new best solutions were found on the classical instances of the Traveling Salesman Problem with Time Windows (TSPTW).

4.5.3 Summary of the Differences

A summary of the key differences between the versions of insertion-based sequence variables is presented in Table 4.6. The third version refers to the one presented in this thesis and to be published in A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables: A Constraint Programming Computational Domain for Routing and Sequencing". Manuscript in preparation. 2025.

Feature	1st version [TKS20; Tho23]	2nd version [DSV22]	3rd version [DSV25]
Required nodes	✓		✓
Required nodes as boolean variables			✓
NP-Complete domain consistency	✓		
Outgoing edges exposed			✓
Arbitrary edge removal	✓	✓	
Persistent notBetween			✓
Automated insertions			✓
Insertion counters		✓	✓

Table 4.6: Major differences between versions of sequence variables.

Some differences in terms of search tree exploration with a sequence applied on $V = \{\alpha, \omega, v_1, v_2\}$ are highlighted in Figure 4.12 (sequences from [TS18; DSV22]) and in Figure 4.13 ([DSV25]). Node expansions in both search trees are obtained through derivations as similar as possible, given that the implementations and operations slightly differ.

More nodes in the search tree are considered with [TS18; DSV22] in Figure 4.12, some of which correspond to the same state. Nodes without any direct insertions (node v_1 in state C) may still be inserted at a later point (state J). Moreover, removing an edge (α, v_1) at the root (the equivalent of a notBetween(α, v_1, ω) in this version) still allows obtaining a sequence with

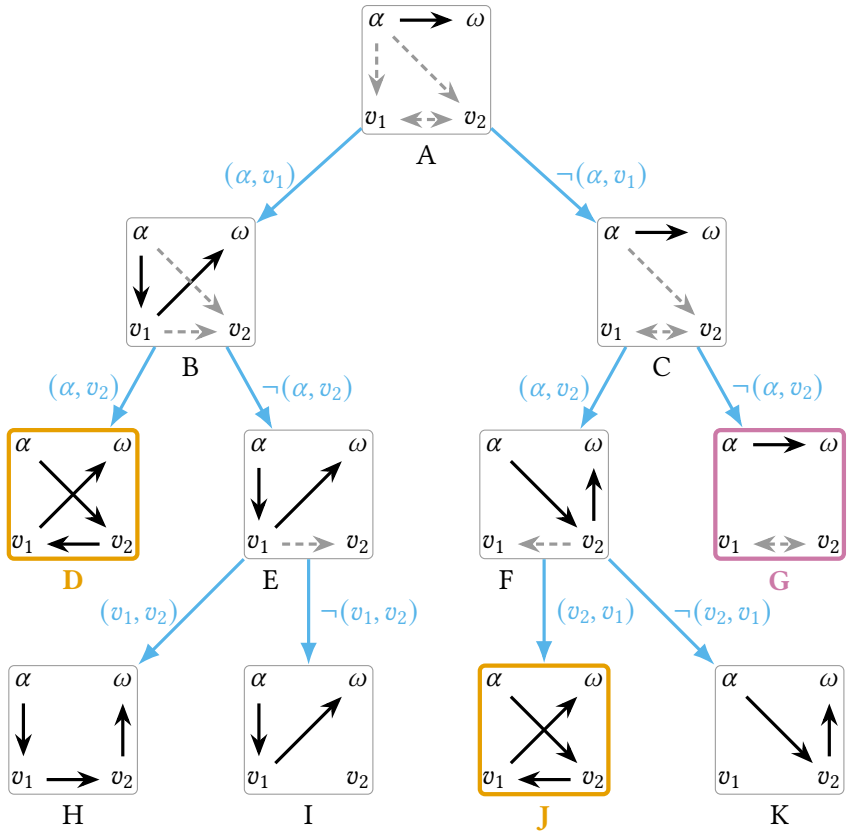


Figure 4.12: Search tree exploration with previous sequence variables [TKS20; DSV22]. (v_i, v_j) **denotes an insertion of** v_j **after** v_i**, and** $\neg(v_i, v_j)$ **the removal of the corresponding insertion. Nodes D and J in the search tree correspond to the same state (and same solution). Node G has no node insertable even though edges outside the partial sequence remain. Finally, in node J** $\alpha \prec v_1$ **even though one of its ancestors (node C) was obtained by** $\neg(\alpha, v_1)$**.**

$\alpha \prec v_1 \prec \omega$. This happens because removing the edge (α, v_1) is not equivalent to the negation of performing an insertion (α, v_1). Finally, a clique of nodes outside the partial path may remain, even though none of the node can be inserted (state G). In contrast, none of those problems is present in Figure 4.13.

The problem of reaching twice the same solution (nodes D and J in Figure 4.12) in [TS18; DSV22] could be mitigated by only performing insertions during the search exploration, instead of performing both insertions and notBetween. This effectively explores each solution once. The drawback is the limitation in search procedure expressiveness.

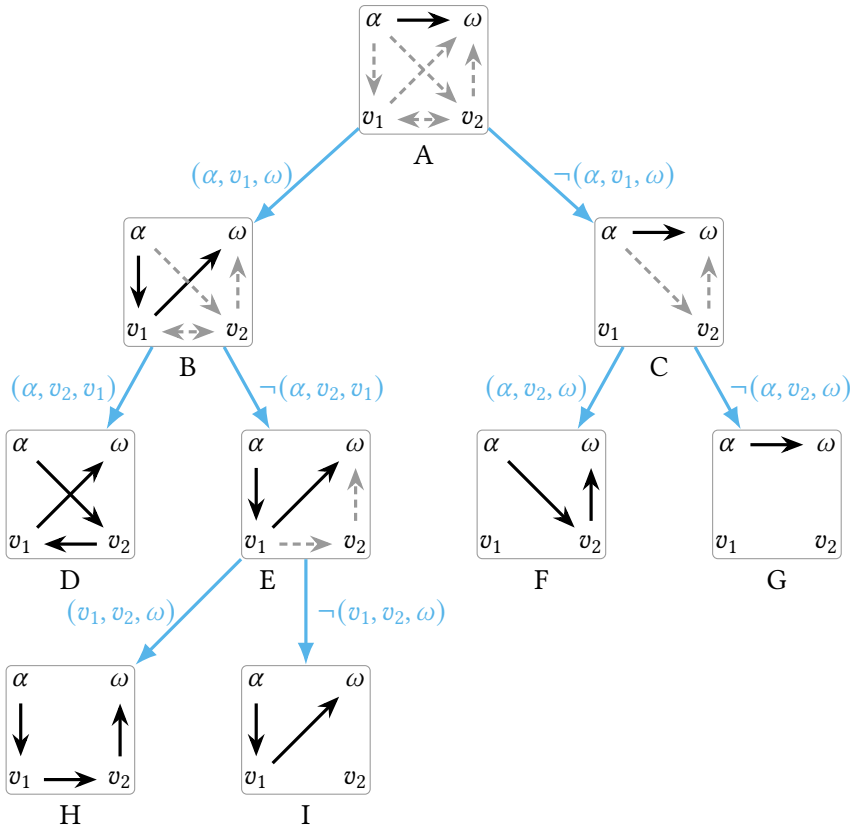


Figure 4.13: Search tree exploration with current sequence variables [DSV25]. (v_i, v_j, v_k) **denotes an** $\text{insert}(v_i, v_j)$ **and** $\neg(v_i, v_j, v_k)$ **a** $\text{notBetween}(v_i, v_j, v_k)$**. Each node in the search tree corresponds to a different state. Node G has no remaining edges between nodes outside the partial sequence. No descendant of node C can be obtained such that** $\alpha \prec v_1$**.**

4.6 Applications

The DARP problem introduced in section 4.1 is first tackled. Then, the Patient Transportation Problem [Cap+18] is presented, followed by the Traveling Salesman Problem with Time Windows. In all used instances, distances and travel times are the same. Finally, a non-routing problem is also solved with sequence variables.

All experiments were conducted using two Intel(R) Xeon(R) CPU E5-2687W in single-threaded mode or a similar machine. The implementation was done in Java using the MaxiCP solver [Sch+24], an extension from MiniCP [MSV21]. Experiments were run in parallel using GNU Parallel [Tan21].

4.6.1 Dial-A-Ride Problem

The state-of-the-art for solving the DARP (introduced in section 4.1), to the best of our knowledge, is the Adaptive LNS proposed by [GD19], combining the operators proposed by [RP06] (random, worst and Shaw removal with greedy and regret based insertions) with additional relaxations, exploiting 9 removal and 5 insertion operators in total. Insertions are evaluated based on a feasibility check specific to the DARP, and obtained solutions close to the best found so far are further optimized using an adaptation of the neighborhood proposed in [BS01]. For further readings on the DARP, the reader is referred to the literature review from [Ho+18].

4.6.1.1 LNS

We use a simple LNS that always relaxes 10 requests chosen randomly and uses Algorithm 3 for the branching. The request to insert is the one having the minimum number of insertions, defined as the product of insertions for its pickup and drop. Insertions having a small increase in tour distance and high preserved time slacks are considered first, as in [JV11]. When considering an insertion triplet point (v_i, v_j, v_k) in vehicle $Route_k$, the heuristic cost is defined as:

$$C_1(\boldsymbol{d}_{i,j} + \boldsymbol{d}_{j,k} - \boldsymbol{d}_{i,k}) - C_2(\lceil \boldsymbol{Time}_k \rceil - \lfloor \boldsymbol{Time}_i \rfloor - \boldsymbol{d}_i - \boldsymbol{d}_{i,j} - \boldsymbol{d}_j - \boldsymbol{d}_{j,k}) \quad (4.66)$$

Where constants C_1, C_2 control the importance of the detour cost and the preserved time slack, respectively. The heuristic cost for inserting a request $\boldsymbol{r}_i \in R$ is the sum of cost for inserting its pickup $\boldsymbol{i}^+$ and drop $\boldsymbol{i}^-$. Values for C_1, C_2 were taken from [JV11] and set to 80 and 1, respectively.

In cases where a request has a node already being inserted (for instance a node that was automatically inserted), this request is selected in priority and its remaining node is inserted.

We compare the sequence variable approach with other approaches suitable for modeling VRPs:

- **A successor model** written in Minizinc [Net+07] and run with the Gecode solver [SLT06], which performed best across all Minizinc backends. The underlying CP model is essentially the same as in [BPR11]. Both exhaustive search (**Succ**) and LNS (**Succ-LNS**) are reported.
- **OrTools** and its routing library, which relies on local search [PFa]. All combinations of first solution strategy and local search meta heuristics in the solver have been tried, and we report the best overall configuration.

- **Hexaly**, a commercial solver specialized in routing [24b], using a model provided by the Hexaly team.
- **CP Optimizer (CPO)**, a commercial scheduling solver, whose model is written using head-tail sequence variables [Lab+18a; Lab+18b].

This list voluntary omits the state-of-the-art methods on the DARP, described earlier. Such methods are specialized towards the DARP only, and cannot be directly adapted to cope with other kinds of VRPs, compared to the present list, where the models can be adapted in a declarative fashion. Nevertheless, we do report the best known solutions found by such methods in our experiments.

The results on the instances from [CL03] are described in Table 4.7. Each approach was run 10 times in single-threaded mode for 15 minutes, using different random seed, and the average and minimum gap are reported. The approach using sequence variable, although relying on a simple LNS, is consistently within 15% of the best known solutions. Only the approach using Hexaly and sequence variables manage to find feasible solutions to all instances. This is still far from the state-of-the-art [GD19], but more adaptable to other VRP variants.

4.6.1.2 Exact Search

On small DARP instances, it is possible to explore the entire search space. This experiment attempts to evaluate the search statistics when enumerating all solutions to an instance.

We first describe in Algorithm 10 an alternative to Algorithm 3 for generating branching decisions. Given a request r_i to insert, it generates all relevant insertion points for it. To do so, the pickup $\boldsymbol{r_i^+}$ is first inserted in a vehicle k, and a fixpoint computation is run (lines 6 and 7). If successful, the fixpoint has filtered out insertions for the drop $\boldsymbol{r_i^-}$, which are used to create the final branching points that will be considered, inserting both nodes $\boldsymbol{r_i^+}, \boldsymbol{r_i^-}$ composing the request (lines 9 to 11). All those operations are surrounded by save and restore operations, similarly to BIVS [FP17]. The branching points created in this manner are also sorted by heuristic (4.66). In some cases, inserting the pickup $\boldsymbol{r_i^+}$ is sufficient to force the insertion of the drop $\boldsymbol{r_i^-}$ (for instance when inserting into in an empty route: the pickup and drop will necessarily be consecutive due to precedence constraints), which needs to be considered as a branching point as well (line 13).

We compare the following approaches, with the following abbreviations:

- **Seq22**: the previous version of sequence variables [DSV22], which used Algorithm 10 for its branching (line 13 was absent in this approach, given that no insertion was automatically performed).

Table 4.7: Gap to the best known solution (bks) in percentage on the DARP instances. Best results are highlighted in bold.

				Succ		Succ-LNS		CPO		Ortools		Hexaly		Seqvar	
Instance	$\|K\|$	$\|R\|$	bks	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min
R1a	3	24	190.02	22.34	22.34	5.41	3.30	7.01	4.23	-	-	4.19	0.98	**0.00**	**0.00**
R1b	3	24	164.46	58.38	58.38	6.97	2.58	17.28	13.18	1.82	1.82	4.83	2.02	**0.40**	**0.00**
R7a	4	36	291.71	-	-	14.85	7.86	19.81	4.26	-	-	6.21	3.06	**1.74**	**1.07**
R7b	4	36	248.21	-	-	13.28	2.92	17.30	10.00	-	-	9.69	4.30	**2.64**	**0.00**
R2a	5	48	301.34	-	-	11.91	3.98	-	-	-	-	8.62	3.35	**1.31**	**0.66**
R2b	5	48	295.66	74.41	74.41	11.82	5.54	26.20	14.49	**2.58**	2.58	11.29	7.47	3.39	**2.35**
R8a	6	72	487.84	-	-	-	-	-	-	-	-	20.08	12.10	**8.39**	**6.65**
R8b	6	72	458.73	-	-	-	-	-	-	11.30	11.30	16.79	12.34	**8.47**	**5.94**
R3a	7	72	532.00	-	-	-	-	-	-	-	-	12.22	8.74	**5.77**	**4.51**
R3b	7	72	484.83	-	-	-	-	-	-	9.49	9.38	15.57	8.57	**8.44**	**7.79**
R9a	8	108	653.94	-	-	-	-	-	-	-	-	34.91	29.49	**8.80**	**4.51**
R9b	8	108	592.23	-	-	-	-	-	-	-	-	23.67	19.99	**11.29**	**7.34**
R4a	9	96	570.25	-	-	-	-	-	-	10.65	10.65	21.84	13.97	**9.62**	**8.12**
R4b	9	96	529.33	-	-	-	-	95.72	40.83	14.66	14.66	20.87	17.53	**11.45**	**8.61**
R10a	10	144	845.47	-	-	-	-	-	-	-	-	28.80	22.98	**14.29**	**12.07**
R10b	10	144	783.81	-	-	-	-	-	-	-	-	32.82	25.68	**15.23**	**14.15**
R5a	11	120	625.64	-	-	-	-	-	-	18.87	18.87	20.61	15.80	**12.01**	**8.85**
R5b	11	120	573.56	-	-	-	-	-	-	15.90	15.90	23.37	17.96	**10.33**	**9.00**
R6a	13	144	783.78	-	-	-	-	-	-	18.10	18.10	23.83	19.75	**12.13**	**10.25**
R6b	13	144	725.22	-	-	-	-	-	-	17.96	17.91	23.27	18.51	**11.24**	**8.84**

Algorithm 10: Enhanced creation of the branching points for the DARP.

Input: r_i: request to insert
1 $branches \leftarrow \{\}$
2 **for** $k \in K$ **do**
3 $\quad I^+ \leftarrow \boldsymbol{Route}_k$.getInsert($\boldsymbol{r}_i^+$)
4 $\quad$ **for** $p^+ \in I^+$ **do**
5 $\quad\quad$ saveState()
6 $\quad\quad \boldsymbol{Route}_k$.insert($p^+, \boldsymbol{r}_i^+$)
7 $\quad\quad$ success $\leftarrow$ fixpoint()
8 $\quad\quad$ **if** success **then**
9 $\quad\quad\quad I^- \leftarrow \boldsymbol{Route}_k$.getInsert($\boldsymbol{r}_i^-$)
10 $\quad\quad\quad$ **for** $p^- \in I^-$ **do**
11 $\quad\quad\quad\quad branches \leftarrow branches \cup \left\{(\boldsymbol{Route}_k.\text{insert}(p^+, \boldsymbol{r}_i^+) \wedge \boldsymbol{Route}_k.\text{insert}(p^-, \boldsymbol{r}_i^-))\right\}$
12 $\quad\quad\quad$ **if** $I^- = \emptyset$ **then**
$\quad\quad\quad\quad$ // $\boldsymbol{r}_i^-$ was automatically inserted
13 $\quad\quad\quad\quad branches \leftarrow branches \cup \left\{\boldsymbol{Route}_k.\text{insert}(p^+, \boldsymbol{r}_i^+)\right\}$
14 $\quad\quad$ restoreState()
15 sort *branches* by increasing order of heuristic cost
16 **return** *branches*

- **LNS-FFPA**: the search from Jain and Van Hentenryck for the DARP [JV11]. This approach is faster than previous sequence version [DSV22], also based on CP, and uses insertions of requests for extending the paths, without sequence variables. An insertion of a node v_2 between node v_1 and v_3 is enforced by adding constraints on the time windows, in the form $\boldsymbol{Time}_1 + \boldsymbol{s}_1 + \boldsymbol{d}_{1,2} < \boldsymbol{Time}_2$, and by modifying a chain of successors represented with reversible integers instead of integer variables (*i.e.* it does not rely on a successor model with a circuit constraint). Insertions for a request are generated by simulating the insertion of one of the nodes, and computing candidate insertion points for the remaining node. This is similar to Algorithm 10, but the implementation was much more complex due to the absence of sequence variables maintaining insertion points. For a fair comparison, we have implemented the COMET source code provided by the authors of [JV11] in Java. Note that this approach shares some similarities with the approach from [GD19], where insertion points for requests are also generated and evaluated on the fly, given the current partial paths.

- **Thesis**, which uses sequence variables from this thesis. In particular,

insertion of requests are evaluated with both Algorithm 3 and 10.

We also introduce two other search procedures, considering the insertion of a single node instead of a request. The node selected is the one having the fewest insertions, prioritizing nodes being required and not yet inserted. Given this node to insert, two ways of exploring the search space are considered:

- **Node (*n*)** generates *n* branches for the node: one for each insertion point in each vehicle. Branches are explored in increasing order of heuristic (4.66).
- **Node (2)** generates 2 branches for the node: the left branch attempts to insert the node at the best insertion according to (4.66), and the right branch removes such insertion with a corresponding notBetween operation. Note that this search would have generated duplicate solutions with previous sequence variables [TKS20; DSV22], as illustrated previously in Figure 4.12.

Results are reported in Table 4.8 for an instance with 2 vehicles and 20 requests. Several points are worth highlighting. Generating branches with Algorithm 10 instead of Algorithm 3 leads to a smaller search tree and three times fewer failures, but is slower as more fixpoint computations occur. The previous version of sequence variables [DSV22] is dominated in all aspects by this version when using the same search strategy (Algorithm 10). Interestingly enough, fewer failures occur when branching on nodes instead of requests (except when compared with Algorithm 10 and this version of sequences). Generating *n* branches for inserting a node produces less exploration in the search tree compared to the binary branching, and is the fastest approach. That being said, when examining the first solutions obtained by branching on nodes, their objective value was larger than the ones obtained by branching on requests, even guided with heuristic cost (4.66). This is simply explained by the fact that the heuristic is guided by more information (computation on 2 nodes) when considering requests, therefore producing quickly more valuable solutions. Finally, in terms of speed, the new version of sequence variables is now on par with the speed obtained with LNS-FFPA [JV11], even being slightly faster depending on the search strategy, which was not the case with the previous version [DSV22].

4.6.2 Patient Transportation Problem

This problem, introduced in [Cap+18] and studied in depth in [Tho23], is an extension of the Dial-A-Ride problem introduced in Section 4.6.1 with a few additional constraints. It considers the transport of patients to a hospital (described as one activity) and possibly back to a given location (another activity)

Method	Time (s)	Nodes	Failures	Solutions
LNS-FFPA [JV11]	974.545	153 864 380	70 033 356	**66 700 800**
Seq22 [DSV22]	1307.447	120 593 739	35 751 093	**66 700 800**
Algorithm 3	897.255	123 018 976	39 472 347	**66 700 800**
Algorithm 10	1170.054	**96 518 288**	**12 863 981**	**66 700 800**
Node (n)	**885.344**	139 173 098	20 645 264	**66 700 800**
Node (2)	936.781	175 965 306	21 281 854	**66 700 800**

Table 4.8: Statistics for finding all feasible solutions on an instance with 2 vehicles and 20 requests. No LNS was used, and no objective tightening was performed.

by using a limited number of vehicles. The trip to the hospital must therefore always occur before the return trip and some patients can only be transported in a particular type of vehicle (patients in wheelchairs for instance). Time during which vehicles are available are constrained by time windows availability (a vehicle may for instance be available between 08h00 until 12h00, and between 14h00 until 18h00). The objective consists in maximizing the number of transported patients. The problem is illustrated in Figure 4.14, adapted from [Tho23].

4.6.2.1 Model and Search

The model is essentially the same as the DARP (4.1)-(4.7), where one vehicle corresponds to one sequence variable. The modifications to the DARP are as follows. The objective consists instead of maximizing the number of serviced patients, which is retrieved through the boolean variables telling if a node is visited. A patient is considered visited if the first node related to it (*i.e.* its pickup location) is visited by a vehicle. Equality constraints over the visits of nodes ensure that all nodes related to a patient (2 nodes for a simple trip, 4 if a backward trip is also present) are always visited together. For cases where a particular patient p can only be transported in a given type of vehicle t, node exclusion occurs for every node related to p from sequence variables whose related vehicle type is different from t. Lastly, the patient may ask that its return trip is performed by the same vehicle, which is enforced by a precedence constraint over all nodes related to the patient.

Finally, for taking into account the time during which vehicles are available, a modeling trick is used. Two nodes per time window availability of each vehicle are introduced, corresponding to times at which the vehicle must be stationed at its depot: the start and end of each time window availability. Those nodes are directly inserted, in the order of the time windows avail-

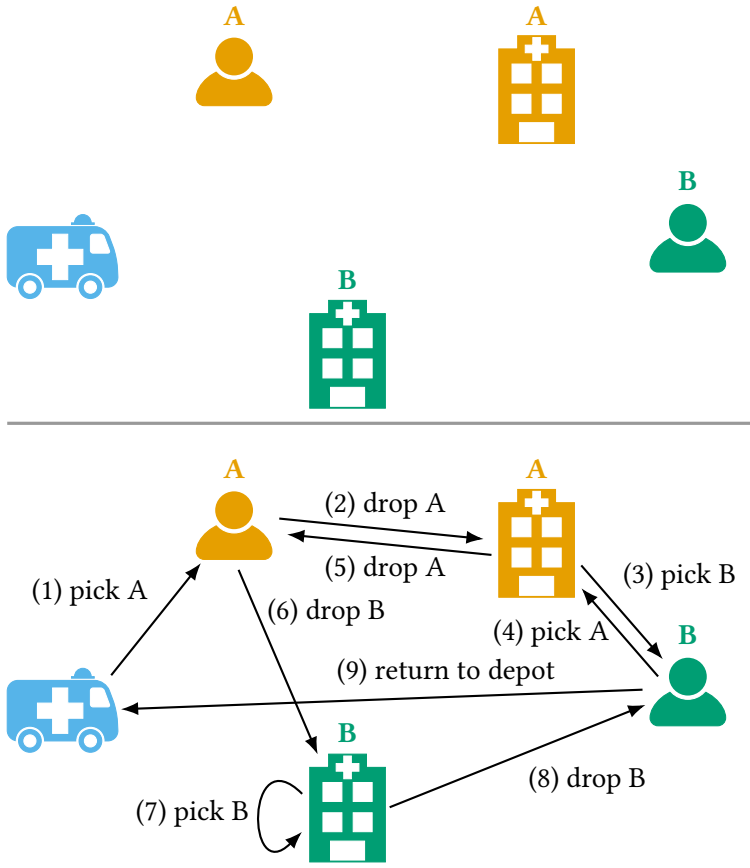


Figure 4.14: Instance for the PTP (top) and one possible solution (bottom). The Number in parentheses indicates the order in which operations are performed. Each patient (A and B) is picked twice and dropped twice: once for its trip to the hospital, and once for its return trip home.

ability, within the sequence variable of their corresponding vehicle. Finally, a "fake" activity with a load equal to the capacity of the vehicle is introduced between the end node of a time window availability and the start node of the subsequent time window. Combined with the cumulative constraint, this "fake" activity ensures that (i) no patient may be visited outside the availability, and that (ii) no patient may stay in the vehicle outside the availability. Regarding the other constraints in the model, those introduced nodes are duplicates of the depot, and their time windows for the TransitionTimes constraint correspond to their time window availability.

Example 4.6.1. Consider a vehicle having two time windows availability: [8h00-12h00] and [14h00-18h00]. Four nodes are introduced: s_0, e_0 correspond

to the start and end of the first time window, and s_1, e_1 to the second (and last) time window. The sequence variable $\overrightarrow{s}$ representing the vehicle is initialized with $\alpha = s_0$ and $\omega = e_1$. Then, the other nodes e_0, s_1 corresponding to the time windows are inserted in order, giving the partial sequence $\overrightarrow{s} = s_0 \cdot e_0 \cdot s_1 \cdot e_1$. An activity for the cumulative constraint is introduced, with start e_0, end s_1 and load being the vehicle capacity. Due to the load of this activity, the cumulative constraint ensures that no node may be visited between $e_0 \cdot s_1$ and that no patient picked between $s_0 \cdot e_0$ may be dropped after s_1. Lastly, for each introduced node, the time windows for the TransitionTimes corresponds to the availability: $\boldsymbol{Time}_{s_0} \in$ [8h00-12h00], $\boldsymbol{Time}_{e_0} \in$ [8h00-12h00], $\boldsymbol{Time}_{s_1} \in$ [14h00-18h00], $\boldsymbol{Time}_{e_1} \in$ [14h00-18h00].

An alternative to this modeling trick would have been to introduce one sequence variable per time window availability, for each vehicle. This corresponds to a vehicle duplication, as originally proposed in [Cap+18]. Compared to the presented modeling trick, this consumes more memory. Moreover, it complicates the integration of constraints related to the return trip: if the patient needs to have a return trip back home by the same vehicle, it implies that the nodes related to the return trip must be required by exactly one other sequence corresponding to the same vehicle. This needs to be captured by several sum constraints. In contrast, the return trip by the same vehicle are handled with only one precedence constraint through this modeling trick.

Regarding the search, it selects the non-inserted patient having the fewest number of insertions, summed over all nodes related to the patient. Patients with required nodes are selected in priority. Then, the node to insert from the patient is the one having the smallest number of insertions. Two branches are generated for this node: one inserting the node at its best insertion place using cost (4.66), and one removing the insertion through a notBetween. LNS is used, whose relaxation selects several patients and removes all nodes related to them.

4.6.2.2 Computational Results

We compare the sequence variable approach with the following

- **SCHED+MSS**: the best model reported in [Cap+18], based on CP.
- **ISEQ+SDS**: the results reported in [Tho23]. This method relies on the first iteration of sequence variables.
- **LIU_CPO**: the model proposed in [LAB18], relying on CP Optimizer and its sequence variables.

Results for those approaches are retrieved from [Tho23]. We use the same experimental setting for our own approach, using a timeout of 10 minutes and reporting the best results across 10 runs. Instances are available on CSPLib [24a].

Table 4.9 presents the results between the sequence variables approach and the other methods. The sequence variable approach clearly outperforms all other methods, and reach the best solutions on all but two instances. The difference in obtained solutions is more pronounced on the largest instances. In fact, for some instances, comparable solutions found by other models in 10 minutes are reached within a few seconds with the proposed method. This is the case on RAND-H-10: a solution of value 84, better than previous best solutions, is already obtained in about 10 seconds, and is further optimized afterward.

It is worth noting that some obtained solutions were better than optimal solutions reported in [Tho23]. However, the solution checker provided on CSPLib [24a] confirmed that those obtained solutions were both feasible and of better objective value than the ones found in [Tho23]. This suggests that their model may have been faulty, and wrongly indicated that optimal solutions were reached.

4.6.3 Traveling Salesman Problem With Time Windows

The Traveling Salesman Problem with Time Windows (TSPTW) is an extension of the TSP which adds time windows to the visit of nodes. Even finding a feasible tour for the TSPTW is NP complete [Sav85]. Therefore, two main settings are considered. The first one considers the feasibility problem, and attempts to find a feasible TSPTW tour as fast as possible. The second one considers the optimization problem, and optimizes an initial solution provided with LNS.

A previous version of sequence variables improved 32 best-known solutions on the classical TSPTW benchmark [Lóp20]. Those results were presented in A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables for Routing Problems". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

After this publication, a model aimed at finding initial solutions to the TSPTW was presented in A. Delecluse, P. Schaus, and P. Van Hentenryck. "SEQUOIA: SEQuence-variable-based Optimization In Action for the Traveling Salesman Problem with Time Windows". In: *Doctoral Program of CP23*. 2023.

Both of those publications were using a previous version of sequence variables [DSV22]. The results presented in this section use the same strategy, but

Instances					LIU	SCHED	ISEQ	Thesis
Set	Name	$\|H\|$	$\|V\|$	$\|R\|$	_CPO	+MSS	+SDS	[DSV25]
Easy	RAND-E-1	4	2	16	**15**	**15**	**15**	**15**
Easy	RAND-E-2	8	4	32	**32**	**32**	**32**	**32**
Easy	RAND-E-3	12	5	48	**28**	**28**	**28**	**28**
Easy	RAND-E-4	16	6	64	**64**	62	**64**	**64**
Easy	RAND-E-5	20	8	80	79	75	**80**	**80**
Easy	RAND-E-6	24	9	96	**96**	94	**96**	**96**
Easy	RAND-E-7	28	10	112	**112**	106	**112**	111
Easy	RAND-E-8	32	12	128	**128**	**128**	**128**	**128**
Easy	RAND-E-9	36	14	144	**144**	142	**144**	**144**
Easy	RAND-E-10	40	16	160	159	157	**160**	**160**
Medium	RAND-M-1	8	2	16	**12**	11	**12**	**12**
Medium	RAND-M-2	16	3	32	**20**	**20**	**20**	**20**
Medium	RAND-M-3	24	4	48	**35**	33	**35**	**35**
Medium	RAND-M-4	32	4	64	41	39	**42**	**42**
Medium	RAND-M-5	40	5	80	**69**	59	67	68
Medium	RAND-M-6	48	5	96	60	51	**61**	**61**
Medium	RAND-M-7	56	6	112	**75**	62	**75**	**75**
Medium	RAND-M-8	64	8	128	96	84	95	**97**
Medium	RAND-M-9	72	8	144	94	82	99	**101**
Medium	RAND-M-10	80	9	160	112	100	117	**120**
Hard	RAND-H-1	16	2	16	**8**	**8**	7	**8**
Hard	RAND-H-2	32	3	32	**19**	**19**	**19**	**19**
Hard	RAND-H-3	48	4	48	34	32	34	**35**
Hard	RAND-H-4	64	4	64	**25**	24	24	**25**
Hard	RAND-H-5	80	5	80	48	45	48	**50**
Hard	RAND-H-6	96	5	96	47	41	45	**48**
Hard	RAND-H-7	112	6	112	41	40	**44**	**44**
Hard	RAND-H-8	128	8	128	86	78	89	**90**
Hard	RAND-H-9	144	8	144	83	75	84	**90**
Hard	RAND-H-10	160	8	160	79	73	83	**89**

Table 4.9: Best solutions obtained with each method on the PTP. $|H|$, $|V|$ and $|R|$ refers to the number of hospitals, vehicles and transportation requests (*i.e.* patients), respectively. Best results are shown in bold.

with the new variables presented in this chapter.

The classical benchmark instances from [Lóp20] are used in the experiments. They are split over six datasets, where the number of nodes in the instances varies up to 232:

AFG includes 50 instances, proposed in [Asc96].

Dumas contains 135 instances, introduced in [Dum+95].

GendreauDumasExtended has 130 instances, presented in [Gen+98].

They were constructed based on the instances from the Dumas dataset [Dum+95].

OhlmannThomas inludes 25 instances, proposed in [OT07].

SolomonPesant contains 27 instances [Pes+98], derived from a vehicle routing problem [Sol87].

SolomonPotvinBengio has 30 instances [PB96], also derived from [Sol87] but different from the ones belonging to the SolomonPesant dataset.

4.6.3.1 Feasibility

We strive to identify a feasible path for the TSPTW. To do so, we use the following model.

$$\max \sum_{v \in V} \mathcal{R}_v(\boldsymbol{Route}) \tag{4.67}$$

subject to:

$$\text{TransitionTimes}(\boldsymbol{Route}, (\boldsymbol{Time}), \boldsymbol{s}, \boldsymbol{d}) \tag{4.68}$$

We convert the satisfaction problem into an optimization problem (4.67) by relaxing the constraint of visiting all nodes, maximizing the number of nodes visited, and satisfying the time window constraints for the ones being visited (4.68). Initially, we employ a greedy approach to construct a tour that attempts to visit as many customers as possible within their time windows, using a regret-based heuristic. If some nodes remain not visited in this first step, we then proceed to the second step, which involves an LNS method. Starting from the partial path obtained in the first step, the LNS aims to maximize the number of visited nodes until a path encompassing every node is established. This proposed approach is described as SEQUOIA - Sequence Variable based optimization in action for the TSPTW.

A regret-based heuristic is employed during the greedy search, continuing with insertions in the sequence until no further additions can be made. This results in a partial sequence of visits, which ideally encompasses a majority of the nodes.

Given a node $v \in V$ to insert, the regret of v is defined as the difference between the two smallest costs c (*i.e.*, the best alternatives according to (4.66)) based on its insertions. The regret is commonly used to guide problem-solving [TC72; PR93]. The regret calculation for our problem is described next.

$$regret(\overrightarrow{S}, v, c) = c(p_2(v), v, \overrightarrow{S}.\text{getNext}(p_2(v))) - c(p_1(v), v, \overrightarrow{S}.\text{getNext}(p_1(v))) \tag{4.69}$$

$$p_1(v) = \underset{p \in \overrightarrow{S}.\text{getInsert}(v)}{\text{argmin}} \quad c(p, v, \overrightarrow{S}.\text{getNext}(p)) \tag{4.70}$$

$$p_2(v) = \underset{p \in \overrightarrow{S}.\text{getInsert}(v), p \neq p_1(v)}{\text{argmin}} \quad c(p, v, \overrightarrow{S}.\text{getNext}(p)) \tag{4.71}$$

If no valid predecessor can be derived from either (4.70) or (4.71), the corresponding term in (4.69) is replaced by a large constant. The regret is used to expand a path, one node at a time, until it can no longer be extended.

Figure 4.15 shows how many nodes are visited after the first step, that is the regret-based greedy search. The smallest percentage of visited nodes 66.6% was found on an instance with 60 nodes. On 69 instances, the greedy search was able to find a feasible path, preventing the need to use an LNS. We can observe that the difficulty of finding a feasible path changes between datasets and cannot be expressed through the number of nodes only.

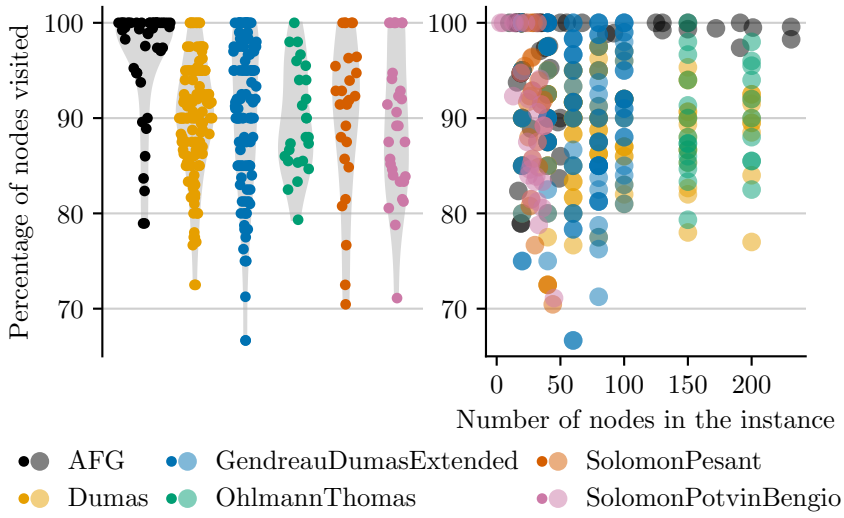


Figure 4.15: Percentage of nodes visited in the path constructed by the greedy search. The left figure shows the observed percentage as a SinaPlot [Sid+18], representing individual observations over the instances as dots, as well as density estimates. The instances datasets are highlighted by colors.

We evaluated SEQUOIA against two local search approaches: VNS [dU10] and ImaxLNS [Pra23], as well as against a dynamic programming approach

- CABS [KB23; Zha98]. Figure 4.16 illustrates the discrepancies in execution time among the techniques, using 100 runs per instance. A timeout was set to 3 minutes. SEQUOIA is dominated by ImaxLNS, the state-of-the-art, and is slower on all instances. It is slightly behind CABS, and offers performance comparable to VNS. Even in instances where our method lags, the average execution time remains under 10 seconds, while the last 2 mentioned techniques can take several minutes or even experience a timeout on some instances.

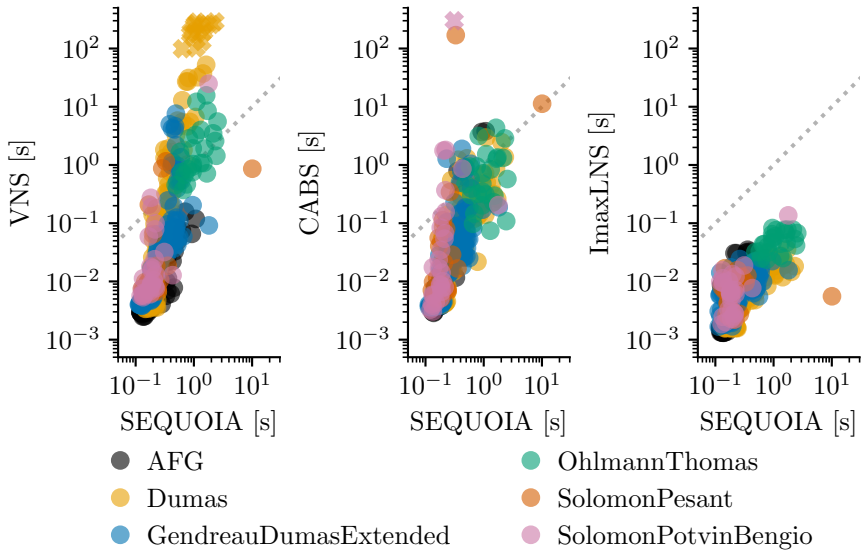


Figure 4.16: Comparison of the execution time between our approach (SEQUOIA) and VNS (left), CABS (middle) and ImaxLNS (right). Each dot corresponds to the mean run time on 100 experiments to find a feasible solution on one given instance. The instances datasets are highlighted by colors. The diagonal indicates that the two approaches need an equal amount of time. A dot above the diagonal means that SEQUOIA was faster. A cross indicates that a least 1 out of the 100 runs resulted in a timeout, assigning the corresponding execution time of the run to the value of the timeout.

4.6.3.2 Optimization

The optimization model is as follows:

$$\min \boldsymbol{Dist} \tag{4.72}$$

subject to:

$$\text{TransitionTimes}(\boldsymbol{Route}, (\boldsymbol{Time}), \boldsymbol{s}, \boldsymbol{d}) \tag{4.73}$$

$$\text{Distance}(\boldsymbol{Route}, \boldsymbol{d}, \boldsymbol{Dist}) \tag{4.74}$$

$$\mathcal{R}_v(\boldsymbol{Route}) = 1 \qquad \forall v \in V \tag{4.75}$$

The objective consists in minimizing the traveled distance (4.72), captured with a Distance constraint (4.74). Visits during time windows are enforced with (4.73), and the visit of every node $v \in V$ is required (4.75).

Interestingly enough, forcing all nodes to be required (4.75) may already extend the initial sequence, by inserting nodes having only one insertion point. Several insertions may be performed automatically with this behavior: a first insertion occurs, then filtering from the initial time windows may leave only one insertion for a node, which also insert it. Even though the sequence variable obtained after those insertions is unfixed, it serves as a strong basis for the problem to solve: all feasible solutions are super-sequence of this first partial sequence. Figure 4.17 reports how many nodes were already visited simply by adding the constraints (4.73) and (4.75). Even on instances with 200 nodes, partial sequence of more than 60 nodes could be identified. We can also observe that the largest initial partial sequences are found on the Dumas dataset, which also has the largest disparity in terms of size for initial partial sequences.

For the optimization problem, we start from initial solutions retrieved by [dU10] and optimize them for 5 minutes using LNS. The relaxation strategy consists in removing several successive nodes.

An important ingredient of the LNS to be successful is the number of nodes to relax. We reuse the scheme proposed in [JV11] for the DARP that consists in gradually augmenting the number of relaxed nodes. The complete LNS is presented in Algorithm 11. It stops when a time limit is reached. In the algorithm, multiple constants are present. The parameters n_{min} and n_{max} dictate the range of neighborhood sizes, commencing with n_{min} and progressively expanding until they reach $n_{max} - \delta$ in the outer loop. The parameter δ regulates the exploration of a variation window starting at i and n_{iter} controls the number of intensification iterations. The values were set to $n_{min} = 5$, $\delta = 5$, $n_{iter} = 3$, $n_{max} = \max(|V|/2 - \delta, |V| - \delta)$. The chosen parameter values have proven to be effective in practice, although a comprehensive set of experiments to identify the optimal values has not been conducted.

The relaxed path is optimized by choosing the node with the smallest number of insertions, and inserting it at all insertion points, evaluated in increasing value of detour cost.

The average gap over time, compared to the best known solutions, are shown in Figure 4.18. Each instance was run 100 times, its average gap over

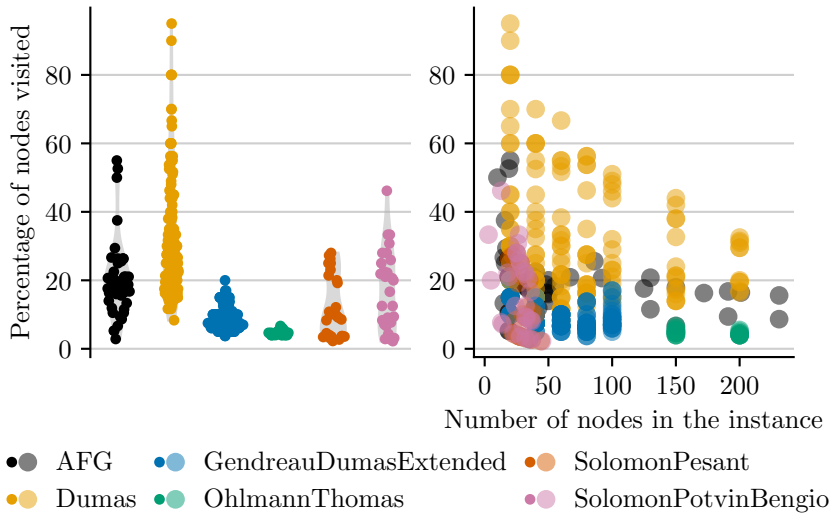


Figure 4.17: Percentage of nodes visited in the partial path simply by adding the optimization constraints (4.73), (4.75) **on the TSPTW. The left figure shows the observed percentage as a SinaPlot [Sid+18], representing individual observations over the instances as dots, as well as density estimates. The instances datasets are highlighted by colors.**

Algorithm 11: LNS

Input : First path *initSol*
Output: Solution *bestSol* minimizing traveled distance

```
1  bestSol ← initSol
2  for i ∈ {n_min ... (n_max − δ)} do
3      if i = n_max − δ then
4          i ← n_min
5      for j ∈ {0 ... (δ − 1)} do
6          for k ∈ {1 ... n_iter} do
7              select a node x not visited in bestSol, randomly.
8              relax i + j consecutive nodes
9              sol ← optimize(relaxed solution)
10             if solution has been improved then
11                 bestSol ← sol
12             if time limit then
13                 return bestSol
14 return bestSol
```

time was computed based on the runs, and results are aggregated per dataset. We observe that the gaps come close to zero in a less than 10 seconds, except on instances from the OhlmannThomas dataset which may take a few minutes to optimize.

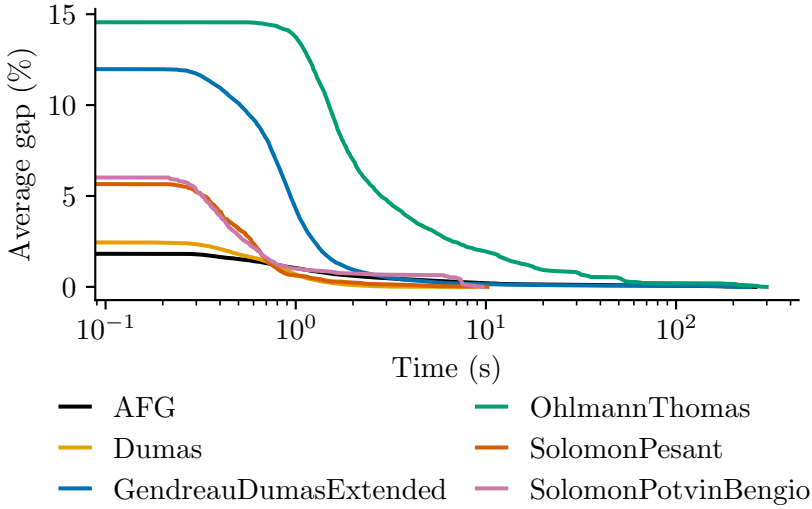


Figure 4.18: Average gap over time compared to the best known solutions for the TSPTW.

A few instances actually had to be discarded in this optimization experiment. Indeed, not all instances from [Lóp20] respect the triangular inequality in their distance matrix. On some of them, the Distance constraint removed some insertions detected as too costly, and ended up discarding all feasible solutions to the instances. However, even when the triangular inequality does not hold, not all feasible solutions are necessarily removed. 38 instances over the 397 used had to be discarded, even though the majority of instances do not respect the triangular inequality.

This problem was absent in the feasibility experiment, as it originates from the Distance constraint, not present in the feasibility model[3]. Ideally, on instances where the triangular inequality is absent and one wishes to use the Distance and TransitionTimes constraints, all-pairs shortest paths algorithms should be performed on the distance matrix, to retrieve a matrix upholding the triangular inequality. The reader is referred to [Cor+22] for a more complete overview of this class of algorithms.

[3]The TransitionTimes constraint also requires the triangular inequality. But in the optimization experiment, the filtering performed by the Distance constraint removed all insertions for nodes due to too costly detours, whereas the TransitionTimes constraint kept those insertions.

4.6.4 Prize-Collecting Sequencing Problem

This problem is not a VRP, but is included to show how sequence variables may be relevant for scheduling problems as well.

The Prize-Collecting Job Sequencing with One Common and Multiple Secondary Resources (PC-JSOCMSR) is a scheduling problem whose objective is to maximize the collected reward from executed tasks, subject to multiple constraints. It was introduced in [HRB18], motivated by the scheduling of patients requiring particle therapy as a treatment for cancer. The problem consists of a set of n tasks $t_i \in T$, each with a processing duration d_i and an associated reward z_i if executed. Each task requires a global resource as well as one specific secondary resource $j \in R$, where R denotes the set of available secondary resources. The set of tasks associated with the secondary resource j is denoted $T(j)$. Both the global resource and the secondary resources can only handle one task at a time: no overlapping of tasks is allowed. Additionally, the execution of a task t_i is further constrained on its secondary resource: it includes a pre-processing duration d_i^{pre} and a post-processing duration d_i^{post} during which the secondary resource is unavailable. The execution of a task t_i must be non-preemptive and must occur within one of its ω_i specified time windows $w_i = \{w_{ik} \mid k = 0, \ldots, \omega_i - 1\}$, where $w_{ik} = [w_{ik}^{start}, w_{ik}^{end}]$. The goal is to maximize the sum of the rewards of the executed tasks.

To simplify the naming, the problem will be abbreviated to the Prize Collecting Scheduling Problem (PCSP) instead of PC-JSOCMSR. An example of PCSP solution is presented in Figure 4.19.

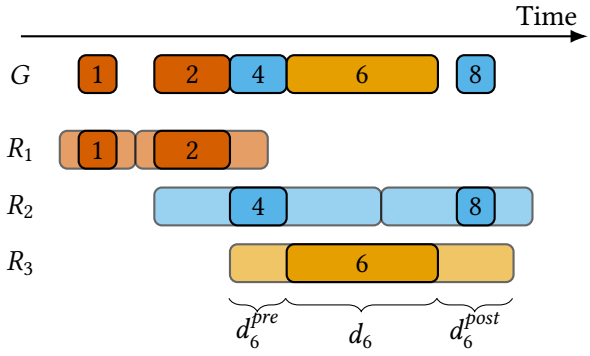


Figure 4.19: PCSP solution

The best published approaches for this problem rely on Mixed Integer Linear Programming for exact search, and local search for heuristic approaches on the largest instances [FS23]. Some other methods, using dynamic programming, decision diagram or CP have also been developed in [Hor+21].

Model Our model for the PCSP is as follows:

$$\min \sum_{t_i \in T} z_i \cdot \mathcal{R}_{t_i}(\overrightarrow{G}) \tag{4.76}$$

subject to:

$$\text{TransitionTimes}(\overrightarrow{G}, (\boldsymbol{Time}), \boldsymbol{d}, \boldsymbol{O}) \tag{4.77}$$

$$\text{TransitionTimes}(\overrightarrow{R}_j, (\boldsymbol{Time}), \boldsymbol{d}, \boldsymbol{Tr}) \qquad \forall j \in R \tag{4.78}$$

$$\text{SubSequence}(\overrightarrow{G}, \overrightarrow{R}_j) \qquad \forall j \in R \tag{4.79}$$

$$\mathcal{R}_{t_i}(\overrightarrow{R}_j) = \mathcal{R}_{t_i}(\overrightarrow{G}) \qquad \forall j \in R\ \forall t_i \in T(j) \tag{4.80}$$

$$\mathcal{R}_{t_i}(\overrightarrow{R}_j) = 0 \qquad \forall j \in R\ \forall t_i \in (T \setminus T(j)) \tag{4.81}$$

One sequence variable is introduced per resource: $\overrightarrow{G}$ for the global resource and $\overrightarrow{R}_j$ for each secondary resource $j \in R$. Their start and end correspond to dummy nodes. The goal consists in maximizing the sum of rewards, summed over the executed tasks (4.76). Visits within time windows are enforced through TransitionTimes constraints, both on the global (4.77) and on the secondary (4.78) sequences. The time window $\boldsymbol{Time}_i$ of a task t_i is defined based on its initial time windows w_i. Regarding transition times, they are zero on the global resource (4.77), captured through a matrix $O \in \mathbb{Z}^{T \times T}$ whose entries are zero. On the secondary resource, transition times between tasks are encoded in a matrix $\boldsymbol{Tr} \in \mathbb{Z}^{T \times T}$, whose entries are set to $\boldsymbol{Tr}_{i,i'} = d_i^{post} + d_{i'}^{pre}$. To ensure coherence between the global resource and the secondary ones, a SubSequence constraint is used (4.79). Executing a task on the global resource means that it must be executed on its secondary resource as well (4.80). Finally, tasks cannot be executed on a secondary resource not related to them (4.81).

Search The search selects a node to insert and generates branches for inserting it or excluding it. The selection of the node depends on the current state of the sequence $\overrightarrow{G}$ corresponding to the global resource. If the sequence $\overrightarrow{G}$ contains nodes being both insertable and required, one of those nodes (the one with the fewest insertion points) is selected. Otherwise, insertable nodes from the secondary resource are considered, selecting the node with the largest reward.

Given the n insertions points for a node v, $n+1$ branching decisions are created. n branches attempt to insert the node at every feasible insertion. The remaining branch attempts to exclude the node. Branches are sorted decreasingly according to a heuristic reward. The reward to insert node v after its predecessor $p \in \overrightarrow{S}$.getInsert() is the number of successors available for p: $|\overrightarrow{S}.\text{getEdgesFrom}(p)|$. The reward for excluding the node is instead

set to a constant x. This heuristic favors insertions after nodes having many successors, which helps to preserve some time slack in the partial sequence for further insertions.

LNS is used and relaxes successive nodes from the main sequence $\overrightarrow{G}$. Nodes being relaxed are also removed from their secondary sequence $\overrightarrow{R}_j$. The LNS scheme is the same as in Algorithm 11, with $n_{min} = 10$, $\delta = 5$, $n_{iter} = 10$, $n_{max} = |T|$ and a limit of 5000 failures in the search tree exploration. x was arbitrarily set to 3 in the experiments.

4.6.4.1 Computational Results

Table 4.10 presents the comparison between the sequence approach (Seqvar-LNS) and ILS, using the reported results by the authors [FS23]. Each instance was run 10 times using different random seeds, a timeout of 15 minutes in our case, and we report the worst, average and best gap observed. Two datasets retrieved from [FS23] are compared. The approach with sequence variables is behind the dedicated ILS approach. Still, average gaps are within 5% of the best known solutions and are found relatively quickly.

4.7 Discussion

Some limitations of sequence variables are first presented, before discussing future work worth considering.

4.7.1 Limitations

Forbid consecutive nodes One simple operation with the successor model, the removal of an invalid direct successor, is harder to perform with sequence variables. This is due to the fact that removals only occur through deletions of detour edges, and that some inconsistent links in a partial sequence may be needed to derive an extended partial sequence, without edges violating the constraints. This was already discussed in section 4.5.1, on the removal of arbitrary edges.

Memory consumption On a problem with n nodes and k vehicles, the space complexity is $O(kn^2)$. This is notably higher than successor models, whose space complexity is typically in the order of $O(n^2)$. This was sufficient for all problems presented in this thesis, involving dozens of vehicles and hundreds of nodes, but additional decomposition techniques may be needed to upscale to very large instances.

		Seqvar-LNS				ILS			
		Gap [%]			Time [s]	Gap [%]			Time [s]
Set	$\|T\|$	Worst	Mean	Best		Worst	Mean	Best	
A_E	50	1.16	0.90	0.67	64.75	**0.01**	**0.00**	**0.00**	21.70
	100	2.29	1.76	1.31	75.54	**0.00**	**0.00**	**0.00**	67.60
	150	2.72	2.27	1.82	73.19	**0.01**	**0.01**	**0.00**	116.36
	200	3.42	2.99	2.51	105.06	**0.03**	**0.01**	**0.00**	161.82
	250	2.78	2.50	2.21	85.72	**0.01**	**0.00**	**0.00**	208.79
	300	3.67	3.09	2.52	117.94	**0.02**	**0.01**	**0.00**	254.50
	350	2.98	2.64	2.44	120.91	**0.01**	**0.01**	**0.00**	294.06
	400	2.98	2.53	2.22	140.32	**0.03**	**0.01**	**0.00**	337.25
	450	2.65	2.32	1.97	103.81	**0.02**	**0.01**	**0.00**	379.37
	500	3.00	2.71	2.42	112.77	**0.06**	**0.02**	**0.01**	420.74
	All	2.76	2.37	2.01	100.00	**0.02**	**0.01**	**0.00**	226.22
T	50	1.53	1.43	1.26	16.91	**0.00**	**0.00**	**0.00**	16.19
	100	3.49	2.44	1.59	155.88	**0.27**	**0.18**	**0.00**	59.80
	150	5.30	4.10	2.98	123.82	**0.76**	**0.48**	**0.13**	128.88
	200	5.69	4.64	3.61	145.12	**1.25**	**0.93**	**0.47**	235.41
	250	5.09	4.14	3.28	215.75	**0.95**	**0.70**	**0.29**	366.72
	300	5.02	4.05	3.15	309.65	**0.61**	**0.36**	**0.01**	517.49
	350	5.24	4.34	3.40	417.21	**0.55**	**0.33**	**0.00**	676.91
	400	5.75	4.81	3.87	516.72	**0.49**	**0.29**	**0.00**	859.58
	450	5.76	4.86	3.97	608.63	**0.47**	**0.28**	**0.00**	1065.05
	500	6.25	5.38	4.50	670.55	**0.48**	**0.28**	**0.00**	1282.29
	All	4.91	4.02	3.16	318.02	**0.58**	**0.38**	**0.09**	520.83

Table 4.10: Comparison between ILS and sequence variables on the PCSP. The gap between obtained solutions and best known solutions are given in percentages. The time is given in seconds and corresponds to the average time after which no subsequent solutions were found.

Forbidden subsequences only defined over the partial sequence As presented in section 4.2.1, a forbidden subsequence (v_1, v_2, v_3) may only be added if both its endpoints (v_1 and v_3) are within the partial sequence. This means that some forbidden subsequences may only be added at a given state of the domain, but it is the price to pay to have a reasonable memory consumption.

4.7.2 Future Work

4.7.2.1 Stronger Filtering

The filtering algorithms presented in Section 4.3 are relatively simple. Some of them could easily be enhanced by adding some reasoning commonly found in dedicated approaches.

For instance, the computation of the lower bound of the Distance constraint in Algorithm 7 is based on the current partial sequence, and does not

take into account that some nodes may be required. An estimation of the distance if such nodes were included could be obtained through a minimum spanning tree computation (or even better, through a 1-tree computation). Such computation could produce better lower bounds.

Similarly, the TransitionTimes filtering could benefit from existing algorithms on scheduling with optional tasks. For instance, it would be worth investigating the incorporation of the algorithms presented in Vilim's thesis [Vil07], where filtering for disjunctive constraints with optional tasks are presented. Those algorithms can be modified to take into account transition times between elements, as shown in [Van+16; DVS15]. Efficiently incorporating these algorithms in the TransitionTimes filtering, while scaling to hundreds or thousands of nodes, remains to be studied.

In any case, one should remember that some of the increase in performance between the first [TS18] and second iteration [DSV22] of sequence variables were due to lighter filtering introduced. Therefore, introducing heavy computation within the filtering algorithms should be considered with caution, and be empirically validated.

4.7.2.2 Reducing Memory Consumption

Memory consumption may be a significant issue on problems with many nodes and sequence variables. Here is a non-exhaustive list of changes that may be considered to tackle large problems.

- Storing the edges between nodes may be implemented by using a reversible sparse bit-set, similar to the one used in [Dem+16], compared to using a sparse set as in [Sai+13]. Edge deletion would occur by setting a corresponding bit to false in the sparse bit-set. This would lower the memory representation at first (fewer array entries are needed) but stores more reversible values on the trail (each word within the bit-set is a reversible value, compared to only one reversible integer in [Sai+13]).

- A significant way of heavily reducing memory consumption with several sequence variables would be to use an approach similar to CP Optimizer for its own sequence variables [Lab+18b; Lab+18a]. Nodes in the problem would be created outside any sequence variable, and correspond to a *node variable*. Then, a sequence variable would be created by giving as input the set of node variables on which the sequence is applied. Edges for each node would be *shared* across all sequence variables related to the node, requiring only two sparse sets per node (ingoing and outgoing edges). This means that deleting an edge (v_i, v_j) would remove this edge in all sequences. The insertion of a node v_i in

a sequence variable would exclude v_i from other sequence variables on which it is applied.

Assuming a problem with n nodes and k sequence variables, the space complexity would be lowered from $O(kn^2)$ to $O(n(n+k)) \approx O(n^2)$ if $k < n$. The drawback is that nodes may be inserted in only one sequence variable. This limits the application on problems such as the PCSP from section 4.6.4, but is still applicable to other problems such as the DARP.

4.7.2.3 Domain Delta

One additional line of research to improve the performance of sequence variables would be to introduce a domain delta. A domain delta allows retrieving, within the filtering of a constraint, the domain modifications that were performed since the last call to the filtering. For instance, if a constraint c filters a sequence variable domain, and then one insertion (v_1, v_2, v_3) happens, the delta would be able to tell that only (v_1, v_2, v_3) has modified the domain since the last call to c.

Domain deltas are available in solvers such as MaxiCP and OscaR [Sch+24; Tea12]. They are reminiscent of the AC5 algorithm introduced to implement the fixpoint computations, where constraints are notified of which values have been removed since their last call [VDT92]. An Implementation of deltas within the Gecode solver is discussed in [LS07], along with some incremental filtering, such as for the AllDifferent constraint [Rég94]. The following discussion presents how they could be used with sequence variables, although they do not exist at the time of writing this thesis. We assume that a delta over a sequence $\overrightarrow{S}$ provides the two following functions:

- $\overrightarrow{S}$.delta.getInsertions() retrieves the newest insertions that have occurred, in the form of insertion triplets.
- $\overrightarrow{S}$.delta.getEdgesInSequence() retrieves the newest edges added onto a sequence due to insertions, in the form of directed edges. This may be computed based on the preceding delta function.

One constraint where the benefits of deltas can be illustrated is the Distance constraint. A filtering using delta is shown in Algorithm 12. Lines in gray are identical to the original one (Algorithm 7). Some reversible integers now need to be introduced, to maintain information across filtering calls. One reversible integer *previousLength* tracks the length of the partial sequence. Similarly, one reversible integer $previousMaxCost_j$ for very node $v_j \in V$ tracks the largest insertion cost for inserting node v_j over filtering calls.

Firstly, the travel length of the partial sequence is computed, by summing to the previous length the cost related to the newest insertions that have occurred (lines 1 to 3). Then, filtering of insertion triplets for each insertable node v_j occurs. Each new edge (v_i, v_k) that has been added onto the sequence is inspected, to see whether the insertion (v_i, v_j, v_k) is feasible, removing insertions that would exceed the maximum length if performed (lines 21 to 24). This is similar to the initial filtering, except that only newly defined insertions are examined instead of all insertions. However, if the largest insertion cost $previousMaxCost_j$ for a node v_j now exceeds the largest allowed distance increase, the past insertions must be examined as well, as they may also exceed the largest distance allowed. In this case, the incremental filtering is skipped and the same filtering as in Algorithm 7 is performed (lines 13 to 17).

Although the worst-case time complexity is the same, the best-case time complexity is lowered, by examining only what portion of the sequence has changed since the last call to the filtering.

Even though Algorithm 12 showcases how deltas could be used for filtering of constraints, their implementation remains to be designed in a future work. One should also decide the level of information that deltas may provide: are we only interested in the insertions that have occurred, as in Algorithm 12, or should the filtering also be aware of the detour edges deletions? Finally, not all filtering algorithms may profit from updates based on deltas, and the memory requirements may be increased due to the information tracked over calls (for instance the reversible integers *previousMaxCost* in Algorithm 12).

4.8 Conclusion

This chapter enhances the previous proposal of sequence variables, which are used in CP to tackle vehicle routing and sequencing problems. Their domain representation, compact implementation, and interactions with boolean variables are proposed and formalized.

These variables are compatible with optional visits, insertion-based heuristics, and can be easily combined with Large Neighborhood Search. Through their usage, complex VRPs such as the DARP or the PTP can be solved in CP while staying close to the state-of-the-art in terms of performance. Common search strategies exploiting the minimum number of insertions performed well across VRP variants, showing a strong reusability of search heuristics. We believe such variables are practical for solving VRPs in a CP framework.

Algorithm 12: Distance($\overrightarrow{S}, \boldsymbol{d}, Dist$) constraint filtering with delta.

```
1  length ← previousLength.value()
2  for (v_i, v_j, v_k) ∈ S⃗.delta.getInsertions() do
3  |  length ← length + d_{i,j} + d_{j,k} − d_{i,k}
4  if S⃗.isFixed() then
5  |  Dist ← length
6  else
7  |  ⌊Dist⌋ ← max(length, ⌊Dist⌋)
8  |  maxDetour ← ⌈Dist⌉ − length
9  |  for v_j ∈ S⃗.getInsertable() do
10 |  |  maxCost_j ← previousMaxCost_j.value()
11 |  |  if maxCost_j > maxDetour then
12 |  |  |  maxCost_j ← −∞
13 |  |  |  for v_i ∈ S⃗.getInsert(v_j) do
14 |  |  |  |  v_k ← S⃗.getNext(v_i)
15 |  |  |  |  cost ← d_{i,j} + d_{j,k} − d_{i,k}
16 |  |  |  |  if cost > maxDetour then
17 |  |  |  |  |  S⃗.notBetween(v_i, v_j, v_k)
18 |  |  |  |  else
19 |  |  |  |  |  maxCost_j ← max (maxCost_j, cost)
20 |  |  else
21 |  |  |  for (v_i, v_k) ∈ S⃗.delta.getEdgesInSequence() do
22 |  |  |  |  cost ← d_{i,j} + d_{j,k} − d_{i,k}
23 |  |  |  |  if cost > maxDetour then
24 |  |  |  |  |  S⃗.notBetween(v_i, v_j, v_k)
25 |  |  |  |  else
26 |  |  |  |  |  maxCost_j ← max (maxCost_j, cost)
27 |  |  previousMaxCost_j.value() ← maxCost_j
28 previousLength.value() ← length
```

Conclusion

5

The goal of this thesis was to push further the performance of CP when used on VRPs. Two main lines of research have been studied.

In Chapter 3, black-box value selection heuristics well suited for VRPs were presented, keeping existing CP models while enhancing their search strategies. They automate the nearest neighbor selection on the TSP, and prove to be valuable on other problems as well.

Chapter 4 studied a new kind of variable specifically designed for routing and sequencing problems: sequence variables. While previous work laid the foundations for those variables [TKS20; Tho23], this thesis went further, by formalizing their domain, enhancing their implementation and constraints, and applying them on more routing problems. With similar insertion-based search across VRP variants, relatively good performance was obtained, demonstrating a strong reusability. During the development of those variables, new best found solutions were found for the TSPTW, and solutions obtained on other problems such as the DARP were improved and are relatively close to the best known.

5.1 Perspectives

In Chapter 3 and 4, some future research directions have already been highlighted. We now discuss additional work that seems interesting to investigate.

Value Heuristics The value heuristics presented in Chapter 3 are designed for optimization problems. It would be valuable to try to apply them on satisfaction problems as well. Some discussions in Section 3.4 already present ways to generalize them to such problems, for instance by maximizing the search space to explore compared to minimizing the impact on the objective.

The restricted fixpoint method presented relies on a subset of the constraints to quickly estimate the impact on the objective. We proposed to use constraints on the shortest paths between a variable and the objective within the constraint network. However, if the goal is to find such an estimate as fast as possible, other sets of constraints may be more valuable to use. For instance, one could try to compute the shortest path by considering that the

edges in the constraint network are weighted according to the average running time of the constraint on which the edge is attached. With this method, constraints on the shortest (weighted) path(s) would be expected to quickly reach the (restricted) fixpoint, providing a faster estimate than the original method. However, this is only targeted at improving the running time to reach the fixpoint, and experiments should be conducted to see if the filtering still provides enough information to guide BIVS+RF and RLA+RF.

Sequence Variables Additional research directions have already been discussed in Section 4.7. Some aim at improving the constraint filtering, referencing algorithms worth considering and describing how incremental filtering could be performed. Some other suggestions were also presented to lower the memory consumption, by adapting some data structures.

A class of VRP that would be interesting to tackle with sequence variables are online transportation problems (also called dynamic vehicle routing problems), where transportation requests are revealed in real time [KPS98]. Given that sequence variables are designed for insertion-based search, it would be worthwhile to solve such problems with them, by inserting into the current vehicle path an additional transportation request added into a system. In particular, work such as [BJM19] explores the routing of taxis in New York, and part of their solution uses insertion-based strategies. Attempting to solve their problem with sequence variables would not only assess their application for online problems, but also force the development of scaling techniques for sequence variables, given the huge set of transportation requests involved (more than 500,000 trips every day [BJM19; 25]). Other similar online problems such as [Zha+23; BV03a] and the book from Van Hentenryck and Bent, [VB06], are valuable resources to help conduct such future research.

The filtering from constraints dealing with distance matrices (namely the Distance and TransitionTimes constraints) require the triangular inequality to hold. Although this is the case on the majority of problems tackled in this thesis, it may be interesting to create variants of those filtering algorithms that handle cases where it does not hold.

Finally, the constraints involving distances matrices are defined using a fixed transition matrix. However, the transitions between elements may instead change over time in some problems, for instance by increasing transition times during certain time windows, to account for traffic congestion. It would be worthwhile to see if some of the techniques presented in [Pra23], tackling the time-dependent TSPTW, can be applied with sequence variables, given that part of their approach relies on insertion-based search. Moreover, there exists CP approaches with the sequence variables from CP Optimizer [Lab+18b] on time-dependent VRPs [MLS15], extending scheduling constraints to take into account time-dependent transitions.

Bibliography

[21] *6.3. Scheduling in or-tools — or-tools User's Manual*. [Online; accessed 14. Apr. 2025]. June 2021. URL: https://acrogenesis.com/or-tools/documentation/user_manual/manual/ls/scheduling_or_tools.html.

[24a] *082: Patient Transportation Problem*. [Online; accessed 20. Apr. 2025]. Oct. 2024. URL: https://www.csplib.org/Problems/prob082/data.

[24b] *Hexaly*. [Online; accessed 15. Jan. 2025]. Oct. 2024. URL: https://www.hexaly.com.

[24c] *QAPLIB-Problem Instances and Solutions – COR@L*. [Online; accessed 5. Apr. 2024]. Apr. 2024. URL: https://coral.ise.lehigh.edu/data-sets/qaplib/qaplib-problem-instances-and-solutions.

[25] *TLC Trip Record Data - TLC*. [Online; accessed 25. Apr. 2025]. Apr. 2025. URL: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[AB93] A. Aggoun and N. Beldiceanu. "Extending CHIP in order to solve complex scheduling and placement problems". In: *Mathematical and computer modelling* 17.7 (1993), pp. 57–73.

[ALL23] G. Audemard, C. Lecoutre, and E. Lonca. "Proceedings of the 2023 XCSP3 Competition". In: *arXiv preprint arXiv:2312.05877* (2023).

[ALP23] G. Audemard, C. Lecoutre, and C. Prud'Homme. "Guiding Backtrack Search by Tracking Variables During Constraint Propagation". In: *International Conference on Principles and Practice of Constraint Programming*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2023.

[App+09] D. L. Applegate, R. E. Bixby, V. Chvátal, W. Cook, D. G. Espinoza, M. Goycoolea, and K. Helsgaun. "Certification of an optimal TSP tour through 85,900 cities". In: *Operations Research Letters* 37.1 (2009), pp. 11–15.

[Asc96] N. Ascheuer. "Hamiltonian path problems in the on-line optimization of flexible manufacturing systems". PhD thesis. 1996.

[BC02] N. Beldiceanu and M. Carlsson. "A new multi-resource cumulatives constraint with negative heights". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2002, pp. 63–79.

[Ber13] T. Berthold. "Measuring the impact of primal heuristics". In: *Operations Research Letters* 41.6 (2013), pp. 611–614.

[BJM19] D. Bertsimas, P. Jaillet, and S. Martin. "Online vehicle routing: The edge of optimization in large-scale applications". In: *Operations Research* 67.1 (2019), pp. 143–162.

[Bjö+20] G. Björdal, P. Flener, J. Pearson, P. J. Stuckey, and G. Tack. "Solving satisfaction problems using large-neighbourhood search". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2020, pp. 55–71.

[Bou+04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. "Boosting systematic search by weighting constraints". In: *ECAI*. Vol. 16. 2004, p. 146.

[Bou+16a] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette. "XCSP3: an integrated format for benchmarking combinatorial constrained problems". In: *arXiv preprint arXiv:1611.03398* (2016).

[Bou+16b] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette. "XCSP3: an integrated format for benchmarking combinatorial constrained problems". In: *arXiv preprint arXiv:1611.03398* (2016).

[BPR11] G. Berbeglia, G. Pesant, and L.-M. Rousseau. "Checking the feasibility of dial-a-ride instances using constraint programming". In: *Transportation Science* 45.3 (2011), pp. 399–412.

[BRV16] K. Braekers, K. Ramaekers, and I. Van Nieuwenhuyse. "The vehicle routing problem: State of the art classification and review". In: *Computers & industrial engineering* 99 (2016), pp. 300–313.

[BS01] E. Balas and N. Simonetti. "Linear time dynamic-programming algorithms for new classes of restricted TSPs: A computational study". In: *INFORMS journal on Computing* 13.1 (2001), pp. 56–75.

[BT93] P. Briggs and L. Torczon. "An efficient representation for sparse sets". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 2.1-4 (1993), pp. 59–69.

[BV03a] R. Bent and P. Van Hentenryck. "Dynamic vehicle routing with stochastic requests". In: *IJCAI*. Citeseer. 2003, pp. 1362–1363.

[BV03b] C. Bessiere and P. Van Hentenryck. "To be or not to be... a global constraint". In: *International conference on principles and practice of constraint programming*. Springer. 2003, pp. 789–794.

[BV04] R. Bent and P. Van Hentenryck. "A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows". In: *Transportation Science* 38.4 (2004), pp. 515–530.

[Cap+18] Q. Cappart, C. Thomas, P. Schaus, and L.-M. Rousseau. "A Constraint Programming Approach for Solving Patient Transportation Problems". In: *Principles and Practice of Constraint Programming*. Ed. by J. Hooker. Cham: Springer International Publishing, 2018, pp. 490–506. ISBN: 978-3-319-98334-9.

[Cap+21] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz, and A. A. Cire. "Combining reinforcement learning and constraint programming for combinatorial optimization". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3677–3687.

[CL03] J.-F. Cordeau and G. Laporte. "A tabu search heuristic for the static multi-vehicle dial-a-ride problem". In: *Transportation Research Part B: Methodological* 37 (July 2003), pp. 579–594. DOI: 10.1016/S0191-2615(02)00045-0.

[Coo+11] W. J. Cook, D. L. Applegate, R. E. Bixby, and V. Chvatal. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[Cor+22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022. ISBN: 9780262046305.

[CV20] J. Christiaens and G. Vanden Berghe. "Slack induction by string removals for vehicle routing problems". In: *Transportation Science* 54.2 (2020), pp. 417–433.

[DCS18] E. Demirović, G. Chu, and P. J. Stuckey. "Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers". In: *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*. Springer. 2018, pp. 99–108.

[DDD05] G. Dooms, Y. Deville, and P. Dupont. "Cp (graph): Introducing a graph computation domain in constraint programming". In: *Principles and Practice of Constraint Programming-CP 2005: 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005. Proceedings 11*. Springer. 2005, pp. 211–225.

[Dem+16] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus. "Compact-table: efficiently filtering table constraints with reversible sparse bit-sets". In: *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*. Springer. 2016, pp. 207–223.

[Dem14] S. Demassey. *Global Constraint Catalog*. [Online; accessed 27. Nov. 2024]. June 2014. URL: https://sofdem.github.io/gccat.

[DS24] A. Delecluse and P. Schaus. "Black-Box Value Heuristics for Solving Optimization Problems with Constraint Programming (Short Paper)". In: *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024.

[DSV22] A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables for Routing Problems". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[DSV23] A. Delecluse, P. Schaus, and P. Van Hentenryck. "SEQUOIA: SEQuence-variable-based Optimization In Action for the Traveling Salesman Problem with Time Windows". In: *Doctoral Program of CP23*. 2023.

[DSV25] A. Delecluse, P. Schaus, and P. Van Hentenryck. "Sequence Variables: A Constraint Programming Computational Domain for Routing and Sequencing". Manuscript in preparation. 2025.

[dU10] R. F. da Silva and S. Urrutia. "A General VNS heuristic for the traveling salesman problem with time windows". In: *Discrete Optimization* 7.4 (2010), pp. 203–211. ISSN: 1572-5286. DOI: https://doi.org/10.1016/j.disopt.2010.04.002.

[Dum+95] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon. "An optimal algorithm for the traveling salesman problem with time windows". In: *Operations research* 43.2 (1995), pp. 367–371.

[DVS15] C. Dejemeppe, S. Van Cauwelaert, and P. Schaus. "The unary resource with transition times". In: *International conference on principles and practice of constraint programming*. Springer. 2015, pp. 89–104.

[FP17] J.-G. Fages and C. Prud'Homme. "Making the first solution good!" In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2017, pp. 1073–1077.

[Fre96] E. Freuder. "In pursuit of the holy grail". In: *ACM Computing Surveys (CSUR)* 28.4es (1996), 63–es.

[FS23] A. Froger and R. Sadykov. "New exact and heuristic algorithms to solve the prize-collecting job sequencing problem with one common and multiple secondary resources". In: *European Journal of Operational Research* 306.1 (2023), pp. 65–82.

[Gay+15] S. Gay, R. Hartert, C. Lecoutre, and P. Schaus. "Conflict ordering search for scheduling problems". In: *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*. Springer. 2015, pp. 140–148.

[GD19] T. Gschwind and M. Drexl. "Adaptive large neighborhood search with a constant-time feasibility test for the dial-a-ride problem". In: *Transportation Science* 53.2 (2019), pp. 480–491.

[Gen+98] M. Gendreau, A. Hertz, G. Laporte, and M. Stan. "A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows". In: *Operations Research* 46.3 (1998), pp. 330–335.

[Ger06] C. Gervet. "Constraints over structured domains". In: *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 2006, pp. 605–638.

[Ger93] C. Gervet. "New structures of symbolic constraint objects: sets and graphs Extended abstract". In: *constraints* 6 (1993), p. 2.

[Ger95] C. Gervet. "Set Intervals in Constraint Logic Programming: Definition and implementation of a language". PhD thesis. Université de Franche Comté Besançon, 1995.

[GHM09] D. Grimes, E. Hebrard, and A. Malapert. "Closing the open shop: Contradicting conventional wisdom". In: *International conference on principles and practice of constraint programming*. Springer. 2009, pp. 400–408.

[GHS15] S. Gay, R. Hartert, and P. Schaus. "Simple and scalable timetable filtering for the cumulative constraint". In: *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*. Springer. 2015, pp. 149–157.

[GLN05] D. Godard, P. Laborie, and W. Nuijten. "Randomized Large Neighborhood Search for Cumulative Scheduling." In: *ICAPS*. Vol. 5. 2005, pp. 81–89.

[GP06] G. Gutin and A. P. Punnen. *The traveling salesman problem and its variations*. Vol. 12. Springer Science & Business Media, 2006.

[HC88] P. V. Hentenryck and J.-P. Carillon. “Generality versus specificity: An experience with AI and OR techniques”. In: *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*. 1988, pp. 660–664.

[HE80] R. M. Haralick and G. L. Elliott. “Increasing tree search efficiency for constraint satisfaction problems”. In: *Artificial intelligence* 14.3 (1980), pp. 263–313.

[Hel00] K. Helsgaun. “An effective implementation of the Lin–Kernighan traveling salesman heuristic”. In: *European journal of operational research* 126.1 (2000), pp. 106–130.

[HG95] W. D. Harvey and M. L. Ginsberg. “Limited discrepancy search”. In: *IJCAI (1)*. 1995, pp. 607–615.

[Ho+18] S. C. Ho, W. Y. Szeto, Y.-H. Kuo, J. M. Leung, M. Petering, and T. W. Tou. “A survey of dial-a-ride problems: Literature review and recent developments”. In: *Transportation Research Part B: Methodological* 111 (2018), pp. 395–421.

[Hor+21] M. Horn, J. Maschler, G. R. Raidl, and E. Rönnberg. “A∗-based construction of decision diagrams for a prize-collecting scheduling problem”. In: *Computers & Operations Research* 126 (2021), p. 105125.

[HRB18] M. Horn, G. Raidl, and C. Blum. “Job sequencing with one common and multiple secondary resources: A problem motivated from particle therapy for cancer treatment”. In: *Machine Learning, Optimization, and Big Data: Third International Conference, MOD 2017, Volterra, Italy, September 14–17, 2017, Revised Selected Papers 3*. Springer. 2018, pp. 506–518.

[JV11] S. Jain and P. Van Hentenryck. “Large Neighborhood Search For Dial-a-Ride Problems”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2011, pp. 400–413.

[Kar09] R. M. Karp. “Reducibility among combinatorial problems”. In: *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*. Springer, 2009, pp. 219–241.

[KB23] R. Kuroiwa and J. C. Beck. “Solving Domain-Independent Dynamic Programming Problems with Anytime Heuristic Search”. In: (2023).

[KPS98] P. Kilby, P. Prosser, and P. Shaw. “Dynamic VRPs: A study of scenarios”. In: *University of Strathclyde Technical Report* 1.11 (1998).

[KS06] P. Kilby and P. Shaw. "Vehicle routing". In: *Handbook of Constraint Programming*. Vol. 2. Elsevier, 2006, pp. 801–836.

[KS99] R. Klein and A. Scholl. "Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling". In: *European Journal of Operational Research* 112.2 (1999), pp. 322–346.

[Lab+09a] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. "Reasoning with Conditional Time-Intervals. Part II: An Algebraical Model for Resources". In: *FLAIRS Conference*. 2009.

[Lab+09b] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. "Reasoning with conditional time-intervals. part ii: An algebraical model for resources". In: *Twenty-Second International FLAIRS Conference*. 2009.

[Lab+18a] P. Laborie, J. Rogerie, P. Shaw, and P. Vilıém. "IBM ILOG CP Optimizer for Scheduling". In: *Constraints* 23.2 (Apr. 2018), pp. 210–250. ISSN: 1383-7133. DOI: 10.1007/s10601-018-9281-x.

[Lab+18b] P. Laborie, J. Rogerie, P. Shaw, and P. Vilıém. "IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG". In: *Constraints* 23 (2018), pp. 210–250.

[Lab09] P. Laborie. "IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 6th International Conference, CPAIOR 2009 Pittsburgh, PA, USA, May 27-31, 2009 Proceedings 6*. Springer. 2009, pp. 148–162.

[LAB18] C. Liu, D. M. Aleman, and J. C. Beck. "Modelling and solving the senior transportation problem". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15*. Springer. 2018, pp. 412–428.

[Lab18] P. Laborie. "Objective landscapes for constraint programming". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15*. Springer. 2018, pp. 387–402.

[Lah82] A. Lahrichi. "Scheduling-the notions of hump, compulsory parts and their use in cumulative problems". In: *Comptes Rendus De L Academie Des Sciences Serie I-mathematique* 294.6 (1982), pp. 209–211.

[Law85] E. L. Lawler. "The traveling salesman problem: a guided tour of combinatorial optimization". In: *Wiley-Interscience Series in Discrete Mathematics* (1985).

[LBC12] A. Letort, N. Beldiceanu, and M. Carlsson. "A scalable sweep algorithm for the cumulative constraint". In: *International conference on principles and practice of constraint programming*. Springer. 2012, pp. 439–454.

[Lec+09] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. "Reasoning from last conflict (s) in constraint programming". In: *Artificial Intelligence* 173.18 (2009), pp. 1592–1614.

[Lec23] C. Lecoutre. "Ace, a generic constraint solver". In: *arXiv preprint arXiv:2302.05405* (2023).

[LK73] S. Lin and B. W. Kernighan. "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations research* 21.2 (1973), pp. 498–516.

[Lóp20] M. López-Ibáñez. *Instances for the TSPTW*. [Online; accessed 15. Feb. 2022]. Sept. 2020. URL: `https://lopez-ibanez.eu/tsptw-instances`.

[LR08] P. Laborie and J. Rogerie. "Reasoning with Conditional Time-Intervals." In: *FLAIRS*. 2008, pp. 555–560.

[LS07] M. Z. Lagerkvist and C. Schulte. "Advisors for incremental propagation". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 409–422.

[LS14] M. Lombardi and P. Schaus. "Cost impact guided LNS". In: *Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings 11*. Springer. 2014, pp. 293–300.

[Meh+88] D. Mehmet, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. "The constraint logic programming language CHIP". In: *Proceedings on the International Conference on Fifth Generation Computer Systems FGCS*. Vol. 88. 1988.

[MLS15] P. A. Melgarejo, P. Laborie, and C. Solnon. "A time-dependent no-overlap constraint: Application to urban delivery problems". In: *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*. Springer. 2015, pp. 1–17.

[MRS06] P. Meseguer, F. Rossi, and T. Schiex. "Soft Constraints, chapter 9 of Handbook of Constraint Programming". In: *Elsevier* 10 (2006), p. 11.

[MSV21] L. Michel, P. Schaus, and P. Van Hentenryck. "MiniCP: a lightweight solver for constraint programming". In: *Mathematical Programming Computation* 13.1 (2021), pp. 133–184. DOI: 10.1007/s12532-020-00190-7.

[MV12a] L. Michel and P. Van Hentenryck. "Activity-based search for black-box constraint programming solvers". In: *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*. Springer. 2012, pp. 228–243.

[MV12b] L. D. Michel and P. Van Hentenryck. "Constraint satisfaction over bit-vectors". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 527–543.

[Net+07] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. "MiniZinc: Towards a standard CP modelling language". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 529–543.

[OQ13] P. Ouellet and C.-G. Quimper. "Time-table extended-edge-finding for the cumulative constraint". In: *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings 19*. Springer. 2013, pp. 562–577.

[OT07] J. W. Ohlmann and B. W. Thomas. "A compressed-annealing heuristic for the traveling salesman problem with time windows". In: *INFORMS Journal on Computing* 19.1 (2007), pp. 80–90.

[PB96] J.-Y. Potvin and S. Bengio. "The vehicle routing problem with time windows part II: genetic search". In: *INFORMS journal on Computing* 8.2 (1996), pp. 165–172.

[Pes+98] G. Pesant, M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau. "An exact constraint logic programming algorithm for the traveling salesman problem with time windows". In: *Transportation Science* 32.1 (1998), pp. 12–29.

[Pes19] G. Pesant. "From support propagation to belief propagation in constraint programming". In: *Journal of Artificial Intelligence Research* 66 (2019), pp. 123–150.

[PFa] L. Perron and V. Furnon. *OR-Tools*. Version 7.2. Google. URL: https://developers.google.com/optimization/.

[PFb] L. Perron and V. Furnon. *OR-Tools Sequence Var*. Version 7.2. Google. URL: https://developers.google.com/optimization/reference/constraint_solver/constraint_solver/SequenceVar.

[PF22] C. Prud'homme and J.-G. Fages. "Choco-solver: A Java library for constraint programming". In: *Journal of Open Source Software* 7.78 (2022), p. 4708. DOI: 10.21105/joss.04708.

[PR18] D. Pisinger and S. Ropke. "Large neighborhood search". In: *Handbook of metaheuristics*. Springer, 2018, pp. 99–127.

[PR93] J.-Y. Potvin and J.-M. Rousseau. "A parallel route building algorithm for the vehicle routing and scheduling problem with time windows". In: *European Journal of Operational Research* 66.3 (1993), pp. 331–340.

[Pra23] C. Pralet. "Iterated maximum large neighborhood search for the traveling salesman problem with time windows and its time-dependent version". In: *Computers & Operations Research* 150 (2023), p. 106078.

[PSF04] L. Perron, P. Shaw, and V. Furnon. "Propagation guided large neighborhood search". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2004, pp. 468–481.

[Pug93] J.-F. Puget. "Set constraints and cardinality operator: Application to symmetrical combinatorial problems". In: *Third Workshop on Constraint Logic Programming–WCLP93*. 1993, p. 211.

[Ref04] P. Refalo. "Impact-based search strategies for constraint programming". In: *Principles and Practice of Constraint Programming–CP 2004: 10th International Conference, CP 2004, Toronto, Canada, September 27-October 1, 2004. Proceedings 10*. Springer. 2004, pp. 557–571.

[Rég94] J.-C. Régin. "A filtering algorithm for constraints of difference in CSPs". In: *AAAI*. Vol. 94. 1994, pp. 362–367.

[Rei91] G. Reinelt. "TSPLIB–A Traveling Salesman Problem Library". In: *ORSA Journal on Computing* 3.4 (1991), pp. 376–384.

[RP06] S. Ropke and D. Pisinger. "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows". In: *Transportation science* 40.4 (2006), pp. 455–472.

[RSL77] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. "An analysis of several heuristics for the traveling salesman problem". In: *SIAM journal on computing* 6.3 (1977), pp. 563–581.

[Sai+13] V. l. C. de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. "Sparse-Sets for Domain Implementation". In: *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*. 2013, pp. 1–10.

[Sav85] M. W. Savelsbergh. "Local search in routing problems with time windows". In: *Annals of Operations research* 4.1 (1985), pp. 285–305.

[Sch+24] P. Schaus, G. Derval, A. Delecluse, L. Michel, and P. V. Hentenryck. *MaxiCP: A Constraint Programming Solver for Scheduling and Vehicle Routing*. 2024. URL: `https://github.com/aia-uclouvain/maxicp`.

[Sch13] P. Schaus. "Variable objective large neighborhood search: A practical approach to solve over-constrained problems". In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE. 2013, pp. 971–978.

[Sch99] C. Schulte. "Comparing Trailing and Copying for Constraint Programming." In: *ICLP*. Vol. 99. 1999, pp. 275–289.

[Sco+17] J. D. Scott, P. Flener, J. Pearson, and C. Schulte. "Design and implementation of bounded-length sequence variables". In: *Integration of AI and OR Techniques in Constraint Programming: 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings 14*. Springer. 2017, pp. 51–67.

[SFP15] J. D. Scott, P. Flener, and J. Pearson. "Constraint solving on bounded string variables". In: *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*. Springer. 2015, pp. 375–392.

[Sha04] P. Shaw. "A constraint for bin packing". In: *International conference on principles and practice of constraint programming*. Springer. 2004, pp. 648–662.

[Sha98] P. Shaw. "Using constraint programming and local search methods to solve vehicle routing problems". In: *International conference on principles and practice of constraint programming*. Springer. 1998, pp. 417–431.

[Sid+18] N. Sidiropoulos, S. H. Sohi, T. L. Pedersen, B. T. Porse, O. Winther, N. Rapin, and F. O. Bagger. "SinaPlot: an enhanced chart for simple and truthful representation of single observations over multiple classes". In: *Journal of Computational and Graphical Statistics* 27.3 (2018), pp. 673–676.

[SLT06] C. Schulte, M. Lagerkvist, and G. Tack. "Gecode". In: *Software download and online material at the website: http://www. gecode. org* (2006), pp. 11–13.

[SMV25] P. Schaus, L. Michel, and P. Van Hentenryck. *Constraint Programming*. `https://www.edx.org/learn/computer-programming/universite-catholique-de-louvain-constraint-programming`. [Online; accessed 11. Feb. 2025]. Feb. 2025.

[Sol87] M. M. Solomon. "Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints". In: *Operations research* 35.2 (1987), pp. 254–265.

[Sol95] I. Solver. "Object-oriented constraint programming". In: *ILOG SA* 12 (1995).

[ST13] C. Schulte and G. Tack. "View-based propagator derivation". In: *Constraints* 18 (2013), pp. 75–107.

[STL10] C. Schulte, G. Tack, and M. Z. Lagerkvist. "Modeling and programming with gecode". In: *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael* 1 (2010).

[Tan21] O. Tange. *GNU Parallel 20210822 ('Kabul')*. Aug. 2021. DOI: 10.5281/zenodo.5233953.

[TC72] F. A. Tillman and T. M. Cain. "An upperbound algorithm for the single and multiple terminal delivery problem". In: *Management Science* 18.11 (1972), pp. 664–682.

[Tea12] O. Team. *OscaR: Scala in OR*. 2012. URL: `https://bitbucket.org/oscarlib/oscar/src/dev`.

[Tho23] C. Thomas. "Advanced modelling and search techniques for routing and scheduling problems." PhD thesis. Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2023.

[TKS20] C. Thomas, R. Kameugne, and P. Schaus. "Insertion Sequence Variables for Hybrid Routing and Scheduling Problems". In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2020, pp. 457–474.

[TS18] C. Thomas and P. Schaus. "Revisiting the self-adaptive large neighborhood search". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15*. Springer. 2018, pp. 557–566.

[TSH21] R. Turkeš, K. Sörensen, and L. M. Hvattum. "Meta-analysis of metaheuristics: Quantifying the effect of adaptiveness in adaptive large neighborhood search". In: *European Journal of Operational Research* 292.2 (2021), pp. 423–442.

[Van+16] S. Van Cauwelaert, C. Dejemeppe, J.-N. Monette, and P. Schaus. "Efficient filtering for the unary resource with family-based transition times". In: *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*. Springer. 2016, pp. 520–535.

[Van01] W.-J. Van Hoeve. "The alldifferent constraint: A survey". In: *arXiv preprint cs/0105015* (2001).

[Van02] P. Van Hentenryck. "Constraint and integer programming in OPL". In: *INFORMS Journal on Computing* 14.4 (2002), pp. 345–372.

[Van87] P. Van Hentenryck. "Consistency Techniques in Logic Programming". PhD thesis. Namur, Belgium: University of Namur, 1987.

[Van89] P. Van Hentenryck. *Constraint satisfaction in logic programming*. MIT press, 1989.

[VB06] P. Van Hentenryck and R. Bent. *Online stochastic combinatorial optimization*. The MIT Press, 2006.

[VDT92] P. Van Hentenryck, Y. Deville, and C.-M. Teng. "A generic arc-consistency algorithm and its specializations". In: *Artificial intelligence* 57.2-3 (1992), pp. 291–321.

[Vil07] P. Vilıém. "Global constraints in scheduling". In: (2007).

[VLS15] P. Vilím, P. Laborie, and P. Shaw. "Failure-directed search for constraint-based scheduling". In: *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*. Springer. 2015, pp. 437–453.

[VM14] P. Van Hentenryck and L. Michel. "Domain views for constraint programming". In: *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings 20*. Springer. 2014, pp. 705–720.

[VRF15] E. Vallada, R. Ruiz, and J. M. Framinan. "New hard benchmark for flowshop scheduling problems minimising makespan". In: *European Journal of Operational Research* 240.3 (2015), pp. 666–677.

[Zha+23] J. Zhang, K. Luo, A. M. Florio, and T. Van Woensel. "Solving large-scale dynamic vehicle routing problems with stochastic requests". In: *European Journal of Operational Research* 306.2 (2023), pp. 596–614.

[Zha98] W. Zhang. "Complete anytime beam search". In: *AAAI/IAAI*. 1998, pp. 425–430.

[ZLZ18] X. Zhang, Q. Li, and W. Zhang. "A Fast Algorithm for Generalized Arc Consistency of the Alldifferent Constraint." In: *IJCAI*. 2018, pp. 1398–1403.