Aggregation and Garbage Collection for Online Optimization

Alexander Ek[1,2](✉), Maria Garcia de la Banda[1], Andreas Schutt[2], Peter J. Stuckey[1,2], and Guido Tack[1,2]

[1] Monash University, Melbourne, Australia
{Alexander.Ek,Maria.GarciadelaBanda,Peter.Stuckey,Guido.Tack}@monash.edu
[2] Data61, CSIRO, Melbourne, Australia
Andreas.Schutt@data61.csiro.au

Abstract. Online optimization approaches are popular for solving optimization problems where not all data is considered at once, because it is computationally prohibitive, or because new data arrives in an ongoing fashion. Online approaches solve the problem iteratively, with the amount of data growing in each iteration. Over time, many problem variables progressively become *realized*, i.e., their values were fixed in the past iterations and they can no longer affect the solution. If the solving approach does not remove these realized variables and associated data and simplify the corresponding constraints, solving performance will slow down significantly over time. Unfortunately, simply removing realized variables can be incorrect, as they might affect unrealized decisions. This is why this complex task is currently performed manually in a problem-specific and time-consuming way. We propose a problem-independent framework to identify realized data and decisions, and remove them by summarizing their effect on future iterations in a compact way. The result is a substantially improved model performance.

1 Introduction

Online optimization tackles the solving of problems that evolve over time. In online optimization problems, in some areas also called *dynamic* or *reactive*, the set of input data is only partially known a priori and new data, such as new customers and/or updated travel times in a dynamic vehicle routing problem, continuously or periodically arrive while the current solution is executed. This new data and the current execution state must be incorporated into the problem in an ongoing fashion to revise previous decisions and to take new decisions.

Online optimization problems are solved iteratively as a *sequence* of *offline* optimization problems, where each problem represents the available information state of the online problem at a particular point in time. In each iteration or *session*, the online optimization approach must create an *update instance* to update the current solution. An *update model* is a model that can be repeatedly

Partly funded by Australian Research Council grant DP180100151.

instantiated to create new update instances in the sequence; it incorporates the new data, the old data (possibly modified by external sources), and the results from solving the previous update instance (also possibly modified).

One inherent challenge of online problems is their continuous growth in size as time passes, which can yield an unnecessary and prohibitive performance decay. As a result, online optimization approaches often try to remove, or *garbage collect*, data that is irrelevant for current and future decisions to be made, such as completed delivery trips in dynamic vehicle routing problems. Unfortunately, garbage collection is a complex task, as it requires understanding the interaction between time, data and the variables and constraints used to model the problem. For this reason, existing removal methods (see related work in Sect. 6) are highly problem-specific and, thus, not easily transferable to other problems, or naïve, thus still causing significant performance issues as shown in Sect. 5.

We propose a problem-independent framework, in which the modeler provides minimal information about the relation between time, data, and variables. The framework performs three main steps. First, the modeler's information is used to analyze each constraint that would be part of the update instance, with the aim to automatically identify the data and variables that are now *realized*, i.e., can no longer affect future decisions, either because they can no longer change, or because any change is now irrelevant. Second, once the realized information for a constraint is inferred, the constraint is modified to aggregate as much of the realized information as it can. And third, once all constraints have been analyzed and aggregated, the garbage from all constraints is collected and removed to ensure it does not form part of the update instance. We note that this paper significantly extends our previous problem-independent online modeling approach [7], with a much more sophisticated and effective method for garbage collection. We also note that our online approach can also be used for solving large-scale offline optimization problem, when applying an iterative solving approach over a rolling horizon [19].

To sum up, our main **contributions** are as follows: (1) A systematic way of modeling and inferring when each part of a model is realized (Sect. 4.1), and how to utilize this. (2) For several kinds of constraints, methods of summarizing the effect that realized parts will have on the future without keeping them (Sect. 4.2). This is done via incrementally aggregating the realized values and slightly reformulating the constraints to use these aggregated values. (3) A garbage collection mechanism that analyzes a high-level constraint model, identifies which parts are never used again, and safely removes them from future sessions (Sect. 4.3). (4) An empirical evaluation of the proposed approach (Sect. 5).

2 Preliminaries

A *constraint satisfaction problem* (CSP) $P = (\mathbf{X}, \mathbf{D}, \mathbf{C})$ consists of a set of variables $\mathbf{X}$, a function $\mathbf{D}$ mapping each variable $x \in \mathbf{X}$ to its domain (usually a finite set of values) D_x, and a set of constraints $\mathbf{C}$ over $\mathbf{X}$. For optimization problems, we add a special variable o to $\mathbf{X}$, called the *objective*, to be w.l.o.g.

minimized. A *dynamic constraint satisfaction problem* (DCSP) [6] is a (possibly infinite) list of CSPs $DP = (P_1, P_2, \dots)$, where each P_i represents the state after the ith problem change is considered, where a change can remove and/or add constraints. It allows us to consider each session s_i as solving the stand-alone CSP P_i. A solving method is *offline* if it is designed to solve one CSP, and *online* if it is designed to iteratively solve and generate the CSPs in a DCSP.

We distinguish between a problem *model*, where the input data is described in terms of *parameters* (i.e., data that will be known before the search starts), and a model *instance*, where the values of the parameters are added to the model. While an instance can be directly represented as a CSP, a model can be represented as a *parameterized CSP* [13] $P[\mathit{Data}]$, that is, a family of CSPs $P[\delta]$ for every $\delta \in \mathit{Data}$. The parameterization also applies to its components $(\mathbf{X}[\mathit{Data}], \mathbf{D}[\mathit{Data}], \mathbf{C}[\mathit{Data}])$. This allows us to extend the online approach of [7] by representing a DCSP as the list $DP = (P[\delta_1], P[\delta_2], \cdots)$, where data can change over time, while the underlying model remains unchanged.

We assume DCSPs have *complete recourse* [5], that is, all their CSPs are feasible. We also assume DCSPs have *uncertainty in execution*, that is, the values of the parameters between consecutive CSPs, $P[\delta_i]$ and $P[\delta_{i+1}]$, can be modified by external sources. As an example for external modifications, consider a dynamic vehicle routing problem. The distance between two locations may change (e.g., due to road works or traffic congestion), or a vehicle may spend more time at a customer due to unforeseen delays. External modifications are, however, no longer possible once the parameters in question become *realized*, that is, once their actual values are either observed (indicating they refer to the past) or guaranteed never to be observed (indicating they will never be used and thus can be safely ignored). For example, once a vehicle has actually left a customer site, the time it left is now realized and can be assumed as fixed forever.

Note that, for simplicity, we do not use stochastic information during online optimization [2,20,21], nor predict future changes based on historical data [3]. We simply react to any changes that occur in what [4] calls *pure reaction*, and resolve each CSP in its entirety for each session. Thus, no information about $P[\delta_{i+1}]$ is used by $P[\delta_i]$. Also, we assume *non-preemption* for our online solving methods, that is, once a $P[\delta_i]$ starts to be solved, the execution cannot abort (either to later resume it or to restart it). However, we believe that the concepts presented in this paper can be generalized in a straightforward way to the stochastic, preemptive, incomplete recourse case.

We denote by *aggregation* the process of simplifying a constraint by replacing any fixed decision variables by their value. Faster solving can be achieved if we precompute the *aggregated value* of the fixed variables in each constraint and remove these variables from it. Consider for example a constraint containing the sum $\sum_i c_i \times x_i$. We can easily partition the coefficients c_i and variables x_i into two sets: those containing fixed decision variables F and the rest V. Then, we can substitute the sum $\sum_i c_i \times x_i$ by $\mathit{fixcost} + \sum_{i \in V} c_i \times x_i$, where the value $\mathit{fixcost} = \sum_{i \in F} c_i \times x_i$ is precomputed. This technique is common in many solvers, particularly copying solvers (e.g., Gecode [9]), which aggressively simplify all constraints before copying them. In our online setting, we aim to *remember* (and not redo) the simplifications done in past sessions, and thus incrementally *aggregate the results* over time (see Sect. 4.2).

3 Basics of Our Framework and Running Example

Given a DCSP $DP = (P[\delta_1], P[\delta_2], \dots)$, our framework consists of a series of (solving) sessions, where each session s_i starts to execute instance $P[\delta_i]$ after session s_{i-1} finishes. The result of session s_i contains the best solution to $P[\delta_i]$ that could be found within a given time limit. As we will see later, the result also contains information about which parts of δ_i have become garbage, and this information will be used to generate $P[\delta_{i+1}]$.

Time, quite obviously, plays an important role in online optimization. We denote by τ_i the *deadline* for session s_i, i.e., the latest time by which the result of s_i must have been produced (which can be calculated as the start time of s_i plus the time limit). Importantly, τ_i is available as a parameter in δ_i, essentially representing the value "now", as seen from the outside world. For example, if the current session s_i makes a decision to start a task on a particular machine, the earliest possible time for that job to start is τ_i, since any earlier time will be in the past by the time the solution can be implemented. Similarly, if an earlier session made the decision to start a task on a particular machine, then the current session may be able to revise that decision if the start time is still in the future, i.e., greater than τ_i. The parameter τ_i thus synchronizes the session with the outside world.

Each $\delta \in \mathit{Data}$ is a tuple (OS_δ, DV_δ), where OS_δ is the set of *object set* parameters and DV_δ is the set of *data value* parameters. The former contains a set per type of dynamic object in the problem, i.e., per type of object that can arrive as time progresses, such as the set of jobs for a job scheduling problem, the set of product orders for a product manufacturing problem, and the set of customer requests for a vehicle routing problem. Typically, these sets are the backbone of the DCSP models, as they index most loops and parameter arrays. For example, the set of jobs will index the array of precedences among jobs, and the set of customer requests will index the array containing the amount of time required by each customer request. This is why object sets are the key to our garbage collection method. It is also why the objects in each object set of OS_δ must be uniquely named, with their number determining the size of the instance.

The set of data value parameters denotes the particular givens in the current state of the instance. For example, the current location of all vehicles to route, the current expected travel times between points in a traffic network, or the durations of the jobs that must be run. Each element of the set can be defined as a singleton or as a multi-dimensional array possibly indexed by object sets.

Many of the data value parameters, decision variables and constraints in instance $P[\delta_{i+i}]$ will need to refer to the values obtained by session s_i when solving $P[\delta_i]$. Thus, every $\delta_{i+1} \in \mathit{Data}$ contains *new* data, i.e., objects and parameter values for session s_{i+1}, and *old* data, i.e., the decisions, objects and parameter values either produced by session s_i when solving $P[\delta_i]$ or modified by an external source after s_i (e.g., if a task previously scheduled required more time to be performed than expected). We will distinguish between old and new data sets by prefixing them with the ω and ν symbols, respectively.

3.1 Running Example: Dynamic Vehicle Routing

The following problem is used throughout the rest of the paper as an illustrative example. Consider a dynamic vehicle routing problem (DVRP), where a fleet of vehicles has to be routed to attend to upcoming customer requests as soon as possible, while minimizing the total travel time. The problem model is as follows.

Object Sets: The arrival of a new customer request c at time τ_c, is modeled by adding object c to the object set of *customer requests* C at time τ_c. We model vehicle availability by means of the availability of a new *tour*. This allows us to ignore whether the vehicle is new, or just finished its previous tour. Thus, a new tour t becoming available at time τ_t, is modeled by adding tour object t to the object set of *tours* T at time τ_t. Each tour $t \in T$ starts at its *start depot* $S(t)$, services some customer requests and ends at its *end depot* $E(t)$. The set of customer requests C, start depots $S(T)$ and end depots $E(T)$, yield a set of locations modeled as *nodes* N. As a result, C and T are input object sets, while N is an object set constructed from them as $N = S(T) \cup E(T) \cup NC(C)$, where $NC(C)$ maps the customer requests to their associated nodes.

Data Value Parameters: They are provided by two tables. The first, $\texttt{wait}\colon N \rightarrow \mathbb{Z}^{\geq}$, provides the amount of time $\texttt{wait}_n$ a vehicle has to wait after reaching node $n \in N$ before it can go on to the next node (this models, e.g., the time required to provide some service at node n). The second, $\texttt{dist}\colon N \times N \rightarrow \mathbb{Z}^{\geq}$, provides the traveling time $\texttt{dist}_{nm}$ from every node n to any node m in N. We assume $\texttt{wait}_n = 0$ if n corresponds to a depot. In addition, the special parameter τ gives the end of solving time for the current session, and hence the earliest time after which decisions can be modified.

Variables: While the set of customer requests serviced by a tour must be decided before the tour begins (i.e., leaves its starting depot), the time of service (and thus the order in which customer requests are serviced) can change. Therefore, the exact routes are modeled using three arrays of decision variables indexed by the set of nodes N: $serv_n$ determines the tour that serves node n, $succ_n$ the node that is served after node n, and $arri_n$ the time at which the tour arrives at node n.

Problem Constraints: The problem is modeled by the following six constraints. PC1 ensures each tour visits its start and end depot. PC2 records the time at which each tour that is new for the current session becomes available, by setting its arrival time at its start depot to τ. This ensures it is ready to start by the end of the current solving session. PC3 connects the nodes serviced by each tour. PC4 ensures the successor of a node n is not reached until n has been serviced for the required amount of time, and the time taken to travel to the successor is considered. The last two constraints ensure the whole successor array forms a single circuit. PC5 establishes a circuit for each tour using the global constraint on the successor array, thus ensuring every node belongs to one tour. SBC1 is a symmetry breaking constraint that closes the circuit by ensuring the successor of the end depot of tour t is the start depot of tour $t+1$ (in a fixed order), and the successor of the last tour is the start depot of the first one.

$$\textbf{forall}_{t \textbf{ in } T}\ \mathit{serv}_{S(t)} = \mathit{serv}_{E(t)} = t \tag{PC1}$$

$$\textbf{forall}_{t \textbf{ in } \nu T}\ \mathit{arri}_{S(t)} = \tau \tag{PC2}$$

$$\underset{n \textbf{ in } S(T)\cup NC(C)}{\textbf{forall}}\ \mathit{serv}_n = \mathit{serv}_{\mathit{succ}_n} \tag{PC3}$$

$$\underset{n \textbf{ in } S(T)\cup NC(C)}{\textbf{forall}}\ \mathit{arri}_n + \mathtt{wait}_n + \mathtt{dist}_{n,\mathit{succ}_n} \leq \mathit{arri}_{\mathit{succ}_n} \tag{PC4}$$

$$\textsc{Circuit}(\mathit{succ}) \tag{PC5}$$

$$\mathit{succ}_{E(\max(T))} = S(\min(T)) \ \wedge \underset{t \textbf{ in } T\setminus\max(T)}{\textbf{forall}}\ \mathit{succ}_{E(t)} = S(t+1) \tag{SBC1}$$

Overlap Constraints: Online solving methods need to build an *update model* that solves the problem while taking into account earlier decisions. The *overlap constraints* add information about how earlier information affects the current session. In the approach of [7], which we extend, these are constructed automatically from annotations of the original model. The overlap constraints focus on old decisions that must be *committed*, that is, decisions taken by the previous session (or modified by external sources) that cannot change in the current one. OC1 commits the customer requests serviced by a tour if the tour has left the start depot by the time the current solving session ends. OC2 commits the successor of a node as soon as the tour starts heading towards it. Successors of end depots are not included in this constraint, as that would interfere with SBC1. Finally, OC3 commits the arrival time if the node has started to be serviced by the time the session ends.

$$\begin{aligned}&\textbf{forall}_{t \textbf{ in } \omega T}\ \textbf{if } \tau \geq \omega \mathit{arri}_{\omega \mathit{succ}_{S(t)}} - \mathtt{dist}_{S(t),\omega \mathit{succ}_{S(t)}}\\ &\quad \textbf{then } \textbf{forall}_{n \textbf{ in } \omega N}\ \mathit{serv}_n = t \leftrightarrow \omega \mathit{serv}_n = t\end{aligned} \tag{OC1}$$

$$\underset{n \textbf{ in } \omega N\setminus E(\omega T)}{\textbf{forall}}\ \textbf{if } \tau \geq \omega \mathit{arri}_n + \mathtt{wait}_n \textbf{ then } \mathit{succ}_n = \omega \mathit{succ}_n \tag{OC2}$$

$$\underset{n \textbf{ in } \omega N}{\textbf{forall}}\ \textbf{if } \tau \geq \omega \mathit{arri}_n \textbf{ then } \mathit{arri}_n = \omega \mathit{arri}_n \textbf{ else } \mathit{arri}_n \geq \tau \tag{OC3}$$

Objective: We minimize the total travel cost, computed as the sum of arrival times at the end depot minus departure times from the start depot:

$$\begin{aligned}&\mathit{cost} = \textbf{sum}_{t \textbf{ in } T}\ \mathit{arri}_{E(t)} - \mathit{depart}_{S(t)},\\ &\quad \textbf{where } \mathit{depart}_{S(t)} = \mathit{arri}_{\mathit{succ}_{S(t)}} - \mathit{dist}_{S(t),\mathit{succ}_{S(t)}}.\end{aligned} \tag{OBJ}$$

4 Automated Garbage Collection

Our *garbage collection* method aims to remove as many *garbage* objects as possible from our object sets, together with their associated data value parameters, decision variables and constraints. An object is considered garbage if it can be removed without affecting the solutions, that is, if all decisions associated with the object are realized, and their values can be safely aggregated by all constraints in the instance. Recall that a data value parameter or decision is realized if it has already been observed (and can therefore no longer change) or will never be observed (and can thus be safely ignored). As we will see, our garbage collection method cannot simply consider the instance being compiled by the current session. Instead, it must become an incremental and cumulative process that operates across multiple sessions, aggregating realized data and decisions.

The method consists of the following three main steps: (1) Identify realized data value parameters and decision variables, in order to determine the fully realized objects (Sect. 4.1); (2) Aggregate the constraints together with the realized data value parameters and decision variables used in these constraints (Sect. 4.2); (3) Collect the garbage resulting from the aggregation and remove it from future instances (Sect. 4.3).

Note that the above automated garbage collection method is performed at the beginning of each session, when constructing the constraint problem *instance* $P[\delta]$ that will be sent to the solver. This is achieved by reformulating the update model to reason about garbage, as we describe below. While it may be more efficient to deal with aggregation and garbage collection outside of the model, this has the downside of making the method problem-specific. In contrast, our proposed automated method deduces what is garbage from the realization information in the model alone and can be used for any problem, since both the aggregation of each constraint and the garbage collection method are independent of the model in which they occur.

4.1 Phase One: Identifying Realizations

As time moves forward, objects, data value parameters, and decision variables become realized. We define a realization function $\mathcal{R}$ for a session solving instance $P[\delta] = (\mathbf{X}[\delta], \mathbf{D}[\delta], \mathbf{C}[\delta])$ as $\mathcal{R}\colon \bigcup OS_\delta \cup DV_\delta \cup \mathbf{X}[\delta] \rightarrow \{true, false\}$ to indicate whether a given object, data value parameter, or decision variable is realized. For ease of notation we define the complement to indicate whether an object, data parameter, or decision variable is not realized $\mathcal{NR}(z) = \neg\mathcal{R}(z)$, and extend both functions in the obvious way to operate on sets of elements.

If online optimization is used to solve a static problem, then any data value parameter that becomes known and any decision variable that is committed is automatically realized. For dynamic problems, however, the modeler's expert knowledge is needed to annotate the data value parameters and decision variables in the model to indicate when they are realized. Object realization can be automatically inferred from this. Let us illustrate this inference by means of our

DVRP model. We will assume that newly added data and variables are never realized, and focus on the old data and variables (identified by ω).

Variable and Data Realizations: As shown in Sect. 3.1, our DVRP model has three arrays of decision variables and two arrays of data value parameters. Their realization is provided by the modeler as the functions (for all $n, m \in N$):

$$\mathcal{R}(\omega serv_n) = \omega arri_{\omega succ_{S(\omega serv_n)}} - \mathtt{dist}_{S(\omega serv_n), \omega succ_{S(\omega serv_n)}} \leq \tau; \tag{$\mathcal{R}$V1}$$

$$\mathcal{R}(\omega succ_n) = \omega arri_{\omega succ_n} \leq \tau; \tag{$\mathcal{R}$V2}$$

$$\mathcal{R}(\omega arri_n) = \omega arri_n \leq \tau; \tag{$\mathcal{R}$V3}$$

$$\mathcal{R}(\omega \mathtt{wait}_n) = \omega arri_n + \omega \mathtt{wait}_n \leq \tau; \text{ and} \tag{$\mathcal{R}$D1}$$

$$\mathcal{R}(\omega \mathtt{dist}_{nm}) = \omega arri_{\omega succ_n} \leq \tau \vee \omega arri_m \leq \tau. \tag{$\mathcal{R}$D2}$$

$\mathcal{R}$V1 marks the tour that serves node n as realized if the vehicle of that tour has left the depot before the end of the current session (i.e., by τ). $\mathcal{R}$V2 marks the successor of node n as realized, if the tour arrives at n's successor by τ. $\mathcal{R}$V3 marks the arrival time at node n as realized, if it is less or equal than τ. $\mathcal{R}$D1 marks the waiting time at node n as realized, if the tour has already arrived at n and finished waiting by τ. $\mathcal{R}$D2 marks the distance from node n to node m as realized, if the tour has already arrived either at the successor of n or at m by τ. Note that if m is not n's successor, distance $\omega \mathtt{dist}_{nm}$ will never be used.

This example illustrates why realization needs to be defined by the modeler: The current model assumes that we cannot change the allocation of customers to vehicles after a vehicle has left the depot. This may be suitable for vehicles that need to pick up goods from the depot and deliver to the customers. But it may be unnecessarily restrictive for vehicles that provide a service, in which case we could change rule $\mathcal{R}$V1 to $\mathcal{R}(\omega serv_n) = \omega arri_n \leq \tau$.

Object Realizations and Correspondence: An object is realized when all the data value parameters and decisions about that object are realized. Since all five arrays of data value parameters and variables are indexed by object sets, we can automatically infer which objects are realized. Note that, since N is constructed from C and T, the realization relationships between them must be examined. We will come back to this later. We introduce the local realization function $\mathcal{R}^L \colon \delta \cup \mathbf{X}[\delta] \rightarrow \{true, false\}$, to reason about direct usage. The (for now manual) analysis determines that C and T are not used to index any of the five arrays. Therefore, all their objects are **locally** realized, i.e., $\mathcal{R}^L(c) = \mathcal{R}^L(t) = true$, for all $c \in \omega C$ and $t \in \omega T$. N is however used as index set in all arrays of decision variables and data value parameters. Its local realization is defined for all $n \in N$ as:

$$\begin{aligned}\mathcal{R}^L(n) = \mathcal{R}(\omega serv_n) \wedge \mathcal{R}(\omega succ_n) \wedge \mathcal{R}(\omega arri_n) \wedge \mathcal{R}(\omega \mathtt{wait}_n) \\ \wedge \left(\forall m \in \omega N \colon \mathcal{R}(\omega \mathtt{dist}_{nm}) \wedge \mathcal{R}(\omega \mathtt{dist}_{mn})\right),\end{aligned}$$

which marks object $n \in N$ as locally realized if all variables and data parameter arrays indexed by n (in one or more dimensions) are also realized.

Direct usage is extended to indirect usage $\forall c \in \omega C$, $t \in \omega T$, and $n \in \omega N$ as:

$$\begin{aligned}
\mathcal{R}(c) &= \mathcal{R}^L(c) \wedge \mathcal{R}^L(NC(c)); \\
\mathcal{R}(t) &= \mathcal{R}^L(t) \wedge \mathcal{R}^L(S(t)) \wedge \mathcal{R}^L(E(t)); \text{ and} \\
\mathcal{R}(n) &= \mathcal{R}^L(n) \wedge ((n = S(t) \vee n = E(t)) \rightarrow \mathcal{R}^L(t)) \wedge (n = NC(c) \rightarrow \mathcal{R}^L(c));
\end{aligned}$$

indicating a customer request c is realized if both c and its associated node $NC(c)$ are locally realized; a tour t is realized if t, its start node, and its end node are locally realized; and a node n is realized if n is locally realized and its associated tour or customer are locally realized (which in this case is always true). For brevity, $\mathcal{R}(S)$ denotes the set of all realized objects in object set S.

4.2 Phase Two: Aggregation

Realized data, variables and objects can never change in future uses, or are never used in the future. Thus, a constraint can be removed if all its elements are realized. Consider, e.g., Eq. PC3: when $succ_n$ is realized the constraint must already hold and can thus be ignored. If a constraint cannot be removed, any subexpression containing only realized elements can be replaced with an aggregated value. In general, the aggregation consists of three steps: (1) identifying the set of realized objects that can be aggregated, (2) redefining the constraint to use the aggregations, and (3) introducing and defining the aggregated value(s).

In most constraints, the realized elements are not used to access any other values (e.g., as array index). For them aggregation is straightforward. Consider, for example, the objective OBJ of the DVRP model, which loops over object set T. Partitioning T into realized and non-realized objects gives:

$$\mathit{cost} = \mathbf{sum}_{t \textbf{ in } \mathcal{R}(T)}(\mathit{arri}_{E(t)} - \mathit{depart}_{S(t)}) + \mathbf{sum}_{t \textbf{ in } \mathcal{NR}(T)}(\mathit{arri}_{E(t)} - \mathit{depart}_{S(t)}).$$

To aggregate the realized part, we simply compute and keep its value by introducing a new data parameter, $\mathtt{agg}_{\mathtt{obj}} : \mathbb{Z}^{\geq}$ to represent this value, and then transform the objective constraint OBJ into:

$$\mathit{cost} = \mathtt{agg}_{\mathtt{obj}} + \mathbf{sum}_{t \textbf{ in } \mathcal{NR}(T)}(\mathit{arri}_{E(t)} - \mathit{depart}_{S(t)}), \tag{AC-OBJ}$$

where $\mathtt{agg}_{\mathtt{obj}}$ sums up everything aggregated so far, $\omega\mathtt{agg}$ (corresponding to objects no longer in T), plus everything aggregated in the current session:

$$\mathtt{agg}_{\mathtt{obj}} = \omega\mathtt{agg}_{\mathtt{obj}} + \mathbf{sum}_{t \textbf{ in } \mathcal{R}(\omega T)}(\mathit{arri}_{E(t)} - \mathit{depart}_{S(t)}). \tag{AV-OBJ}$$

For constraints where the realized values are indeed used to access other values, aggregation can be quite complex. Consider, for example, the circuit constraint for a graph with six nodes $a, \ldots, f$. Suppose the previous session obtained the solution illustrated in Fig. 1(a). Suppose that nodes c, d and e are

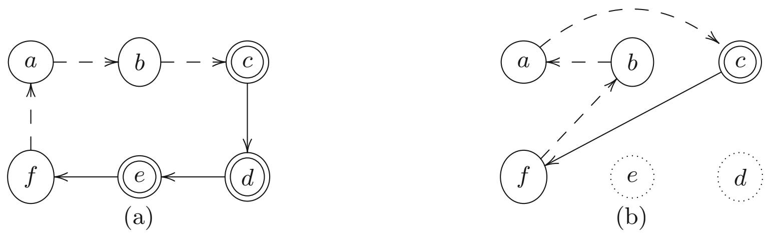

Fig. 1. Visualization of circuit aggregation via short-circuiting. The current solution (a) has realized nodes shown in double circles. Nodes d and e can be removed as garbage and node c is short-circuited to point at f, but must remain since (b) some non-garbage node will point at it, in the next session.

completely realized. Aggregation for this circuit can then be achieved by short-circuiting it as visualized in Fig. 1(b). Note we cannot remove c because some variable must point to c in future sessions. We can however remove d and e from the model, and they become garbage.

In general aggregation is constraint specific. Thankfully it is well understood, if not well publicized, and modern solvers, in particular Gecode [9], aggressively aggregate constraints. The extra challenge that arises in automatic garbage collection is that the result of the aggregation (e.g., $\texttt{agg}_{\texttt{obj}}$) needs to be communicated *across* sessions. Our automated garbage collection modifies the update model to both make use of the modified form of constraints with aggregation (e.g., AC-OBJ) values (e.g., AV-OBJ) as well as output this new aggregate value, so that it is available to the next session.

We can create a library of aggregating versions for common global constraints, which may require adding new arguments to transmit the aggregate values. We can also create a library of functions to compute the aggregate values required.

Finally, some constraints can be ignored throughout the aggregation process. In particular, redundant (or implied) constraints aim to speed up solving, but are not necessary for defining the problem, and are only relevant to the current instance. Hence, redundant constraints can be safely ignored during aggregation, and kept as-is for solving. Similarly, the overlap constraints are only required to communicate the effects from the previous session on the next session. Hence, they do not need to be considered for aggregation either. In contrast, while symmetry-breaking constraints can be safely ignored during aggregation (as they are only relevant in the current instance), they might need to be reformulated. This is because they eliminate solutions and, thus, can only be kept as-is if the remaining solutions are compatible with those left by symmetry breaking constraints of previous instances and all aggregations. In general, symmetry-breaking constraints need to be carefully designed on a model-by-model basis. This is outside the scope of this paper and remains future work. The symmetry-breaking constraints for our DVRP model are compatible with the aggregations since the end nodes point at the next start nodes, except for the last end node, which points to the first start node. This is the exact pattern our circuit aggregation will enforce as well.

4.3 Phase Three: Garbage Collection

Our method aims to determine *objects* that are garbage for removal of any data or variables associated to these objects. Without this, instances would constantly grow with time. As shown for the circuit constraint above, some realized (and aggregated) objects cannot be removed. Therefore, we need to determine what can be safely removed and what cannot. We will say that realized objects, data value parameters, and variables are *garbage* if they can be removed without creating an inconsistency, and are *non-garbage* otherwise. Note that only realized objects, data value parameters, and variables can be garbage since, otherwise, their new values might change the solutions found by subsequent sessions.

The garbage collection phase for $P[\delta]$ is itself divided into three steps. The first step identifies the set of realized objects that are non-garbage for each constraint in $P[\delta]$ when considered *in isolation*, that is, the non-garbage objects local to each constraint. Note that these objects, together with the data value parameters and variables associated to them, are the only realized elements that have *not* been already aggregated during the previous phase (i.e., phase two). Formally, the identification of the non-garbage local to a given constraint in $P[\delta] = (\mathbf{X}[\delta], \mathbf{D}[\delta], \mathbf{C}[\delta])$ is defined in terms of the function $\mathcal{NG}^L \colon \mathbf{C}[\delta] \rightarrow \{G \mid G \subseteq \bigcup OS_\delta\}$, which returns the set of realized objects in every object set that the given constraint c considers to be non-garbage and has, therefore, not been aggregated by c. For example, in our DVRP model, if c is the objective constraint OBJ, then $\mathcal{NG}^L(c)$ will return the empty set, since all realized objects are garbage for OBJ. However, if c is the circuit constraint PC5, then $\mathcal{NG}^L(c)$ will return any realized node n whose predecessor node is not realized (e.g., an end node is the predecessor of the start node of another tour), that is, it returns $\{n \in \mathcal{R}(N) \mid succ_m = n \text{ and } m \in \mathcal{NR}(N)\}$. In the example for the circuit constraint provided in the previous section, this set would contain the realized node named c, but not the nodes named d and e.

The second step in this phase collates the local non-garbage information to determine the realized objects that are non-garbage for the entire instance $P[\delta]$. To do this we define a global non-garbage function:

$$\mathcal{NG} \colon OS_\delta \rightarrow \{NG \mid NG \subseteq S,\ S \in OS_\delta\},\ \textbf{s.t.}\ \mathcal{NG}(S) \subseteq S,\ \forall S \in OS_\delta,$$

which maps each objects set $S \in OS_\delta$ to the objects in S that are non-garbage:

$$\mathcal{NG}(S) = \mathcal{R}(S) \cap \bigcup\nolimits_{c \in \mathbf{C}[\delta]} \mathcal{NG}^L(c), \quad \forall S \in OS_\delta,$$

that is, the subset of all objects identified by any constraint c in the instance as non-garbage, that are also realized objects of S. For ease of notation we use the complement to indicate the set of realized objects that are garbage $\mathcal{G}(S) = \mathcal{R}(S) \setminus \mathcal{NG}(S)$ and can therefore be removed.

The third and last step in this phase removes any garbage identified at the beginning of session s_i to create instance $P[\delta_i]$, thus eliminating the garbage from all future instances as well. This is already partially achieved in phase two, by modifying the constraints to aggregate all realized data value parameters and variables. To complete the task, we must remove any objects identified as garbage from the input object sets. We do this by redefining each input object set S as $\mathcal{NG}(S)+\mathcal{NR}(S)$, thus ensuring S retains every realized object that has been identified as non-garbage ($\mathcal{NG}(S)$), plus every object that is not realized ($\mathcal{NR}(S)$). Note that $\mathcal{NR}(S)$ contains all new objects of S and all old, non-realized ones. Note also that constructed object sets do not need to be redefined, as they are built from the input sets. For example, for our DVRP this step would require redefining the object set of tours $T = \mathcal{NG}(T) + \mathcal{NR}(T)$ and the object set of customer requests $C = \mathcal{NG}(C)+\mathcal{NR}(C)$. Once this is done, the definition of the constructed set of nodes $N = S(T) \cup E(T) \cup NC(C)$ automatically takes advantage of the (possibly smaller) T and C sets. We also extend the model to output the data for the new collections of objects, to use in the next session. This implicitly removes any data associated with a garbage object. Garbage variables are implicitly removed since the object set used to construct them no longer contains garbage objects. Therefore, once phase three finishes, $P[\delta_i]$ does not refer to objects, data value parameters, or variables identified as garbage.

5 Experimental Results

We present experimental results that demonstrate the effectiveness of our garbage collection approach. For this purpose, we took the standard MiniZinc [15] models of two optimization problems and transformed them (by hand), adding auxiliary functions, predicates and parameter definitions that implement the three phases of the approach. We use the MiniZinc 2.4.3 toolchain and the Gecode 6.1.1 solver. The iterative online optimization algorithm is implemented as a Python script that, for each session s_i, prepares the session data δ_i based on the previous session's result, and then calls MiniZinc to compile and solve the $P[\delta_i]$ instance.

5.1 Dynamic Vehicle Routing

Our first experiment is based on the DVRP model used as the running example, and the class 1 input data file `100-0-rc101-1`,[1] which provides coordinates for each node, service times for each customer request, the number of vehicles available (16), the start and end depots (always the same), and the start time of each time-window. We use this to compute the distance between any two nodes as their Euclidean distance, and to set the time at which a customer request arrives within a given time-window, as the start time of that time-window. No

[1] http://becool.info.ucl.ac.be/resources/benchmarks-dynamic-and-stochastic-vehicle-routing-problem-time-windows.

Table 1. Shows the session number, total number of customer requests so far, total number of tours added so far, (using garbage collection (GC) and aggregation:) best objective value found, compilation time, and runtime, number of garbage collected tours, and number of alive customers, (without GC and aggregation:) best objective value found, compilation time, and runtime.

#	custs.	tours	with GC					without GC		
			obj.	comp. (s)	run (s)	$\mathcal{G}(T)$	alive C	obj.	comp. (s)	run (s)
1	2	16	64	0.18	>45.00	0	2	64	0.21	>45.00
2	3	31	128	0.46	>45.00	14	3	128	3.49	>45.00
3	10	47	572	0.55	>45.00	29	8	572	38.01	>45.00
4	18	63	861	0.76	>45.00	45	15	—	>45.00	>45.00
5	24	79	1346	0.99	>45.00	60	21	—	>45.00	>45.00
6	36	95	2010	1.61	>45.00	76	32	—	>45.00	>45.00
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
14	91	223	5003	10.57	>45.00	204	82	—	>45.00	>45.00
15	95	239	5352	13.34	>45.00	220	86	—	>45.00	>45.00
16	96	255	5370	13.46	>45.00	236	87	—	>45.00	>45.00
17	99	271	5432	13.52	>45.00	252	90	—	>45.00	>45.00
18	99	287	5458	15.38	>45.00	268	90	—	>45.00	>45.00
19	100	303	5483	15.61	>45.00	284	91	—	>45.00	>45.00

other information in the data file was used. Note that whenever a tour ends, a new tour is added in that same session. We increased the current time τ by 50 time units each session, which is equivalent to 1 minute wall-clock time. Thus, the fleet gets updated instructions each minute. We also set the optimization runtime timeout to 45 s, to allow for potential overheads. We keep running until all 100 customers have been added.

The results in Table 1 show that our method significantly improves the compilation time; and without it, MiniZinc quickly gets overwhelmed with irrelevant tours and customers, slowing down the compilation. Further, using our method makes the problem suitable to run within a 45-s time-out. Note that empty tours in a session will be garbage collected in the next one.

5.2 Job-Shop Scheduling with Long Running Jobs

Our second experiment uses Job Shop Scheduling [10], a well studied, hard scheduling problem. We consider a rolling horizon version where jobs are spaced out by earliest start time. To highlight the advantages of automated garbage collection over the simple approach of [7], some of the jobs are given very long running times compared to other jobs. In the simple approach a job is garbage only if all tasks that arrived earlier are also garbage. These long running jobs prevent effective garbage collection under this policy, representing its worst case.

Table 2. Shows the session number, total number of jobs added so far, best objective value (makespan) found (all methods reported the same makespan), compilation time, and runtime without garbage collection, plus the same information and the garbage collected jobs for the simple garbage collection method of [7], and our proposed method.

#	jobs	obj.	no GC		simple GC			our GC		
			comp. (s)	run (s)	comp. (s)	run (s)	$\mathcal{G}(J)$	comp. (s)	run (s)	$\mathcal{G}(J)$
1	4	4278	0.09	0.11	0.09	0.11	0	0.08	0.10	0
2	8	4308	0.09	0.11	0.08	0.11	0	0.08	0.10	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
8	32	5428	0.46	0.50	0.47	0.51	0	0.27	0.30	7
9	36	5633	0.65	>1.00	0.68	>1.00	0	0.34	>1.00	9
10	40	5874	0.88	>1.00	0.90	>1.00	0	0.23	>1.00	18
11	44	8358	>1.00	>1.00	>1.00	>1.00	0	0.23	0.26	22
12	48	8388	—	—	—	—	—	0.17	0.20	30
13	52	8508	—	—	—	—	—	0.11	0.14	39
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
92	368	41028	—	—	—	—	—	0.19	0.22	350
93	372	41148	—	—	—	—	—	0.13	0.17	359
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

We run a similar experiment to that of [7], with the same model and data file `ft20` from the MiniZinc benchmarks,[2] and the same method to obtain an endless queue of jobs by repeatedly adding copies of the jobs in sequence. In each session we increase the current time τ by 408 scheduling time units, add the next 4 jobs in the queue, and set a time limit of 1 second. The first job (and every 40th subsequent job) is what we call a long-running job. We multiply the processing time, over all machines, of this job by 20.

Table 2 shows the benefit of using our method compared to no garbage collection and to the simple method proposed in [7]. By reasoning on objects individually (instead of finding the first non-realized job) we can remove any object that becomes realized. The simple method is prevented from garbage collecting them by the first job and, hence, becomes too slow.

As our experiments show, the overhead of the garbage collection steps is negligible compared to the overall compilation time, and compared to the performance gains in later sessions. The time complexity of the garbage collection steps depends on the complexity of the model, and is at most $O(n \log n)$ if the time complexity of the compilation without it is $O(n)$. Since garbage collection aims to minimize compilation time in future sessions, it will be quick and worthwhile in practice.

[2] https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop.

6 Related Work

Garbage collection is a well studied topic for programming languages and runtime environments [12], where it refers to the elimination of data that has no live references. In the online optimization context the data and past decisions always have references, but may still not be required for the current and future sessions, depending on which decisions are realized. Hence, determining what is garbage is more complex. As we have seen above, the model may require transformations such as aggregation in order to remove references to past decisions and data.

While online problems have been studied in great detail, almost all of this work concentrates on particular application problems (e.g., [11,16]). Any online problem has to tackle the difficulty that, without deletion, the problem continues to grow. While for particular application problems there may be simple rules that allow the modeler to define *garbage*, for a general definition, the interaction of realized decisions and earlier data with future decisions is complex.

Modeling support for online problem solving is still in its infancy. AIMMS has a notion of aggregation when using rolling horizon [17], which allows the effect of previous realized decisions to be accounted for in some parts of the model, such as the objective. The notion of realization used is much simpler, and automated aggregation of complex constraints like circuit is not considered.

Online MiniZinc [7] provides a framework that supports the automatic construction of the update model from an annotated offline version of the problem. It considers a simple form of garbage collection which only removes consecutive garbage objects until a non-garbage object is reached. The modeler is responsible for determining this directly, and aggregation is ignored. As a result, it is only usable when all aggregates are *true*. In contrast, the approach presented in this paper requires a complex model analysis, since we automate the understanding of how realized data and decisions can affect future decisions. In turn, our analysis requires a new view of modelling, where object sets are carefully used to represent dynamic data, and dependent object set creation is introduced.

Open global constraints [1,8] are a form of extensible constraint useful for online optimization problems whose size grows. The focus is on correct propagation of open constraints when not all information about them is available. They do not examine garbage collection since there is no notion of realization in the general open-world setting. The constraints simply grow as time progresses.

There is surprisingly little published work on aggregation of constraints. While the simplification of constraints when their variables become fixed is well understood, it is rarely documented. There are preprocessing methods that consider this, such as Boolean equi-propagation [14]. There has been work on eliminating variables during propagation [18], although we are not aware of a system that currently implements this. Aggregation, where information resulting from the simplification must be stored, does not seem to have been considered before.

7 Conclusion

Building online optimization systems has in the past been a rather complex process. Essentially, the modeler does not build a model of the problem, but an update model, which reasons about how to take information from the previous session and the new data to solve a new problem. We have previously [7] showed how to construct the update model automatically from the original model using annotations, but only introduced a very simple form of garbage collection: the modeler determines the latest object such that all previous objects are realized. In this paper, we now provide a comprehensive and automatic approach to garbage collection. The modeler specifies the rules for realization, and the garbage is automatically determined. We rely on uniquely named objects to ensure consistency of information across sessions. We tackle the key problem of aggregation, ignored in [7], where some constraints may need to be modified to record the effect of previously realized decisions.

References

1. Barták, R.: Dynamic global constraints in backtracking based environments. Ann. Oper. Res. **118**, 101–119 (2003). https://doi.org/10.1023/A:1021805623454
2. Bent, R., Van Hentenryck, P.: Scenario-based planning for partially dynamic vehicle routing with stochastic customers. Oper. Res. **52**(6), 977–987 (2004)
3. Bent, R., Van Hentenryck, P.: Online stochastic optimization without distributions. In: ICAPS 2005, pp. 171–180 (2005)
4. Brown, K.N., Miguel, I.: Uncertainty and change. In: Handbook of Constraint Programming, pp. 731–760. Elsevier (2006). (Chap. 21)
5. Dantzig, G.B.: Linear programming under uncertainty. Manag. Sci. **1**(3/4), 197–206 (1955)
6. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In: AAAI 1988, pp. 37–42. AAAI Press, June 1988
7. Ek, A., Garcia de la Banda, M., Schutt, A., Stuckey, P.J., Tack, G.: Modelling and solving online optimisation problems. In: AAAI 2020, pp. 1477–1485. AAAI Press (2020)
8. Faltings, B., Macho-Gonzalez, S.: Open constraint programming. Artif. Intell. **161**(1–2), 181–208 (2005)
9. Gecode Team. Gecode: A generic constraint development environment (2020). http://www.gecode.org
10. Graham, R.: Bounds for certain multiprocessing anomalies. Bell Syst. Tech. J. **45**(9), 1563–1581 (1966)
11. Jaillet, P., Wagner, M.R.: Online vehicle routing problems: a survey. In: Golden, B., Raghavan, S., Wasil, E. (eds.) The Vehicle Routing Problem: Latest Advances and New Challenges. Operations Research/Computer Science Interfaces, vol. 43, pp. 221–237. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-77778-8_10
12. Jones, R.E., Hosking, A.L., Moss, J.E.B.: The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman and Hall/CRC Applied Algorithms and Data Structures Series. CRC Press, Boca Raton (2011)

13. Mears, C., Garcia de la Banda, M., Wallace, M., Demoen, B.: A method for detecting symmetries in constraint models and its generalisation. Constraints **20**(2), 235–273 (2014). https://doi.org/10.1007/s10601-014-9175-5
14. Metodi, A., Codish, M., Stuckey, P.J.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. J. Artif. Intell. Res. **46**, 303–341 (2013)
15. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
16. Pruhs, K., Sgall, J., Torng, E.: Online scheduling. In: Handbook of Scheduling - Algorithms, Models, and Performance Analysis. Chapman and Hall/CRC (2004)
17. Roelofs, M., Bisschop, J.: Time-based modelling. In: AIMMS: The Language Reference, May 2, 2019 edn. (2019). www.aimms.com. (Chap. 33)
18. Schulte, C., Stuckey, P.J.: Dynamic variable elimination during propagation solving. In: Antoy, S., Albert, E. (eds.) Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Valencia, Spain, 15–17 July 2008, pp. 247–257. ACM (2008)
19. Sethi, S., Soerger, G.: A theory of rolling horizon decision making. Ann. Oper. Res. **29**, 387–415 (1991). https://doi.org/10.1007/BF02283607
20. Van Hentenryck, P., Bent, R.: Online Stochastic Combinatorial Optimization. MIT Press, Cambridge (2009)
21. Verfaillie, G., Jussien, N.: Constraint solving in uncertain and dynamic environments: a survey. Constraints **10**(3), 253–281 (2005). https://doi.org/10.1007/s10601-005-2239-9