Simple and Scalable Time-Table Filtering for the Cumulative Constraint

Steven Gay, Renaud Hartert, Pierre Schaus

UCLouvain, ICTEAM,
Place Sainte Barbe 2,
1348 Louvain-la-Neuve, Belgium
{firstname.lastname}@uclouvain.be

Abstract. `Cumulative` is an essential constraint in the CP framework, and is present in scheduling and packing applications. The lightest filtering for the cumulative constraint is time-tabling. It has been improved several times over the last decade. The best known theoretical time complexity for time-table is $O(n \log n)$ introduced by Ouellet and Quimper. We show a new algorithm able to run in $O(n)$, by relying on range min query algorithms. This approach is more of theoretical rather than practical interest, because of the generally larger number of iterations needed to reach the fixed point. On the practical side, the recent synchronized sweep algorithm of Letort et al, with a time-complexity of $O(n^2)$, requires fewer iterations to reach the fix-point and is considered as the most scalable approach. Unfortunately this algorithm is not trivial to implement. In this work we present a $O(n^2)$ simple two step alternative approach: first building the mandatory profile, then updating all the bounds of the activities. Our experimental results show that our algorithm outperforms synchronized sweep and the time-tabling implementations of other open-source solvers on large scale scheduling instances, sometimes significantly.

Keywords: Constraint programming, Large-Scale, Scheduling, Cumulative Constraint, Time-table.

1 Preliminaries

In this paper, we focus on a single cumulative resource with a discrete finite capacity $C \in \mathbb{N}$ and a set of n tasks $\Omega = \{1, \ldots, n\}$. Each task i has a start time $s_i \in \mathbb{Z}$, a fixed duration $d_i \in \mathbb{N}$, and an end time $e_i \in \mathbb{Z}$ such that the equality $s_i + d_i = e_i$ holds. Moreover, each task i consumes a fixed amount of resource $c_i \in \mathbb{N}$ during its processing time. Tasks are non-preemptive, i.e., they cannot be interrupted during their processing time. In the following, we denote by $\underline{s}_i$ and $\overline{s}_i$ the earliest and the latest start time of task i and by $\underline{e}_i$ and $\overline{e}_i$ the earliest and latest end time of task i (see Fig. 1). The `cumulative` constraint [1] ensures that the accumulated resource consumption does not exceed the maximum capacity C at any time t (see Fig. 2): $\forall t \in \mathbb{Z} : \sum_{i \in \Omega \,:\, s_i \leq t < e_i} c_i \leq C$.

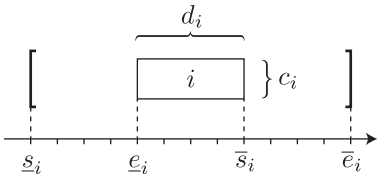

Fig. 1: Task i is characterized by its start time s_i, its duration d_i, its end time e_i, and its resource consumption c_i.

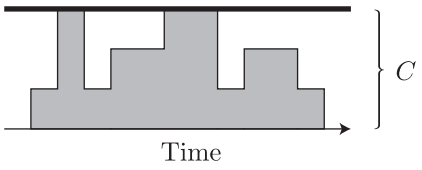

Fig. 2: Accumulated resource consumption over time. The `cumulative` constraint ensures that the maximum capacity C is not exceeded.

Even tasks that are not fixed convey some information that can be used by filtering rules. For instance, tasks with a tight execution window must consume some resource during a specific time interval known as *mandatory part*.

Definition 1 (Mandatory part). *Let us consider a task $i \in \Omega$. The mandatory part of i is the time interval $[\overline{s}_i, \underline{e}_i[$. Task i has a mandatory part only if its latest start time is smaller than its earliest end time.*

If task i has a mandatory part, we know that task i will consume c_i units of resource during all its mandatory part no matter its start time. Fig. 3 illustrates the mandatory part of an arbitrary task i.

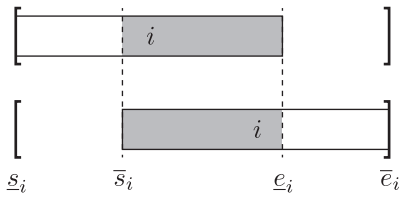

Fig. 3: Task i has a mandatory part $[\overline{s}_i, \underline{e}_i[$ if its latest start time $\overline{s}_i$ is smaller than its earliest end time $\underline{e}_i$: $\overline{s}_i < \underline{e}_i$. Task i always consumes the resource during its mandatory part no matter its start time.

By aggregation, mandatory parts allow us to have an optimistic view of the resource consumption over time. This aggregation is known as the *time-table*.

Definition 2 (Time-Table). *The time-table TT_Ω is the aggregation of the mandatory parts of all the tasks in Ω. It is defined as the following step function:*

$$\mathrm{TT}_\Omega = t \in \mathbb{Z} \longrightarrow \sum_{i \in \Omega \,|\, \overline{s}_i \leq t < \underline{e}_i} c_i. \tag{1}$$

The capacity of the resource is exceeded if $\exists t \in \mathbb{Z} : \mathrm{TT}_\Omega(t) > C$.

The time-table of a resource can be computed in $\mathcal{O}(n)$ by a sweep algorithm given the tasks sorted by latest start time and earliest end time [2,11,15].

The *time-table filtering rule* is formalized as follows:

$$(t < \underline{e}_i) \quad \wedge \quad (c_i + \mathrm{TT}_{\Omega\setminus i}(t) > C) \quad \Rightarrow \quad t < s_i. \tag{2}$$

Observe that this filtering rule only describes how to update the start time of a task. End times are updated in a symmetrical way.

Let j be a rectangle denoted $\langle a_j, b_j, h_j\rangle$ with $a_j \in \mathbb{Z}$ (resp. $b_j \in \mathbb{Z}$) its start (resp. end) time, $h_j \in \mathbb{N}$ its height, and $b_j - a_j$ its duration (length). The time-table TT_Ω can be represented as a contiguous sequence of rectangles

$$TT_\Omega = \langle -\infty, a_1, 0\rangle, \langle a_1, b_1, h_1\rangle, \ldots, \langle a_m, b_m, h_m\rangle, \langle b_m, \infty, 0\rangle \tag{3}$$

such that $b_i = a_{i+1}$ and that the following holds:

$$\forall \langle a_j, b_j, h_j\rangle \in TT_\Omega,\ \forall t \in [a_j, b_j[\quad : \quad TT_\Omega(t) = h_j. \tag{4}$$

We assume that the sequence is minimal, i.e., no consecutive rectangles have the same height. The maximum number of rectangles is thus limited to $2n + 1$.

Definition 3 (Conflicting Rectangle). *For a task i, a left-conflicting rectangle is a rectangle $\langle a_j, b_j, h_j\rangle \in \mathrm{TT}_{\Omega\setminus i}$ such that $(a_j < \underline{e}) \wedge (b_j \geq \underline{s}_i) \wedge (h_j > C - c_i)$. We say that the task is in left-conflict with rectangle j. Right-conflicting rectangles are defined symmetrically.*

The time-table filtering rule can thus be rephrased as follows:

$$\forall i \in \Omega,\ \forall \langle a_j, b_j, h_j\rangle \in TT_{\Omega\setminus i} \ :\ j \text{ is in left-conflict with } i \Rightarrow b_j \leq s_i. \tag{5}$$

Definition 4 (Time-Table Consistency). *A cumulative constraint is left (resp. right) time-table consistent if no task has a left (resp. right) conflicting rectangle. It is time-table consistent if it is left and right time-table consistent.*

2 Existing algorithms for Time-Tabling

Using the notion of conflicting rectangles, one can design a naive time-tabling algorithm by confronting every task to every rectangle of the profile. The following algorithms improve on this, mainly by avoiding fruitless confrontations of rectangles and tasks.

Sweep-line algorithm. The sweep-line algorithm introduced by Beldiceanu et al. [2] introduces tasks from left to right, and builds the mandatory profile on-the-fly. This allows to confront tasks and rectangles only if they can overlap in time. It can factorize confrontations of a rectangle against several tasks, by organizing tasks in a heap. It pushes tasks to the right until they have no left-conflicting rectangle, as pictured in Figure 4(c). This algorithm runs in $O(n^2)$.

Idempotent sweep-line algorithm. The sweep-line algorithm by Letort et. al [11] improves on building the profile on-the-fly, by taking in consideration mandatory parts that appear dynamically as tasks are pushed. It reaches left-time-table consistency in $O(n^2)$, or $O(n^2 \log n)$ for its faster practical implementation.

Interval tree algorithm. The algorithm of Ouellet and Quimper [14] first builds the profile, then introduces rectangles and tasks in an interval tree. Rectangles are introduced by decreasing height, tasks by increasing height. This allows tasks and rectangles to be confronted only when their heights do conflict. For each task introduction, the tree structure decides in $\log n$ if its time domain conflicts with some rectangle. Its filtering is weaker, since it pushes a task i only after left-conflicting rectangles that overlap $[\underline{s}_i, \underline{e}_i[$, as pictured in Figure 4(b). The algorithm has time complexity $O(n \log n)$.

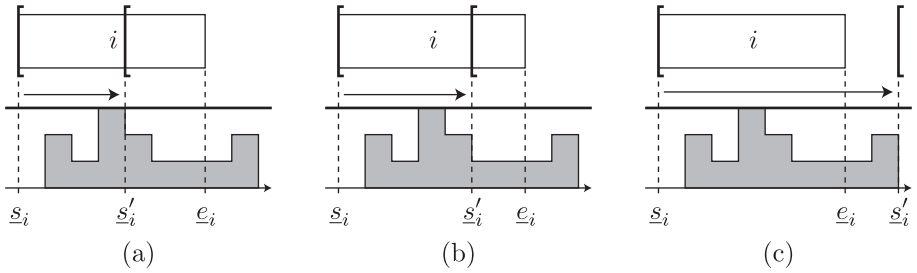

Fig. 4: Filtering obtained for (a) our linear time-tabling (b) Ouellet et al [14] and (c) Beldiceanu et al [2].

New algorithms. In this paper, we introduce two new algorithms for time-tabling. The first one is of theoretical interest and runs in $O(n)$. It uses range-max-query algorithms to determine whether a task has a conflicting rectangle. As the algorithm of Ouellet et al [14], it confronts task i only with rectangles overlapping $[\underline{s}_i, \underline{e}_i[$, but only chooses the one with the largest height instead of the largest end. Thus, it prunes even less, as depicted in Figure 4(a).

The second algorithm is practical, and runs in $O(n^2)$. It separates profile building from task sweeping. To locate tasks on the profile, it exploits residues from previous computations, and incrementally removes fixed tasks that cannot lead to any more pruning. It uses sweeping, thus pruning as much as [2] per call, but it updates both the earliest start and latest end times of the tasks in a single execution.

3 A linear time-table filtering algorithm

In order to obtain linear time complexity, we will confront task i to only one well-chosen rectangle, for every task i.

Proposition 1. *Suppose the mandatory profile does not overload the resource. Let i be a task, and j^* be a highest rectangle of the profile overlapping $[\underline{s}_i, \min(\underline{e}_i, \overline{s}_i)[$: $j^* = argmax_j \{h_j \mid \langle a_j, b_j, h_j\rangle \in \mathrm{TT}_\Omega$ and $[a_j, b_j[\cap[\underline{s}_i, \min(\underline{e}_i, \overline{s}_i)[\neq \emptyset\}$.*

Then j^* is in left-conflict iff $h_{j^*} + c_i > C$; otherwise i has no rectangles in left-conflict.

Proof. If i has a mandatory part, we only need to look for conflict rectangles overlapping $[\underline{s}_i, \overline{s}_i[$, since the profile already includes the mandatory part of i. Otherwise, we need to look at $[\underline{s}_i, \underline{e}_i[$. If rectangle j^* is not in left-conflict with i, then no other rectangle can, since it would need to be higher than j^*.

To retrieve the index j^*, we must answer the question: given a vector of values and two indices on this vector, what is the index between those two indices that has the highest value? This kind of query corresponds to the range max query problem[1], it can be done in constant time, given a linear time preprocessing [7].

Example 1. Assume the vector is $values = [5, 4, 2, 1, 4, 3, 0, 8, 2, 3]$. The range max query between index 4 and index 7 is 5, denoted $rmq(values, 4, 7) = 5$. This is indeed at index 5 that there is the maximum value on the subvector $[1, 4, 3, 0]$.

In our case the vector is composed of the heights of the rectangles of the profile $heights = [h_1, h_2, \ldots, h_m]$. The two indices of the query are respectively:

- $j_1(i)$ is the index j of the rectangle $\langle a_j, b_j, h_j\rangle$ s.t. $\underline{s}_i \in [a_j, b_j[$.
- $j_2(i)$ is the index j of the rectangle $\langle a_j, b_j, h_j\rangle$ s.t. $\min(\underline{e}_i, \overline{s}_i) - 1 \in [a_j, b_j[$

The whole algorithm is given in Algorithm 1. An example of the filtering is given in Figure 4 (a). Notice that the task is pushed after a highest conflicting rectangle, which is not as good as the filtering of [14] (Figure 4 (b)).

Algorithm 1: MinLeftTTLinearTime(Ω,C)

Input: A set of tasks Ω, capacity C.
Output: *true* iff propagation failed, i.e. if the problem is infeasible.

```
1 initialize TT_Ω                                   // ⟨a_j, b_j, h_j⟩∀i ∈ {1...m}
2 if max_{j∈[1;m]} h_j > C then return true
3 heights ← [h_1, h_2, ..., h_m]
4 ∀i ∈ Ω compute j_1(i), j_2(i)
5 initialize rmq(heights)
6 for i ∈ Ω such that s < s̄ do
7 |   j* ← rmq(heights, j_1(i), j_2(i))
8 |   if h_{j*} + c_i > C then s_i ← b_j
9 return false
```

[1] a straightforward variant of the well-known range min query problem

Time Complexity. As in [6], we assume that all the time points are encoded with $w - bit$ integers and can thus be sorted in linear time. Given the sorted time points the profile TT_Ω can be computed in linear time using a sweep line algorithm, and all the indices $j_1(i), j_2(i)$ can be computed in linear time as well. The range min/max query is a well studied problem. Preprocessing in line 5 can be done in linear time, so that any subsequent query at Line 7 executes in constant time [7]. Thus, the whole algorithm executes in $O(n)$.

Discussion. Although the linear time complexity is an improvement over the $O(n \log n)$ algorithm introduced in [14], we believe that this result is more of theoretical rather than practical interest. The linear time range max query initialization hides non-negligible constants. The range max query used in Line 7 to reach this time complexity could be implemented by simply iterating on the rectangles from $j_1(i)$ to $j_2(i)$. On most problems, the interval $[\underline{s}_i, \min(\overline{s}_i, \underline{e}) - 1]$ only overlaps a few rectangles of the profile, so the $O(n)$ cost is not high in practice. Another limitation of the algorithm is that (as for the one of [14]) it may be called several times before reaching the fix-point (although it does not suffer from the slow convergence phenomenon described in [3]) either. It may be more efficient to continue pushing a task further to the right as in [2] rather than limiting ourselves to only one update per task per call to the procedure. This is precisely the objective of the algorithm introduced in the next section.

4 An efficient $O(n^2)$ time-table filtering

In this section, we introduce a practical algorithm for time-table filtering. It proceeds in two main phases: first the computation of the mandatory profile, then a per-task sweeping from left to right and from right to left. This modular design makes the algorithm simple, and its scalability comes from being able to exploit structures separately, for instance using sorting only on few tasks. We review the main phases of Algorithm 2 in execution order.

Building the profile. Line 2 computes the mandatory profile as a sequence of rectangles. We process only those tasks in Ω that have a mandatory part. We will try to reduce that set of tasks further ahead, reducing the work in this part.

Computing profile indices. Line 4 computes, for all unfixed tasks i, the profile rectangle containing $\underline{s}_i$. This value is saved between consecutive calls in a residue[2]; most of the time, it is still valid and we do not have to recompute it, if not, a dichotomic search is performed, at a cost of $O(\log n)$. Note that [2] sorts tasks by $\underline{s}$ to locate tasks on the profile, at a theoretical cost of $O(n \log n)$. Here we pay $O(\log n)$ only for tasks where the residue is invalid. Similarly, line 5 computes the rectangle containing the last point of i, $\overline{e}_i - 1$.

[2] this is similar to residual supports for AC algorithms [10]

Per-Task Sweeping. The loop in line 6 looks for left and right-conflicting rectangles for i linearly. The main difference with the global sweeping in [2] is that our method does not factorize sweeping according to height, wagering that the cost of moving tasks in height-ordered heaps is higher than that of pushing every task until no conflict remains. This main part has a worst case cost $O(n^2)$.

Fruitless fixed tasks removal. After the main loop, line 24 removes fixed tasks at profile extremities that can no longer contribute to pruning. This filtering is $O(n)$. Note that Ω is kept along the search tree using a reversible sparse-set [5].

Algorithm 2: ScalableTimeTable(Ω,C)

Input: A set of tasks Ω, capacity C.
Output: *true* iff propagation failed, i.e. if the problem is infeasible.

1 $\Omega_u \leftarrow \{i \mid \underline{s}_i < \overline{s}_i\}$ `// unfixed tasks`
2 initialize TT_Ω `//` $\langle a_j, b_j, h_j \rangle, \forall j \in \{1 \dots m\}$
3 **if** $\max_{j \in [1;m]} h_j > C$ **then return** true
4 $\forall i \in \Omega_u$, compute $j_1(i)$ such that $\underline{s}_i \in [a_{j_1(i)}; b_{j_1(i)}[$
5 $\forall i \in \Omega_u$, compute $j_2(i)$ such that $\overline{e}_i - 1 \in [a_{j_2(i)}; b_{j_2(i)}[$
6 **for** $i \in \Omega_u$ **do**
7 $\quad j \leftarrow j_1(i)$
8 $\quad \underline{s}^*_i \leftarrow \underline{s}_i$
9 $\quad$ **while** $j \leq m$ *and* $a_j < \min(\underline{s}^*_i + d_i, \overline{s}_i)$ **do**
10 $\quad\quad$ **if** $C - c_i < h_j$ **then**
11 $\quad\quad\quad \underline{s}^*_i \leftarrow \min(b_j, \overline{s}_i)$ `// j in left-conflict`
12 $\quad\quad j \leftarrow j + 1$
13 $\quad$ **if** $\underline{s}^*_i > \underline{s}_i$ **then** $\underline{s}_i \leftarrow \underline{s}^*_i$
14
15 $\quad j \leftarrow j_2(i)$
16 $\quad \overline{e}^*_i \leftarrow \overline{e}_i$
17 $\quad$ **while** $j \geq 1$ *and* $b_j \geq \max(\overline{e}^*_i - d_i, \underline{e}_i)$ **do**
18 $\quad\quad$ **if** $C - c_i < h_j$ **then**
19 $\quad\quad\quad \overline{e}^*_i \leftarrow \max(a_j, \underline{e}_i)$ `// j in right-conflict`
20 $\quad\quad j \leftarrow j - 1$
21 $\quad$ **if** $\overline{e}^*_i < \overline{e}_i$ **then** $\overline{e}_i \leftarrow \overline{e}^*_i$
22 $s^u_{\min} \leftarrow \min_{i \in \Omega_u} \underline{s}_i$
23 $e^u_{\max} \leftarrow \max_{i \in \Omega_u} \overline{e}_i$
24 $\Omega \leftarrow \Omega \setminus \{i \in Omega \mid \overline{e}_i \leq s^u_{\min} \vee e^u_{\max} \leq \underline{s}_i\}$
25 **return** false

5 Experiments

We have tested our ScalableTimeTable filtering against the time-table filtering of or-tools [12], Choco3 [4] and Gecode [9] solvers. The algorithm in Choco3

and Gecode solver is the one of [2]. Similarly to our algorithm, or-tools also builds the time-table structure before filtering. To the best of our knowledge, no implementation of Letort et al [11] algorithm is publicly available. We have thus implemented the heap-based variant of the algorithm, faster in practice, with the same quality standard as our new ScalableTimeTable [13]. In the models used for this experiment, cumulative propagators enforce only the resource constraint, precedences are enforced by separate propagators

We have generated randomly n (ranging from 100 to 12800) tasks with duration between 200 and 2000 and heights between 1 and 40. The capacity is fixed to 100. The search is a simple greedy heuristic selecting the current tasks with the smallest earliest possible start, hence there is no backtrack. The search finishes when all the tasks have been placed. This simple experimental setting guarantees that every solver has exactly the same behavior. The results are given on Figure 5. As can be seen, the time-table implementation of Choco3 and Gecode are quickly not able to scale well for more than 1600 tasks. The algorithm of or-tools, Letort et al and our new algorithm are still able to handle 12800 tasks. Surprisingly, our algorithm outperforms the one of Letort et al despite its simplicity.

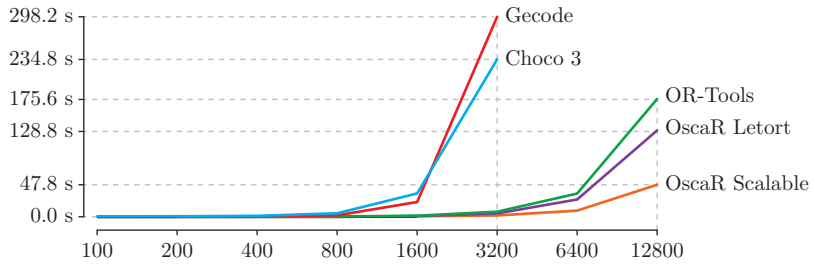

Fig. 5: Comparison of Time-Table implementations.

6 Conclusion

We have introduced an $O(n)$ time-table filtering using range min queries. We believe that the usage of range min query may be useful for subsequent research on scheduling, for instance in time-table disjunctive reasoning [8]. We introduced simple but scalable $O(n^2)$ filtering for the cumulative constraint. Our results show that despite its simplicity, it outperforms current implementations of time-table constraints in some open-source solvers and also the recent synchronized sweep algorithm. The resources related to this work are available here `http://bit.ly/cumulativett`.

References

1. Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.
2. Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In *CP*, Lecture Notes in Computer Science, pages 63–79, 2002.
3. Lucas Bordeaux, Youssef Hamadi, and Moshe Y Vardi. An analysis of slow convergence in interval propagation. In *Principles and Practice of Constraint Programming–CP 2007*, pages 790–797. Springer, 2007.
4. Xavier Lorca Charles Prud'homme, Jean-Guillaume Fages. Choco3 documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.
5. Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.
6. Hamed Fahimi and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.
7. Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *Combinatorial Pattern Matching*, pages 36–48. Springer, 2006.
8. Steven Gay, Renaud Hartert, and Pierre Schaus. Time-table disjunctive reasoning for the cumulative constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR15)*. Springer, 2015.
9. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
10. Christophe Lecoutre, Fred Hemery, et al. A study of residual supports in arc consistency. In *IJCAI*, volume 7, pages 125–130, 2007.
11. Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. A scalable sweep algorithm for the cumulative constraint. In *Principles and Practice of Constraint Programming*, pages 439–454. Springer, 2012.
12. Or-tools Team. or-tools: Google optimization tools, 2015. Available from `https://developers.google.com/optimization/`.
13. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
14. Pierre Ouellet and Claude-Guy Quimper. Time-table extended-edge-finding for the cumulative constraint. In *Principles and Practice of Constraint Programming*, pages 562–577. Springer, 2013.
15. Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 230–245. Springer, 2011.