Biclique Factorisation of Explanations for a Sparser Implication Graph in Lazy Clause Generation

Sulian Le Bozec-Chiffoleau, Charles Prud'Homme

Biclique Factorisation of Explanations for a Sparser Implication Graph in Lazy Clause Generation

Sulian Le Bozec-Chiffoleau[1]
IMT Atlantique, LS2N, UMR CNRS 6004, F-44307 Nantes, France

Charles Prud'homme
IMT Atlantique, LS2N, UMR CNRS 6004, F-44307 Nantes, France

Abstract

In the context of lazy clause generation, we propose factorising bicliques in the implication graph to reduce the associated overhead. This process consists of *(1)* identifying a set P of prunings that share a common subset L of literals in their reason, *(2)* introducing an artificial literal f, called a factor, and *(3)* adding the arcs $(L \times f) \cup (f \times P)$ to the implication graph instead of the arcs $L \times P$. When P and L are large, biclique factorisation significantly reduces the number of generated arcs. This technique offers two main advantages: *(1)* faster explanation generation and *(2)* a sparser implication graph, leading to more efficient conflict analysis and reduced memory usage. We provide tight worst-case bounds on both the time complexity of explanation generation and the maximum number of generated arcs for two global constraints: ALLDIFFERENT and CUMULATIVE. Experiments were also conducted to evaluate and validate our approach.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases lazy clause generation, explanations, implication graph, graph compression, biclique factorisation, global constraint, alldifferent, cumulative

Acknowledgements The authors thank Nicolas Beldiceanu, Xavier Lorca and Gilles Simonin for their advices regarding this work.

1 Introduction

Designing efficient Constraint Programming (CP) [35] solvers is often a matter of finding an optimal trade-off between inference time and inference strength. Lazy Clause Generation (LCG) [30] is one inference technique which is no exception to this rule. LCG is a hybrid solving framework that combines CP with clause learning from *boolean satisfiability* (SAT) [9]. It extends classical propagation by recording explanations for domain reductions (prunings); these explanations form an implication graph that is analysed during conflict analysis to derive a learned clause. This integration enables the solver to benefit from powerful global constraints while exploiting nogood learning and non-chronological backtracking, significantly improving performance on combinatorial problems. However, LCG may incur a non-negligible overhead due to literal creation, explanation generation, and implication graph traversal during conflict analysis. This has motivated the introduction of *lazy explanations* [20], which defer explanation generation until conflict analysis, thereby avoiding the computation of explanations that are not useful for clause learning.

In this paper, we propose another method to reduce this overhead: *biclique factorisation*. This idea stems from the observation that many prunings may share a common subset of literals in their reasons, leading to a high worst-case time complexity for explanation generation—especially for large global constraints—as well as to a locally, and potentially globally, very dense implication graph. Such density slows down conflict analysis and increases the memory footprint. Biclique factorisation originates from a graph compression technique

[1] corresponding author

and consists of *(1)* identifying a set P of prunings that share a common subset L of literals in their reason, *(2)* introducing an artificial literal f, called a factor, and *(3)* adding the arcs $(L \times f) \cup (f \times P)$ to the implication graph instead of the arcs $L \times P$. This significantly reduces the number of generated arcs while preserving reachability relationships. This technique offers two main advantages: *(1)* faster explanation generation and *(2)* a sparser implication graph, leading to more efficient conflict analysis and reduced memory usage.

We implement biclique factorisation for two core global constraints: ALLDIFFERENT [34] and CUMULATIVE [1]. We establish tight worst-case bounds on both the time complexity of explanation generation and the maximum number of generated arcs. Experiments are conducted using `Choco-solver` [33] to evaluate and validate the proposed approach. In particular, we highlight the differences in the number of generated arcs in practice with and without biclique factorisation, and show that memory requirements can be significantly affected. Although explanation generation is usually not the main bottleneck in resolution and therefore does not have a major impact on overall performance, we observed noticeable differences in total computation time in some cases.

Section 2 introduces the basics of constraint programming and lazy clause generation. Section 3 presents the general principle of biclique factorisation of explanations. Sections 4 and 5 detail how to efficiently factorise explanations for ALLDIFFERENT and CUMULATIVE. Finally, Section 6 reports the experimental evaluation conducted with `Choco-solver`.

2 Preliminaries

2.1 Constraint Programming

Constraint Programming (CP) [35] solves combinatorial problems by modelling variables with finite domains subject to constraints and combining systematic search with inference to efficiently explore the solution space. Modern solvers rely on *backtracking search* interleaved with *constraint propagation* [8], where *propagators* prune infeasible variable-value pairs to maintain (or approximate) local consistency. *Generalised arc consistency* (GAC) ensures that every value of each variable in a constraint's scope participates in at least one assignment satisfying the constraint. *Bound consistency* (BC) is a weaker form of consistency that considers the domains as intervals: it ensures that the lower and upper bounds of each variable are consistent, i.e., each bound can participate in at least one assignment satisfying the constraint. Constraint propagation iteratively calls propagators until the *fix point* is reached, reducing the search space while preserving completeness. A key design trade-off in CP is between inference strength—the amount of pruning achieved—and its computational cost: stronger propagation can shrink the search tree but is costlier per node, whereas weaker propagation is cheaper but may lead to more extensive search.

2.2 Lazy Clause Generation

Lazy clause generation (LCG) [17, 30] is a hybrid technique combining CP and *boolean satisfiability* (SAT) solvers. It adapts *conflict-driven clause learning* (CDCL) from SAT [9] by encoding a boolean variant of the domains and learning clauses upon failure to avoid repeating the same failing subtrees. The encoding uses *value literals* ($[\![x = v]\!]$, $[\![x \neq v]\!]$) and *bound literals* ($[\![x \leq v]\!]$, $[\![x \geq v]\!]$) for each variable x and each value v in its initial domain. A *clause*—a disjunction of literals—is learned whenever a *conflict* arises, i.e., when the current partial assignment is unsatisfiable. This clause represents a *nogood* (a partial assignment and subset of constraints that necessarily lead to a conflict) and is added to a SAT formula

in *conjunctive normal form* (CNF). *Unit propagation* [9] is then performed over this formula to derive further inferences along the search.

To enable clause learning, the filtering performed during constraint propagation is *explained*: for each pruning p—typically an *instantiation* ($[\![x = v]\!]$), a *value removal* ($[\![x \neq v]\!]$), or a *bound update* ($[\![x \leq v]\!]$ or $[\![x \geq v]\!]$)—a conjunction of literals, called the *reason* of p and denoted $R(p)$, is computed. The literals in $R(p)$ are all set to true at the moment of the pruning, and the logical implication $R(p) \longrightarrow p$ holds at any time. This information is stored in the *implication graph*, a *directed acyclic graph* (DAG) dynamically updated along the search, where the predecessors $N^-(v)$ of a node v form the reason of the corresponding literal. By definition, search *decisions* have no reason and are thus source nodes of the DAG. When a conflict arises, its reason is computed and the implication graph is analysed to find a vertex-cut separating decision nodes from the conflict node—this is *conflict analysis*. The vertex-cut forms a nogood [42], which is then converted into a learned clause. The *first unique implication point* (1-UIP) vertex-cut is considered the best in practice [42, 13], as it can be found in linear time in the size of the explored subgraph [28] while enabling *non-chronological backtracking*. Modern LCG solvers such as `Chuffed` [11], `OR-Tools` [32] and `Choco-solver` [33] implement 1-UIP learning. *Clause minimisation* [38], i.e., removing redundant literals, can further strengthen the learned nogoods.

Another version of lazy clause generation generates explanations *lazily* [20], in contrast to the *eager* approach described above. The motivation is that many explanations computed along the search are never used during conflict analysis, wasting time. In the lazy approach, sufficient information is stored when a pruning occurs so that explanations can be generated on demand during conflict analysis, ensuring that all generated explanations are actually used. Although this delayed generation may incur some overhead, in practice it is typically outweighed by the savings from avoiding unused explanations [20].

In this paper, we do not use lazy explanations but eagerly generate them. The combination of lazy explanations and biclique factorisation constitutes a direction for future work.

3 Biclique Factorisation of Explanations

To reduce the overhead of lazy clause generation—particularly when global constraints produce many explanations—we propose to *factorise the bicliques* that would otherwise be generated in the implication graph. This overhead arises in particular because multiple prunings may share common subsets of literals in their reasons, leading to a large number of arcs and a dense implication graph.

Given a directed graph $G = (V, A)$, let S_L and S_R be two disjoint subsets of V such that $S_L \times S_R \subseteq A$. The subgraph $(S_L \cup S_R, S_L \times S_R)$ is then a *complete directed bipartite graph*, called a (directed) *biclique* of G. Factorising this biclique means creating a new vertex f, called the *factor*, and replacing $(S_L \cup S_R, S_L \times S_R)$ in G by the directed graph $(S_L \cup \{f\} \cup S_R, (S_L \times \{f\}) \cup (\{f\} \times S_R))$. We then say that G has been *factorised*. By doing this, the number of arcs in G is reduced by $(|S_L| \times |S_R|) - (|S_L| + |S_R|)$ while preserving the reachability relationships. Biclique factorisation is then interesting as soon as both S_L and S_R contain at least 2 vertices and one of them (or both) contains 3 or more vertices.

Let us go back to LCG and the implication graph. Let P be a set of prunings such that $R(p) = L \cup R_p$ for each $p \in P$, where L is the common subset of literals in the reason of the prunings and R_p denotes the other literals in addition to L in the reason of p (potentially $R_p = \emptyset$). Instead of generating all arcs $(\ell, p) \in L \times P$ in the implication graph, we add the artificial node f, the factor, and we create an arc (ℓ, f) for each $\ell \in L$ and an arc (f, p) for

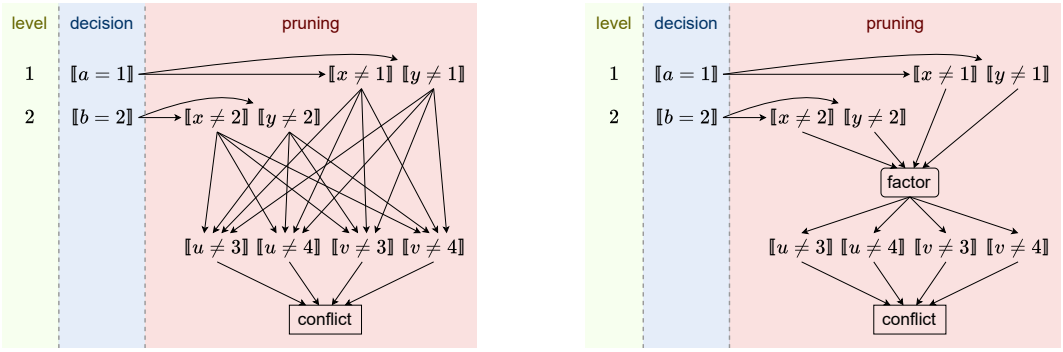

Figure 1 The original (left) and a factorised (right) implication graph at some point of the search for the following CSP. Variables: $a \in \{1, \ldots, 10\}$, $b \in \{2, \ldots, 10\}$, $u \in \{3, \ldots, 6\}$, $v \in \{3, \ldots, 6\}$, $x \in \{1, \ldots, 4\}$ and $y \in \{1, \ldots, 4\}$. Constraints: $\textsc{alldifferent}(a, b, u, v, x, y)$ and $[\![u = 3]\!] \vee [\![u = 4]\!] \vee [\![v = 3]\!] \vee [\![v = 4]\!]$. The explanations for $\textsc{alldifferent}$ are the ones described in [14].

each $p \in P$ (Figure 1). The arcs $R_p \times \{p\}$ for each $p \in P$ are generated as usual.

When a pruning occurs, the clause expressing the explanation is added to the CNF formula to maintain the consistency between the boolean encoding and the actual domains. This clause will be removed from the formula upon backtracking. Without biclique factorisation, the following clauses are added to the current CNF formula:

$$(p \vee \bigvee_{\ell \in L} \neg \ell \vee \bigvee_{r \in R_p} \neg r) \quad \forall p \in P$$

With biclique factorisation, the following clauses are added instead:

$$(f \vee \bigvee_{\ell \in L} \neg \ell) \quad \text{and} \quad (p \vee \neg f \vee \bigvee_{r \in R_p} \neg r) \quad \forall p \in P$$

The same updates will be made by unit propagation. This formulation reminds us of common subclause elimination [5].

Biclique factorisation has already been studied in many fields including circuit design [27], linear boolean operators [24] and graph compression [16]. However, to the best of our knowledge, no such technique was implemented nor described within the CP community for LCG [30, 17, 20, 39, 12], nor within the SAT community for CDCL [42, 6, 3, 9, 2].

3.1 Conflict Analysis in the Factorised Implication Graph

The whole point of constructing the implication graph is to analyse it when a conflict occurs to deduce clauses to learn. The question that naturally arises is then the following: Can we adapt the original conflict analysis techniques to the factorised implication graph ? The answer is yes and we shall see why. The difference brought by biclique factorisation is the way to get access to the reason of a literal. In the presence of factors, one must recursively expand the factors to retrieve the reason, as described in Algorithm 1.

The 1-UIP procedure [28], as well as *local* [6] and *recursive* [38] minimisation are graph exploration procedures that mark the nodes based on certain properties and that traverse their incoming arcs at most once. Therefore, they run in linear time in the size of the explored subgraph of the implication graph. They can easily be adapted to biclique factorisation by marking the factors as well so that they are not expanded several times when processing the reason of different literals. By doing this, the algorithms will return the same output but in

Algorithm 1 The algorithm skeletons to process the reason of a literal during conflict analysis in the original (left) and factorised (right) implication graph. $N^-(v)$ are the predecessors of a node v.

```
procedure processReason(v)
    for u ∈ N⁻(v) do
        Do something with u
```

```
procedure processReason-F(v)
    for u ∈ N⁻(v) do
        if u is a factor then
            processReason-F(u)
        else
            Do something with u
```

linear time in the size of the explored subgraph of the factorised implication graph, which is sparser. Those three algorithms adapted to biclique factorisation are described in Section A.

Note that the order in which literals appear in the learned clause may be different when using biclique factorisation, depending on the way explanations are generated by propagators and on the way conflict analysis constructs the clause. This may lead to different search trees because unit propagation may not find the same conflict due to different watched literals. In this paper, there will be no difference in the order of literals for the considered propagators and explanations when using biclique factorisation, so the search tree will remain the same.

3.2 Implementation for the ALLDIFFERENT and CUMULATIVE constraints

In this paper, we implement biclique factorisation for two core global constraints widely used in practice and extensively studied in the literature: ALLDIFFERENT [34] (Section 4) and CUMULATIVE [1] (Section 5). We show how to identify bicliques that arise in the implication graph and how to factorise them efficiently. This leads to improved time complexities for explanation generation for these two constraints. We analyse the worst-case time complexity of explanation generation only, excluding the rest of the propagator and focusing solely on the additional overhead introduced by explanations. We provide theoretical results for a single propagator call as well as along any branch of the search tree, both with and without biclique factorisation. In particular, we establish upper bounds on the worst-case time complexity and prove their tightness by exhibiting families of instances that attain them. All these complexity results are summarised in Table 1.

Upper bounds are obtained by bounding the computation time of explanations, which simultaneously provides a bound on the output size, which corresponds to the number of generated arcs. Inversely, we derive lower bounds on the running time by establishing lower

Worst-Case Time Complexity →	**Single Call**		**Along a Branch**	
Propagator | Explanations ↓	Original	Factorised	Original	Factorised
GAC ALLDIFFERENT | Graph	$\Theta(r^3\, d)$	$\boldsymbol{\Theta(r\, d)}$	$\Theta(r^3\, d)$	$\boldsymbol{\Theta(r^2\, d)}$
Time-Table CUMULATIVE | Big-Step	$\Theta(n^3)$	$\boldsymbol{\Theta(n^2)}$	$\Theta(n^2\,\lvert\Delta\rvert)$	$\Theta(n\,\lvert\Delta\rvert)$?
Time-Table CUMULATIVE | Pointwise	$\Theta(n^2\,\lvert\Delta\rvert)$	$\boldsymbol{\Theta(n\,\lvert\Delta\rvert)}$	$\Theta(n^2\,\lvert\Delta\rvert)$	$\Theta(n\,\lvert\Delta\rvert)$?

Table 1 Worst-case time complexity of generating explanations for ALLDIFFERENT [14] and CUMULATIVE [37]. *i)* For ALLDIFFERENT, there are r variables and initially d distinct values. *ii)* For CUMULATIVE, the complexity of a single call corresponds to one iteration of the time-table propagator, not for reaching the fix point. The number of tasks is n and the time interval is Δ. A question mark '?' means the claim requires further investigation.

bounds on the number of generated arcs. Therefore, these tight bounds hold for both the worst-case time complexity and the maximum number of generated arcs.

4 The ALLDIFFERENT Constraint

Given a set of integer variables, the ALLDIFFERENT constraint ensures these variables are assigned to a different value (see [21] for a survey):

$$\text{ALLDIFFERENT}(x_1, \ldots, x_r) \quad \equiv \quad x_i \neq x_j \;\; \forall (i,j) \in \{1, \ldots, r\}^2 \mid i \neq j$$

The number of variables within the scope of the constraint is denoted by r and the total number of distinct values from the variable domains is denoted by d. We assume that $r \leq d$, otherwise the constraint can not be satisfied.

The state-of-the-art propagators enforcing GAC on ALLDIFFERENT [21, 25] are based on Régin's algorithm [34], which is a three-step procedure:

1. Find a *maximum matching* in the *variable-value graph*;
2. Compute the *strongly connected components* (SCCs) of the *residual graph*;
3. Prune all *variable-value pairs* that do not belong to the same SCC.

Specific explanations for ALLDIFFERENT were introduced in [14]. These explanations are based on the variable-value and residual graphs, we thus call them *graph explanations*.

The variable-value graph is the undirected bipartite graph whose vertices are the variables from the scope and the values from their domains. An edge links together a variable x and a value v whenever v belongs to the domain of x. A maximum matching is computed by iteratively finding *augmenting paths* in the variable-value graph from unmatched variables. An *alternating path* is a path for which edges alternatively belong or not to the current matching, and an augmenting path is an alternating path whose both extremities are unmatched. The size of the current matching is then increased by one by switching the matching membership of the edges forming the augmenting path. The constraint is unsatisfiable whenever no augmenting path can be found from an unmatched variable x_u. The explanation of the conflict κ proposed in [14] is then all missing edges between variables reachable from x_u by an alternating path and values that are not (see Figure 2-left). Let $A_{var}(x_u)$ (resp. $A_{val}(x_u)$) denote the set of variables (resp. values) reachable from x_u by an alternating path:

$$\bigwedge_{x \in A_{var}(x_u)} (\llbracket x \geq v_{min} \rrbracket \wedge \llbracket x \leq v_{max} \rrbracket \wedge \bigwedge_{v \in [v_{min}, v_{max}] \mid v \notin A_{val}(x_u)} \llbracket x \neq v \rrbracket) \longrightarrow \kappa$$

Where v_{min} and v_{max} are respectively the minimum and maximum values from $A_{val}(x_u)$. Modern LCG solvers [11, 33] use bound literals to spare many value literals.

The residual graph is a directed variant of the variable-value graph where edges are directed from variables and towards values, except for those in the maximum matching which are in the other direction. An artificial vertex t is added along with an arc (t, v_m) for each matched value v_m and an arc (v_u, t) for each unmatched value v_u. Whenever a variable x_p and a value v_p from the domain of x_p do not belong to the same SCC, the pair (x_p, v_p) is pruned, i.e. v_p is removed from the domain of x_p. However, if v_p is in a singleton SCC and if x_p and v_p are matched together, then x_p is assigned to v_p instead. Let $\texttt{scc}(v_p)$ denote the SCC that contains v_p. The graph explanation for the pruning $\llbracket x_p \neq v_p \rrbracket$ is made of all missing outgoing arcs from $\texttt{scc}(v_p)$ (see Figure 2-right). Let $X_{\texttt{scc}(v_p)}$ (resp. $V_{\texttt{scc}(v_p)}$) denote the set of variables (resp. values) in $\texttt{scc}(v_p)$.

$$\bigwedge_{x \in X_{\texttt{scc}(v_p)}} (\llbracket x \geq v_{min} \rrbracket \wedge \llbracket x \leq v_{max} \rrbracket \wedge \bigwedge_{v \in [v_{min}, v_{max}] \mid v \notin V_{\texttt{scc}(v_p)}} \llbracket x \neq v \rrbracket) \longrightarrow \llbracket x_p \neq v_p \rrbracket$$

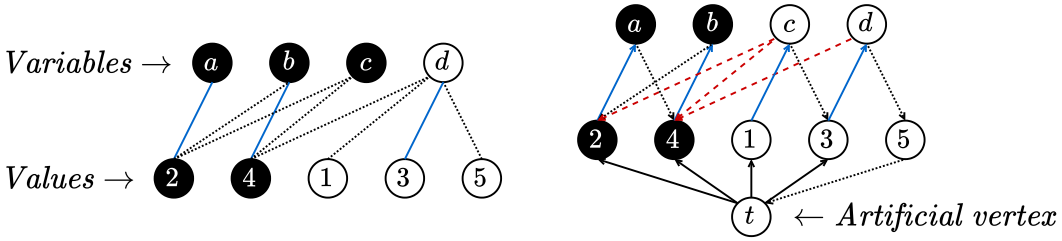

Figure 2 LEFT: an illustration of a variable-value graph. The current matching is $\{(a,2),(b,4),(d,3)\}$ and there is no augmenting path starting from variable c. The reason of the conflict is made of the visited (black) variables paired with the unvisited (white) values, $R(\kappa) = \{[\![a \geq 2]\!], [\![a \leq 4]\!], [\![a \neq 3]\!], [\![b \geq 2]\!], [\![b \leq 4]\!], [\![b \neq 3]\!], [\![c \geq 2]\!], [\![c \leq 4]\!], [\![c \neq 3]\!]\}$. RIGHT: an illustration of a residual graph. The maximum matching is $\{(a,2),(b,4),(c,1),(d,3)\}$ and there are two SCCs (black and white). The variable-value pairs $\{(c,2),(c,4),(d,4)\}$ are pruned. Because their values belong to the same SCC they have the same reason, which is $\{[\![a \geq 2]\!], [\![a \leq 4]\!], [\![a \neq 3]\!], [\![b \geq 2]\!], [\![b \leq 4]\!]\}, [\![b \neq 3]\!]\}$.

Where v_{min} and v_{max} are respectively the minimum and maximum values from $V_{\texttt{scc}(v_p)}$. In case $\texttt{scc}(v_p) = \{v_p\}$, the reason is made of the sole instantiation literal $[\![x_m = v_p]\!]$ instead, where x_m is the variable matched to v_p.

Finding a maximum matching can be done in $O(r^{1.5}\, d)$ time thanks to Hopcroft-Karp's algorithm [23]. The strongly connected components are computed in $O(r\, d)$ time by Tarjan's algorithm [41] and the domains are then pruned in $O(r\, d)$ time. When using an incremental matching [34], the time spent computing a maximum matching is $O(r^2\, d^2)$ along a branch, because a pair from the matching can be broken at most $r \times d$ times and finding an augmenting path takes $O(r\, d)$ time. Also, computing the SCCs and pruning the domains takes $O(r^2\, d^2)$ time along a branch, because the propagator is called at most $r \times d$ times along a branch.

We provide tight bounds on the worst-case time complexity of generating graph explanations for ALLDIFFERENT both for a single propagator call and along a branch of the search tree, with and without biclique factorisation. The upper-bound proofs are provided in this section while the lower-bound proofs are in the appendix (Section B).

▶ **Lemma 1.** *Along a branch of the search tree, a GAC propagator for* ALLDIFFERENT *performs at most* $r \times (r-1)$ *value removals.*

Proof. All unmatched values that appear in the domain of at least one variable belong to $\texttt{scc}(t)$, which is the source of the SCC DAG and therefore has no incoming arcs. Consequently, a value can be removed only after it becomes matched and moves to an SCC distinct from $\texttt{scc}(t)$. Moreover, once a value is matched and leaves $\texttt{scc}(t)$, it remains matched and cannot return to $\texttt{scc}(t)$ further down the same branch. Since the number of matched values is bounded by the size of the maximum matching, which is r, at most r such values can appear. Furthermore, the variable-value pairs making the matching are not pruned, so at most $r-1$ values can be removed from the domain of a given variable along a branch. ◀

▶ **Theorem 2.** *The worst-case time complexity of generating graph explanations is* $\Theta(r^3\, d)$, *both for a single propagator call and along a branch of the search tree.*

Upper-bound proof. *(i)* The graph explanation of a conflict is computed in $O(r\, d)$ time by scanning the universe of values for each variable visited during the failed attempt to find an augmenting path. At most one conflict can arise along a branch.

(ii) By Lemma 1, the number of pruned variable-value pairs along a branch is $O(r^2)$. Also, a graph explanation for a pruned pair (x, v) is computed in $O(r\,d)$ time by scanning the universe of values for each variable in $\texttt{scc}(v)$.

Therefore, the worst-case time complexity of generating graph explanations is $O(r^3\,d)$ along a branch, and then also $O(r^3\,d)$ for a single propagator call. ◀

4.1 Factorised Explanations for ALLDIFFERENT

In this section, we show how to factorise graph explanations [14] for the ALLDIFFERENT constraint. First, let us identify the bicliques that would appear in the implication graph.

▶ **Lemma 3.** *Let (x, v) and (x', v') be two variable-value pairs pruned by a GAC propagator for* ALLDIFFERENT *and let us consider graph explanations.*

$$\texttt{scc}(v) = \texttt{scc}(v') \implies R([\![x \neq v]\!]) = R([\![x' \neq v']\!])$$
$$\texttt{scc}(v) \neq \texttt{scc}(v') \implies R([\![x \neq v]\!]) \cap R([\![x' \neq v']\!]) = \emptyset$$

Proof. The explanation of a pruning depends only on the SCC to which the value of the pruned pair belongs. Therefore, if two pruned pairs (x, v) and (x', v') satisfy $\texttt{scc}(v) = \texttt{scc}(v')$, they are associated with the same SCC and thus have identical reasons.

For a given SCC c, the corresponding reason consists of the non-arcs from the variables in c to values outside c. Since two distinct SCCs contain disjoint sets of variables, the sets of literals forming their respective reasons are also disjoint. ◀

Thanks to Lemma 3, whose claim was already partly mentioned in [20], we can identify at no extra cost which prunings will have the same reason, and thus the bicliques that would be generated in the implication graph during a single call to the propagator. Therefore, we can factorise them without any overhead as described in Algorithm 2.

Algorithm 2 Generate a factorised graph explanation for the value removal $[\![x \neq v]\!]$

if $f_{\texttt{scc}(v)}$ is not already in the implication graph **then**
  Add $f_{\texttt{scc}(v)}$ with the reason associated with $\texttt{scc}(v)$ as predecessors
Add the arc $f_{\texttt{scc}(v)} \to [\![x \neq v]\!]$

When the reason is made of a unique literal, typically an instantiation $[\![x' = v']\!]$, there is no need to create the factor as an intermediary.

▶ **Theorem 4.** *The worst-case time complexity of generating factorised graph explanations is $\Theta(r\,d)$ for a single propagator call.*

Upper-bound proof. *(i)* The graph explanation of a conflict is computed in $O(r\,d)$ time.

(ii) At most $r \times (r-1)$ pairs can be pruned (Lemma 3), and every pruned pair (x, v) has exactly one predecessor in the factorised implication graph: the factor $f_{\texttt{scc}(v)}$. Thus, computing the predecessors for all the pruned pairs takes $O(r^2)$ time.

(iii) At most one factor is created per SCC. The predecessors of a factor f_c associated with the SCC c are computed by scanning the universe of values for each variable in c. This is done in $O(|X_c|\,d)$ time. The sets of variables of two distinct SCCs are disjoints so $\sum_{c \in SCCs} |X_c| = r$. Thus, computing the predecessors for all the factors takes $O(r\,d)$ time.

Therefore, the worst-case time complexity of generating factorised graph explanations is $O(r\,d)$ for a single propagator call. ◀

Thanks to Theorem 4, generating graph explanations is no longer the theoretical bottleneck in terms of worst-case time complexity for a single propagator call for enforcing GAC on and explaining ALLDIFFERENT.

▶ **Lemma 5.** *Along a branch of the search tree, the number of calls to a GAC propagator for* ALLDIFFERENT *during which pruning occurs is less than* r*.*

Proof. Let us follow the evolution along a branch of the number n_{var} of connected components (CCs) in the variable-value graph that contain at least one variable.

Between two calls to the propagator. Let us consider two consecutive propagator calls C_a and C_b, and let $n_{var}^{a,end}$ (resp. $n_{var}^{b,start}$) be n_{var} at the end of call C_a (resp. at the start of call C_b). Call C_b happens later than call C_a on the same branch, some edges may then have been deleted but none has been added to the variable-value graph, so $n_{var}^{a,end} \leq n_{var}^{b,start}$.

Between the start and the end of the same propagator call. Let C be one call to the propagator and let n_{var}^{start} (resp. n_{var}^{end}) be n_{var} at the start of call C (resp. at the end of call C). If no pruning occurs during call C, then no changes are made in the variable-value graph and $n_{var}^{start} = n_{var}^{end}$. Let us consider the case where a pair (x, v) is pruned.

First, all inter-SCC arcs in the residual graph are pruned by the propagator. So, the CC in the variable-value graph that contained both x and v is split into at least two new CCs once the propagator has ended: one of the new ones contains x, and another v.

Second, the unmatched values all belong to `scc(`t`)`, which is the source of the DAG of SCCs. So, all pruned variable-value pairs have their value matched. Also, the pairs making the matching are not pruned, so a matched value always remain in the same CC as its matched variable. Then, both x and v will belong to a CC that contains at least one variable. Therefore we have $n_{var}^{start} < n_{var}^{end}$, meaning that n_{var} strictly increases.

Finally, n_{var} is bounded by 1 and r by definition, so we deduce that the number propagator calls during which pruning occurs along a branch is less than r. ◀

▶ **Theorem 6.** *The worst-case time complexity of generating factorised graph explanations is* $\Theta(r^2\, d)$ *along a branch of the search tree.*

Upper-bound proof. If no conflict arises nor any pruning occurs during a propagator call, then no time is spent generating graph explanations. Moreover, at most one conflict can arise along a branch, and its reason is computed in $O(r\, d)$ time. Therefore by Theorem 4 and Lemma 5, the worst-case time complexity of generating factorised graph explanations is $O(r^2\, d)$ along a branch. ◀

By Theorem 4 and Theorem 6, biclique factorisation allows for a faster explanation generation for ALLDIFFERENT, as well as a reduced number of generated arcs in the implication graph as explained in Section 3.2, both for a single propagator call and along a branch.

5 The CUMULATIVE Constraint

A task i is made of an integer variable s_i defining its starting time, as well as a positive integer d_i corresponding to its duration, and a positive integer h_i called the height and representing the resource consumption of the task. Given a set of tasks, denoted as *Tasks*, and a capacity c, the CUMULATIVE constraint ensures the cumulated resource consumption never exceeds the capacity:

$$\text{CUMULATIVE}(\mathit{Tasks}, c) \quad \equiv \quad \sum_{i \in \mathit{Tasks} \;\mid\; s_i \leq t_p < s_i + d_i} h_i \leq c \quad \forall t_p \in \Delta$$

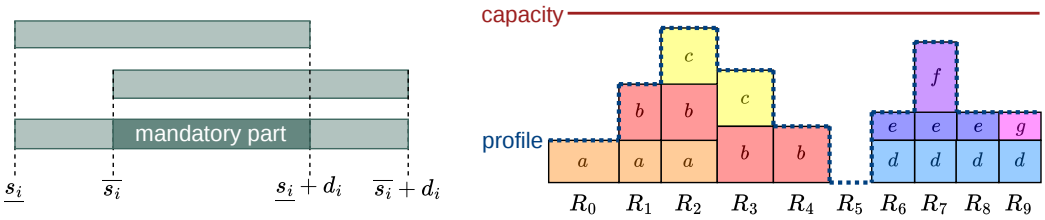

Figure 3 LEFT: an illustration of the mandatory part of a task i. RIGHT: an illustration of the mandatory part of tasks a to g, as well as the corresponding profile and rectangle decomposition.

Where $\Delta = \{min_{i \in Tasks}(\underline{s_i}), \ldots, max_{i \in Tasks}(\overline{s_i} + d_i) - 1\}$ is the discrete time interval over which the tasks can be performed. The number of tasks is denoted by n.

Because the *resource-constrained project scheduling problem* (RCPSP) is NP-Hard [10], enforcing bound consistency on CUMULATIVE is also NP-Hard. So, many filtering rules were proposed for CUMULATIVE, and a filtering rule that received considerable attention in the literature is the *time-table* propagator [7, 26, 31, 19]. In time-table propagation, the resource profile induced by the mandatory parts of the tasks can be represented as a step function over time (Figure 3). For a task i with earliest start time $\underline{s_i}$, latest start time $\overline{s_i}$ and duration d_i, the mandatory part is the time interval $[\overline{s_i}, \underline{s_i} + d_i]$ whenever $\overline{s_i} < \underline{s_i} + d_i$. The resource profile obtained by aggregating these mandatory parts is piecewise constant and changes value only at event points corresponding to the start or end of mandatory parts. This profile can be viewed as a decomposition of rectangles $(R_j)_{j \in J}$ in the time–resource plane. A rectangle R_j is defined over a time interval $[a_j, b_j]$ and is made of all tasks whose mandatory part contains $[a_j, b_j]$: $Tasks(R_j) = \{i \in Tasks \mid \overline{s_i} \leq a_j \wedge \underline{s_i} + d_i \geq b_j\}$. The height of R_j is then the cumulated mandatory resource consumption over $[a_j, b_j]$: $h_j = \sum_{\ell \in Tasks(R_j)} h_\ell$. The number of rectangles can not exceed $2n+1$. During propagation, these rectangles are used to update the bounds of the start times: for each task i and for each rectangle R_j, the algorithm verifies whether i would overload R_j. More precisely, it verifies whether scheduling i at $\underline{s_i}$ (resp. $\overline{s_i}$) would make i overlap with $[a_j, b_j]$ and cause the total resource consumption to exceed the capacity; if so, s_i (resp. $s_i + d_i$) must be greater than or equal to b_j (resp. less than or equal to a_j). Rectangles are processed from left to right to update lower bounds, and then from right to left to update upper bounds, using a sweep algorithm, resulting in an $O(n^2)$ time complexity for the time-table propagator [19].

Three types of explanations were proposed in [37]: *naive*, *big-step* and *pointwise*. Here, we focus on explanations for bound updates, but explanations are similar for conflicts, i.e. when the constraint is detected as unsatisfiable. Let us consider the earliest start time updates, the latest start times are updated symmetrically. Given a task i that would overload a rectangle R_j over $[a_j, b_j]$, the naive explanation for setting $\underline{s_i}$ to b_j is made of the *current* lower bound of s_i and the *current* bounds of the tasks in $Tasks(R_j)$. Big-step explanations are more general by considering *sufficient* bounds for having this overload:

$$[\![s_i \geq a_j + 1 - d_i]\!] \wedge \bigwedge_{\ell \in Tasks(R_j)} ([\![s_\ell \leq a_j]\!] \wedge [\![s_\ell \geq b_j - d_\ell]\!]) \longrightarrow [\![s_i \geq b_j]\!]$$

Pointwise explanations refine big-step explanations by focusing on individual time points rather than whole rectangles. Each time point corresponds to a position within a rectangle that would be overloaded; and two consecutive time points can not be more than d_i apart for a task i. Let $Tasks(t_p) = \{i \in Tasks \mid \overline{s_i} \leq t_p < \underline{s_i} + d_i\}$ be the set of tasks whose mandatory part intersects with a time point t_p. Let TP be a valid set of time points for updating $\underline{s_i}$,

pointwise explanations are generated in increasing time point order:

$$[\![s_i \geq t_p + 1 - d_i]\!] \wedge \bigwedge_{\ell \in Tasks(t_p)} ([\![s_\ell \leq t_p]\!] \wedge [\![s_\ell \geq t_p + 1 - d_\ell]\!]) \longrightarrow [\![s_i \geq t_p + 1]\!] \quad \forall t_p \in TP$$

A single pointwise explanation can be seen as a big-step explanation whose rectangle that would be overloaded has width 1. There are many ways to choose a valid set of time points and the complexity results presented in this section hold for any time points choices.

We analyse worst-case time complexity as a function of the parameters that independently affect the running time. These parameters are n, c and $|\Delta|$. We assume that the integer variables are represented by their bounds, which is usually the case when considering scheduling problems, so the size of a CUMULATIVE instance is $O(n\,(log(|\Delta|) + log(c)))$ (assuming that the minimum starting time is 0). Then, c and $|\Delta|$ are numeric parameters and are included in the analysis only when they can not be avoided to express the worst-case complexity. The influence of c in the theoretical bounds lies in the maximum number of tasks that can be performed in parallel, which is also bounded by n. Because the worst cases are attained when $min(n, c) = n$, we do not use c to express the worst-case bounds. For the same reason, we do not use $|\Delta|$ in the worst-case complexity of generating (factorised) big-step explanations for a single propagator call. Indeed, $|\Delta|$ influences the maximum number of rectangles, which is also $O(n)$. However, we can not avoid $|\Delta|$ in the other complexity results, meaning that these problems are pseudo-polynomial. The upper-bound proofs are provided in this section while the lower-bound proofs are in the appendix (Section C).

▶ **Theorem 7.** *The worst-case time complexity of generating big-step explanations is* $\Theta(n^3)$ *for a single propagator call.*

Upper-bound proof. During a single propagator call, a big-step explanation is computed at most once per task-rectangle pair. The number of tasks and the number of rectangles are $O(n)$. The number of tasks per rectangle is $O(n)$, so a big-step explanation is computed in $O(n)$ time. Therefore, the worst-case time complexity of generating big-step explanations is $O(n^3)$ for a single propagator call. ◀

▶ **Theorem 8.** *The worst-case time complexity of generating big-step explanations is* $\Theta(n^2\,|\Delta|)$ *along a branch of the search tree.*

Upper-bound proof. The number of bound updates, and thus the required number of big-step explanations, can not exceed $n \times |\Delta|$ along a branch. Indeed, beyond $n \times |\Delta|$ bound updates a domain is necessarily emptied and a conflict arises. Furthermore, a big-step explanation is computed in $O(n)$ time. Therefore, the worst-case time complexity of generating big-step explanations is $O(n^2\,|\Delta|)$ along a branch. ◀

▶ **Theorem 9.** *The worst-case time complexity of generating pointwise explanations is* $\Theta(n^2\,|\Delta|)$*, both for a single propagator call and along a branch of the search tree.*

Upper-bound proof. The number of bound updates, and thus the required number of pointwise explanations, can not exceed $n \times |\Delta|$ along a branch. Furthermore, a pointwise explanation is computed in $O(n)$ time. Therefore, the worst-case time complexity of generating pointwise explanations is $O(n^2\,|\Delta|)$ along a branch, and then also $O(n^2\,|\Delta|)$ for a single propagator call. ◀

5.1 Factorised Explanations for CUMULATIVE

In this section, we show how to factorise big-step and pointwise explanations [37] for the time-table propagator of CUMULATIVE. Each explanation can be split into two parts: *(a)* a literal explaining why the updated task intersects the rectangle (resp. time point), and *(b)* literals describing the mandatory parts forming the overloaded rectangle (resp. time point). Part *(b)* is independent of the updated bound and forms a large biclique in the implication graph when many tasks run in parallel due to high capacity and multiple explanations are generated for the same rectangle (resp. time point), since mandatory parts remain unchanged during a propagator call. Exploiting this property, big-step (resp. pointwise) explanations can be efficiently factorised within a single call, as shown in Algorithm 3.

Algorithm 3 Generate a factorised explanation for the bound update $[\![s_i \geq b]\!]$ caused by β, where β is either a rectangle R_j (big-step explanation) or a time point t_p (pointwise explanation).

if f_β is not already in the implication graph **then**
  Add f_β with part *(b)* of the corresponding explanation as predecessors
Add the arcs $\ell_\alpha \to [\![s_i \geq b]\!]$ and $f_\beta \to [\![s_i \geq b]\!]$, where ℓ_α is the literal from part *(a)*

▶ **Theorem 10.** *The worst-case time complexity of generating factorised big-step explanations is* $\Theta(n^2)$ *for a single propagator call.*

Upper-bound proof. *(i)* The big-step explanation of a conflict is computed in $O(n)$ time.

(ii) At most two bound updates (lower and upper) can happen per task-rectangle pair during one propagator call. The number of tasks and the number of rectangles are $O(n)$, so the number of bound updates is $O(n^2)$. The predecessors in the factorised implication graph of a bound update for a task i and a rectangle R_j are a literal related to i and the factor f_{R_j}. Thus, computing the predecessors for all the bound updates takes $O(n^2)$ time.

(iii) At most one factor is created per rectangle, and its predecessors are computed in $O(n)$ time. Thus, computing the predecessors for all the factors takes $O(n^2)$ time.

Therefore, the worst-case time complexity of generating factorised big-step explanations is $O(n^2)$ for a single propagator call. ◀

▶ **Theorem 11.** *The worst-case time complexity of generating factorised pointwise explanations is* $\Theta(n\,|\Delta|)$ *for a single propagator call.*

Upper-bound proof. *(i)* The pointwise explanation of a conflict is computed in $O(n)$ time.

(ii) At most two bound updates (lower and upper) can happen per task-time point pair during one propagator call. The number of tasks is $O(n)$ and the number of time points is $O(|\Delta|)$, so the number of bound updates is $O(n\,|\Delta|)$. The predecessors of a bound update for a task i and a time point t_p are a literal related to i and the factor f_{t_p}. Thus, computing the predecessors for all the bound updates takes $O(n\,|\Delta|)$ time.

(iii) At most one factor is created per time point, and its predecessors are computed in $O(n)$ time. Thus, computing the predecessors for all the factors takes $O(n\,|\Delta|)$ time.

Therefore, the worst-case time complexity of generating factorised pointwise explanations is $O(n\,|\Delta|)$ for a single propagator call. ◀

By Theorem 10 and Theorem 11, biclique factorisation allows for a faster explanation generation for CUMULATIVE for a single propagator call, as well as a reduced number of generated arcs in the implication graph as explained in Section 3.2.

5.2 A Stronger Factorisation Along a Branch

Stronger factorisation can be achieved by reusing factors created earlier along the same search branch. For pointwise explanations, let A be the set of tasks intersecting a time point t_p at some point in the search, and B the set intersecting t_p later on the same branch. Since mandatory parts can only grow along a branch, we have $A \subseteq B$. If $A = B$, we reuse the previously created factor f_A; otherwise ($A \subset B$), we create a new factor f_B whose predecessors are f_A and the bound literals of tasks in $B \setminus A$. This ensures that the number of generated arcs along a branch is $O(n\,|\Delta|)$ (see Section D). For big-step explanations, part *(b)* is split into two parts: *(b-left)* containing upper-bound literals and *(b-right)* containing lower-bound literals. Each part is then associated with a single time point rather than the whole rectangle interval, allowing the same reasoning to be applied independently.

Investigating the possibility of having a $O(n\,|\Delta|)$ time complexity for generating big-step and pointwise explanations would require studying an explanation-oriented incremental algorithm for the time-table propagator, which is not the purpose of this paper. Yet, this would constitute an interesting future work.

6 Experimental Study

Biclique factorisation was implemented in `Choco-solver` v5.0.0[2,3]. We declared factors as specific type of SAT variables. Thus, each of them requires allocation of additional auxiliary objects that link CP and SAT. This cumulative allocation may mitigate the efficiency of the biclique factorisation when the factorised biclique are not large. The main objective of the experimental evaluation is to demonstrate the contribution of biclique factorisation. Specifically, we aim to quantify the benefits provided by the approach in terms of reduced memory footprint and computation time.

To do this, we selected four distinct problem categories: three involving one or more ALLDIFFERENT constraints and one involving one CUMULATIVE constraint with a large capacity. The three ALLDIFFERENT-based problems include the Golomb Ruler, Latin Square and Langford problems, all sourced from the MiniZinc Challenges [40]. The Latin Square instances converge to a solution without failures, enabling the measured memory consumption to accurately isolate the cost associated with the size of the implication graph. The fourth category consists of the Large Scale Scheduling problem, from XCSP3 library [4]. For each problem, we ran multiple instances to observe the behaviour under increasing sizes.

The experiments were carried out with OpenJDK 21.0.1 on an Intel Xeon 6230 (2.10GHz) with up to 8Go RAM per job. For each instance, a first resolution using biclique factorisation with a hard time limit of 5 minutes is launched; the number of nodes processed during this phase was recorded. Then, the same instance was solved again under a fixed limit of the recorded node count, with and without biclique factorisation. A dynamic memory management strategy was employed via the Java `-Xmx` flag: if the resolution failed due the memory exhaustion, the available heap was increased iteratively until a successful run was obtained. Once a valid resolution was achieved for the required parameters, this configuration was locked and the instance was resolved 9 more times to ensure statistical stability of computation time. Finally, the elapsed time and the cumulative size of generated clauses (i.e., the total number of arcs generated in the implication graph) were reported.

[2] `https://github.com/SulianLBC/BicliqueFactorisation-March-2026`
[3] Conflict analysis consists of the 1-UIP learning scheme without minimisation. Plus, the threshold below which variables' domain are enumerated and not bounded is set at 6500.

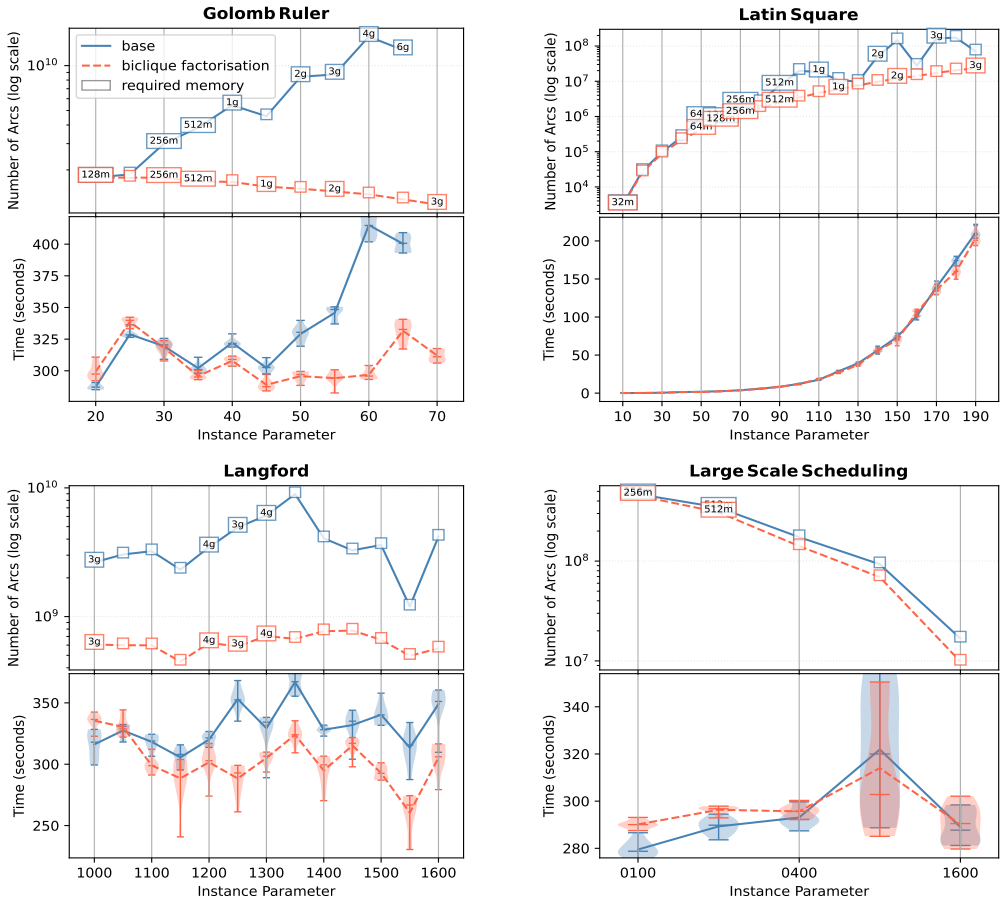

Figure 4 Total number of generated arcs and computation time without (namely, 'base') and with biclique factorisation on four problems: Golomb Ruler (upper-left), Latin Square (upper-right), Langford (lower-left) and Large Scale Scheduling (lower-right). For each problem, the top figure represents the total number of generated arcs and values within the rectangles denote the minimum memory required to solve the instance, while empty squares indicate that the memory requirement is identical to that of the previously evaluated rectangle. The bottom figure represents the computation time with the violin plot for each parameter value, the curve passes through the mean value.

The experimental results are presented in Figure 4. Each sub-figure contains two sub-plots illustrating the performance differences. The upper plot reports the resource overhead, showing the total number of generated arcs and the minimum memory required to successfully process the instance as the problem size grows. The lower plot shows the computation time over ten runs, as violin plots, to either reach the node limit or close the instance, evaluated under the strict memory constraints imposed by the protocol.

Number of Arcs The upper plots reveal a consistent reduction in the number of arcs when biclique factorisation is activated, significantly decreasing the arc count compared to the standard baseline across all problem categories, with reductions of up to 13-fold. This compactness translates to a decrease in the memory footprint, an effect that is particularly pronounced in the Golomb Ruler problem. On the largest instances of Golomb Ruler, the memory requirement is reduced by nearly a factor of three, and one instance could not be

solved under 8Go without biclique factorisation, confirming the interest of the proposed approach in managing resource consumption.

Time One must first contextualise the variance observed in certain instances, which is primarily attributable to the non-deterministic triggering of the Java Virtual Machine's garbage collector under critical memory pressure. This is particularly evident in the Langford and Large Scale Scheduling problems. Beyond this variability, a positive correlation emerges between the reduction in arc count and the improvement in solving speed; the greater the difference in arc count, the more significant the gain observed for the Golomb Ruler, Latin Square, and Langford instances (up to 28% reduction). On the smallest instances of Golomb Ruler and Large Scale Scheduling, the additional creation of factors introduces a slight overhead which is not compensated by arcs saving. In addition, the largest instances of Large Scale Scheduling do not yield noticeable improvements with biclique factorisation. This happens because the overhead incurred in explanation generation and conflict analysis is insignificant when compared to the time spent propagating the CUMULATIVE constraint.

Equivalence Test We compared the runtimes on all MiniZinc Challenge instances since 2020 that contain at least one ALLDIFFERENT or CUMULATIVE constraint, evaluating the solver without and with biclique factorisation, with a time limit of 600 seconds per instance. Out of 170 managed instances, both variants solved the same 58 instances, while most instances (112) remained unsolved within the time limit, and no instance was solved by only one variant. To assess performance in terms of runtime ratios, we applied a log transformation to the paired runtimes and conducted a log-transformed two one-sided tests (TOST) [36] equivalence test with a $\pm 5\%$ margin per instance. The test confirms that the runtimes are statistically equivalent for the instances solved by both approaches ($p \approx 0.017$ for the lower bound, $p \approx 0.012$ for the upper bound). These results indicate that biclique factorisation does not degrade performance on this benchmark, which consists of instances that are not favourable to biclique factorisation because the used global constraints are not very large.

7 Conclusion

We proposed *biclique factorisation* as a method to reduce the overhead of lazy clause generation, particularly for large global constraints where bicliques naturally arise in the implication graph. We implemented this technique for the ALLDIFFERENT and CUMULATIVE constraints, and established tight bounds on both the worst-case time complexity of explanation generation and the maximum number of generated arcs. An experimental evaluation conducted with `Choco-solver` demonstrates the practical impact of the approach.

Several directions for future work emerge from this study. First, it would be valuable to implement biclique factorisation for other global constraint propagators. In particular, alternative propagators for CUMULATIVE such as *edge-finding* and *energetic reasoning* [22] would be relevant candidates. Also, flow-based global constraints [15]—such as those generalising ALLDIFFERENT—also appear to be promising targets for further investigation. Second, combining biclique factorisation with lazy explanations is a natural direction. Moreover, since lazy explanations preserve information accumulated along the current search branch, their integration could potentially yield stronger factorisations. Investigating the extent to which this knowledge can be exploited constitutes an interesting research direction. Finally, an open question is whether factorisation can be extended beyond the scope of a single propagator, by automatically detecting bicliques induced jointly by multiple propagators.

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. URL: https://www.sciencedirect.com/science/article/pii/089571779390068A, doi:10.1016/0895-7177(93)90068-A.

2 Sahel Alouneh, Sa'ed Abed, Mohammad H. Al Shayeji, and Raed Mesleh. A comprehensive study and analysis on sat-solvers: advances, usages and achievements. *Artificial Intelligence Review*, 52(4):2575–2601, Dec 2019. doi:10.1007/s10462-018-9628-0.

3 G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, pages 21–27, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

4 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2023 xcsp3 competition. *arXiv preprint arXiv:2312.05877*, 2023.

5 Forrest Sheng Bao, Chris Gutierrez, Jeriah Jn Charles-Blount, Yaowei Yan, and Yuanlin Zhang. Accelerating boolean satisfiability (sat) solving by common subclause elimination. *Artificial Intelligence Review*, 49(3):439–453, Mar 2018. doi:10.1007/s10462-016-9530-6.

6 Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of artificial intelligence research*, 22:319–351, 2004.

7 Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 63–79, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

8 Christian Bessiere. Chapter 3 - constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29–83. Elsevier, 2006. URL: https://www.sciencedirect.com/science/article/pii/S1574652606800076, doi:10.1016/S1574-6526(06)80007-6.

9 A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009.

10 J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983. URL: https://www.sciencedirect.com/science/article/pii/0166218X83900124, doi:10.1016/0166-218X(83)90012-4.

11 Geoffrey G Chu. *Improving combinatorial optimization*. University of Melbourne, Department of Computer Science and Software Engineering, 2011.

12 Jip J. Dekker, Alexey Ignatiev, Peter J. Stuckey, and Allen Z. Zhong. Towards Modern and Modular SAT for LCG. In Maria Garcia de la Banda, editor, *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:12, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2025.42, doi:10.4230/LIPIcs.CP.2025.42.

13 Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. Towards a better understanding of the functionality of a conflict-driven sat solver. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 287–293, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

14 Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In *Proceedings of the Thirty-Fifth Australasian Computer Science Conference - Volume 122*, ACSC '12, page 115–124, AUS, 2012. Australian Computer Society, Inc.

15 Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining flow-based propagation. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, pages 146–162, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

16 T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995. URL: https://www.sciencedirect.com/science/article/pii/S0022000085710653, doi:10.1006/jcss.1995.1065.

17 Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 352–366, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

18 Mathias Fleury and Armin Biere. Efficient all-uip learned clause minimization. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 171–187, Cham, 2021. Springer International Publishing.

19 Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, pages 149–157, Cham, 2015. Springer International Publishing.

20 Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages*, pages 217–233, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

21 Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008. Special Review Issue. URL: https://www.sciencedirect.com/science/article/pii/S0004370208001410, doi:10.1016/j.artint.2008.10.006.

22 Stefan Heinz and Jens Schulz. Explanations for the cumulative constraint: An experimental study. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, pages 400–409, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

23 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. doi:10.1137/0202019.

24 Stasys Jukna and Igor Sergeev. Complexity of linear boolean operators. *Foundations and Trends® in Theoretical Computer Science*, 9(1):1–123, 2013. URL: http://dx.doi.org/10.1561/0400000063, doi:10.1561/0400000063.

25 Sulian Le Bozec-Chiffoleau, Nicolas Beldiceanu, Charles Prud'homme, Gilles Simonin, and Xavier Lorca. Bimodal depth-first search for scalable gac for alldifferent. In James Kwok, editor, *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 2610–2618. International Joint Conferences on Artificial Intelligence Organization, 8 2025. Main Track. doi:10.24963/ijcai.2025/291.

26 Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. A scalable sweep algorithm for the cumulative constraint. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 439–454, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

27 Oleg B Lupanov. On rectifier and switching-and-rectifier schemes. In *Dokl. Akad. Nauk SSSR*, volume 111, pages 1171–1174, 1956. URL: https://web.vu.lt/mif/s.jukna/boolean/lupanov56.pdf.

28 J.P. Marques Silva and K.A. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *31st Design Automation Conference*, pages 705–711, 1994. doi:10.1109/DAC.1994.204192.

29 J.P. Marques-Silva and K.A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.

30 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, Sep 2009. doi:10.1007/s10601-008-9064-x.

31 Pierre Ouellet and Claude-Guy Quimper. Time-table extended-edge-finding for the cumulative constraint. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, pages 562–577, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

32 Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

33 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022.

34 Jean-Charles RÉGIN. A filtering algorithm for constraints of difference in csps. In *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, AAAI'94, page 362–367. AAAI Press, 1994.

35 Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., USA, 2006.

36 Donald J. Schuirmann. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of Pharmacokinetics and Biopharmaceutics*, 15(6):657–680, Dec 1987. doi:10.1007/BF01068419.

37 Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, Jul 2011. doi:10.1007/s10601-010-9103-2.

38 Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 237–243, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

39 Peter J. Stuckey. Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 5–9, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

40 Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15:307–316, 2010.

41 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

42 Lintao Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, pages 279–285, 2001. doi:10.1109/ICCAD.2001.968634.

A First-UIP, Local Minimisation and Recursive Minimisation with Biclique Factorisation

The 1-UIP nogood [29] is found by exploring the reverse implication graph (arcs are in the other direction) by considering the nodes from the top of the trail (the stack of literals added to the implication graph) and replacing them by their reason in the nogood under construction. This is done until exactly one literal p from the nogood belongs to the same decision level as the conflict node (p is a *unique implication point* (UIP)). A property of the 1-UIP clause is that the most recent decision level among its literals, excluding the UIP, determines the backtrack level. After backtracking to this level, the clause becomes unit and forces the UIP to true by unit propagation. Such clauses are called *asserting* clauses and enable *non-chronological backtracking*. We present in Algorithm 4 the 1-UIP procedure adapted to biclique factorisation.

We assume that all literals of a given decision level are reachable in the implication graph from the decision node of that same level. This property is ensured, for example, when the strongest available propagators are systematically used during constraint propagation. If this assumption does not hold then the presented algorithms should be adapted accordingly.

The found nogood may not be minimal, as some of its proper subsets may also be nogoods. These subsets are of better quality, as they may be triggered earlier by unit propagation or allow larger jumps through non-chronological backtracking. Therefore, clause-learning-based

Algorithm 4 Finding the 1-UIP nogood with biclique factorisation

Require: The factorised implication graph, the `trail` and the conflict node κ
Ensure: Find the 1-UIP nogood

```
1:  NG ← ∅                                          ▷ The 1-UIP nogood under construction
2:  count ← 1
3:  Mark κ
4:  do                                              ▷ κ is initially on the top of the trail
5:      v ← trail.pop()
6:      if v is marked ∧ v is not a factor then
7:          processReason-F-1-UIP(v)
8:          count ← count − 1
9:  while count > 1
10: do
11:     p ← trail.pop()
12: while p is not marked
13: Add p to NG
14: return NG
15: procedure processReason-F-1-UIP(v)
16:     for u ∈ N⁻(v) : level(u) > 0 do             ▷ Ignore literals from initial propagation
17:         if u is not marked then
18:             if u is a factor then
19:                 processReason-F-1-UIP(u)
20:             else if level(u) = level(κ) then    ▷ Literal from conflict level
21:                 count ← count + 1
22:             else                                ▷ Literal from anterior level
23:                 Add u to NG
24:             Mark u
```

Algorithm 5 Local minimisation of the 1-UIP nogood with biclique factorisation

Require: The factorised implication graph, the 1-UIP nogood NG and its UIP p
Ensure: Apply local minimisation to NG

$NG_{init} \leftarrow NG$ ▷ *Store the information of the initial nogood*
for $n \in NG_{init} \setminus \{p\}$ **do**
  if n is not a decision $\wedge$ processReason-F-Local(v) **then**
    Remove n from NG
procedure processReason-F-Local(v)
  for $u \in N^-(v)$ **do**
    if u is a factor **then**
      if u is not marked **then**
        $res \leftarrow$ processReason-F-Local(u)
      if u is marked as *included* $\vee$ $res =$ `TRUE` **then**
        Mark u as *included*
      else if u is marked as *non-included* $\vee$ $res =$ `FALSE` **then**
        Mark u as *non-included*
        return `FALSE` ▷ *A literal from the reason does not belong to the nogood*
    else if $u \notin NG_{init}$ **then**
      return `FALSE` ▷ *A literal from the reason does not belong to the nogood*
  return `TRUE` ▷ *All literals from the reason belong to the nogood*

Algorithm 6 Recursive minimisation of the 1-UIP nogood with biclique factorisation

Require: The factorised implication graph, the `trail`, the 1-UIP nogood NG and its UIP p
Ensure: Apply recursive minimisation to NG

$levels \leftarrow \{\texttt{level(}n\texttt{)} \mid n \in NG\}$
$minRank(\ell) \leftarrow min_{n \in NG \mid \texttt{level(}n\texttt{)}=\ell}(\texttt{rank(}n\texttt{)})$ $\quad \forall \ell \in levels$
$NG_{init} \leftarrow NG$ ▷ *Store the information of the initial nogood*
for $n \in NG_{init} \setminus \{p\}$ **do**
  if IsDominated(n, n) **then**
    Remove n from NG
function IsDominated(v, n)
  if $v \in NG_{init} \setminus \{n\} \vee v$ is marked as *dominated* **then return** `TRUE`
  else if v is marked as *undominated* **then return** `FALSE`
  else if v is a decision $\vee$ $\texttt{level(}v\texttt{)} \notin levels \vee \texttt{rank(}v\texttt{)} < minRank(\texttt{level(}v\texttt{)})$ **then**
    Mark v as *undominated*
    return `FALSE` ▷ *The literal v is not dominated by NG_{init}*
  else
    for $u \in N^-(v) : \texttt{level(}u\texttt{)} > 0$ **do** ▷ *Ignore literals from initial propagation*
      if $\neg$IsDominated(u, n) **then**
        Mark v as *undominated*
        return `FALSE` ▷ *The literal v is not dominated by NG_{init}*
    Mark v as *dominated*
    return `TRUE` ▷ *The literal v is dominated by NG_{init}*

solvers often apply internal procedures to minimise the learned nogood. Two minimisation methods that yield good practical performance are *local* [6] and *recursive* [38] minimisation. The former removes from the nogood all literals whose reason is included in the nogood. The

latter removes every literal n that is dominated by the nogood NG, i.e. every path from a decision to n passes through at least one literal of $NG \setminus \{n\}$.

We present in Algorithm 5 and Algorithm 6 the local and recursive minimisations adapted to biclique factorisation. Actually, no particular changes are required for recursive minimisation in the presence of factors. Let `rank(`ℓ`)` denote the index of a literal ℓ in the trail. The smaller the rank, the longer ago the literal was added to the implication graph. For recursive minimisation, the conditions in line 10 related to *levels* and *minRank* are optimisations that reduce the size of the explored subgraph [38, 18].

B Lower-Bound Proofs for ALLDIFFERENT

To establish lower bounds on the worst-case time complexity of generating graph explanations for ALLDIFFERENT, we construct infinite families of *constraint satisfaction problems* (CSPs) that each contain an ALLDIFFERENT constraint. In these families, the number of variables r and the number of values d for the ALLDIFFERENT constraint can grow independently subject only to the necessary condition $r \leq d$, ensuring that all considered instances are non-trivial.

Proof of the lower bound in Theorem 2. Let $(G^C_{k,\ell})_{k,\ell \in \mathbb{N}_{>0}}$ be a family of CSPs defined as follows (see Figure 5-left). Let $k, \ell \in \mathbb{N}_{>0}$, the CSP $G^C_{k,\ell}$ is made of:

- A set V of k values whose minimum is v_{min} and whose maximum is v_{max}, and a set A of k integer variables whose domain is V;
- A set W, disjoint from V, of $k+\ell$ values greater than v_{min} and smaller than v_{max}, and a set B of k integer variables whose domain is $V \cup W$;
- A constraint ALLDIFFERENT$(A \cup B)$.

When the GAC propagator for ALLDIFFERENT is called for the first time, the residual graph has two SCCs: one made up of $A \cup V$ and another of $B \cup W \cup \{t\}$. All variable-value pairs in $B \times V$ are pruned, and the explanations are

$$\bigwedge_{a \in A} (\llbracket a \geq v_{min} \rrbracket \wedge \llbracket a \leq v_{max} \rrbracket \wedge \bigwedge_{w \in W} \llbracket a \neq w \rrbracket) \longrightarrow \llbracket b \neq v \rrbracket \quad \forall b, v \in B, V$$

So, a number of $k^2 \times k \times (k+\ell+2)$ arcs are generated in $\Omega(k^3\,(k+\ell))$ time. Also, the number of variables is $r = 2k$, so $k = \Omega(r)$. And the number of distinct values is $d = 2k+\ell$, so $k+\ell = \Omega(d)$. Therefore, the worst-case time complexity of generating graph explanations is $\Omega(r^3\,d)$ for a single propagator call, and then also $\Omega(r^3\,d)$ along a branch. ◀

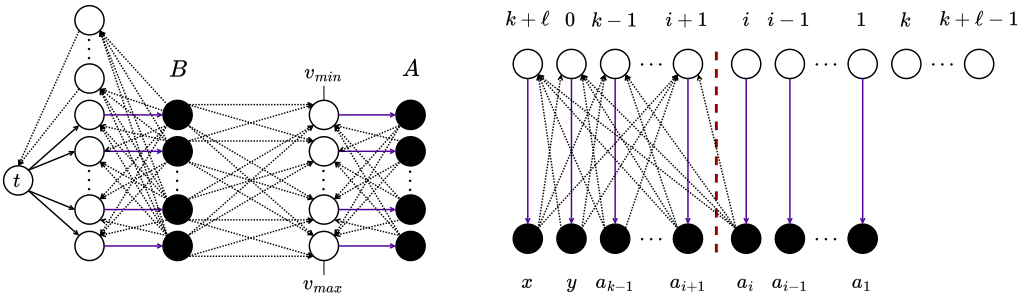

Figure 5 LEFT: an illustration of the residual graph during initial propagation for the CSP $G^C_{k,\ell}$. RIGHT: an illustration of the residual graph along a branch for the CSP $G^B_{k,\ell}$. The artificial vertex t is not represented because its presence would not change the SCCs.

Proof of the lower bound in Theorem 4. Let $k, \ell \in \mathbb{N}_{>0}$, and let $G^C_{k,\ell}$ be the CSP defined previously. A factor is created for the SCC $A \cup V$, and computing its $\Omega(k(k+\ell))$ predecessors requires $\Omega(k(k+\ell))$ time. The number of variables is $r = 2k$, so $k = \Omega(r)$, and the number of values is $d = 2k + \ell$, so $k + \ell = \Omega(d)$. Therefore, the worst-case time complexity of generating factorised graph explanations is $\Omega(r\,d)$ for a single propagator call. ◀

Proof of the lower bound in Theorem 6. Let $(G^B_{k,\ell})_{k,\ell \in \mathbb{N}_{>0}}$ be a family of CSPs defined as follows (see Figure 5-right). Let $k, \ell \in \mathbb{N}_{>0}$, the CSP $G^B_{k,\ell}$ is made of:

- A set $A = (a_i)_{1 \leq i \leq k-1}$ of $k-1$ integer variables whose initial domain is $\{0, \ldots, k+\ell\}$;
- Two integer variables x and y whose initial domain is $\{0, \ldots, k+\ell\}$;
- A constraint ALLDIFFERENT$(A \cup \{x, y\})$;
- A boolean variable b and a constraint $b \implies z \neq i$ for $k \leq i \leq k+\ell-1$ and $z \in A \cup \{x, y\}$;
- A set $B = (b_i)_{1 \leq i \leq k-1}$ of $k-1$ boolean variables and a constraint $b_i \implies z \neq i$ for $1 \leq i \leq k-1$ and $z \in (A \setminus \{a_i\}) \cup \{x, y\}$.

The search consists in branching first on b and then on the boolean variables in B by increasing order of index, and setting them first to true. No pruning occurs during initial propagation nor after setting b to true. After b_i is set to true, with $1 \leq i \leq k-1$, the variables a_j for $1 \leq j \leq i-1$ are already instantiated to j, the values $\{k, \ldots, k+\ell-1\}$ are present in the domain of no variable, and the current domain of a_i is $\{i, \ldots, k-1\} \cup \{0, k+\ell\}$. Also, the variables $\{a_j | i+1 \leq j \leq k-1\} \cup \{x, y\}$ and the values $\{i+1, \ldots, k-1\} \cup \{0, k+\ell\}$ form an SCC. See Figure 5-right.

Thus, a_i is forcibly instantiated to i and the explanation is

$$\bigwedge_{z \in \{a_j | i+1 \leq j \leq k-1\} \cup \{x,y\}} ([\![z \geq 0]\!] \wedge [\![z \leq k+\ell]\!] \wedge \bigwedge_{v \in \{1,\ldots,i\} \cup \{k,\ldots,k+\ell-1\}} [\![z \neq v]\!]) \longrightarrow [\![a_i = i]\!]$$

So, a number of $(k-i+1) \times (2+i+\ell)$ arcs are generated in $\Omega((k-i)\,(i+\ell))$ time. Because $\sum_{i \in \{1,\ldots,k-1\}} (k-i) \times (i+\ell) = \Omega(k^2\,(k+\ell))$, $\Omega(k^2\,(k+\ell))$ time is spent generating factorised graph explanations along the branch. Finally, the number of variables for the ALLDIFFERENT constraint is $r = k+1$, so $k = \Omega(r)$. And the initial number of distinct values is $d = k+\ell+1$, so $k + \ell = \Omega(d)$. Therefore, the worst-case time complexity of generating factorised graph explanations is $\Omega(r^2\,d)$ along a branch. ◀

C Lower-Bound Proofs for CUMULATIVE

To establish lower bounds on the worst-case time complexity of generating big-step and point-wise explanations for CUMULATIVE, we construct infinite families of CSPs, each containing a CUMULATIVE constraint. In these families, either the number of tasks n, or both n and the size of the time interval $|\Delta|$, can grow independently depending on the parametrisation of the bound.

Proof of the lower bound in Theorem 7. Let $(BS^C_k)_{k \in \mathbb{N}_{>0}}$ be a family of CSPs defined as follows (see Figure 6-left). Let $k \in \mathbb{N}_{>0}$, the CSP BS^C_k is made of:

- A set F^L of k fixed long tasks starting at time 0, of duration k and height 1;
- A set $F^S = (f^S_i)_{0 \leq i < \frac{k}{2}}$ of $\lceil \frac{k}{2} \rceil$ fixed short tasks such that f^S_i starts at time $2i$, has duration 1 and height 1;
- A set U of k unfixed tasks starting at time 0 or later, of duration 1 and height 2;
- A constraint CUMULATIVE$(F^L \cup F^S \cup U, k+1)$.

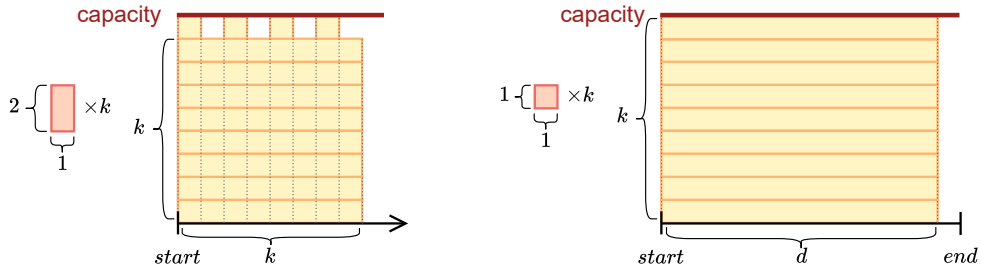

Figure 6 Illustration of the initial profile of BS_k^C (left) and $PW_{k,d}$ (right) for arbitrary $k, d \in \mathbb{N}_{>0}$

For each task u in U, one call to the time-table propagator iteratively sets the lower bound of the starting time of u to 1, then to 2, and so on until k because the profile is made of k steps. A big-step explanation of size $\Omega(k)$ is computed for each one of these bound updates. So, the time spent generating big-step explanations is $\Omega(k^3)$. The number of tasks is $n = 2k + \lceil \frac{k}{2} \rceil$, so $k = \Omega(n)$. Therefore, the worst-case time complexity of generating big-step explanations is $\Omega(n^3)$ for a single propagator call. ◀

Proof of the lower bound in Theorem 8. Let $(BS_{k,d}^B)_{k,d \in \mathbb{N}_{>0}}$ be a family of CSPs defined as follows (see Figure 7). Let $k, d \in \mathbb{N}_{>0}$, the CSP $BS_{k,d}^B$ is made of:

- A set F of k fixed tasks starting at time $d-1$, of duration d and height 1;
- A set U of k unfixed tasks starting between 0 and $2d-1$, of duration d and height 1;
- A task t starting between 0 and $d-1$, of duration d and height 1;
- A constraint $\text{CUMULATIVE}(F \cup U \cup \{t\}, k+1)$;
- A set $B = (b_i)_{i \in \{1,\dots,d-1\}}$ of $d-1$ boolean variables;
- A constraint $b_i \implies s_t \geq i$ for all $i \in \{1, \dots, d-1\}$.

Let us say the search consists in branching first on the boolean variables in increasing index order by setting them first to true. During initial propagation, all the tasks in F have a mandatary part of height 1 between $d-1$ and $2d-1$ while t has a mandatory part of height 1 between $d-1$ and d. Therefore, the lower bound of the starting time of each task in U is updated and set to $d-1$. A big-step explanation of size $\Omega(k)$ is generated for each bound update. The fix point for the Time-Table propagator is reached by a single call. Then, the search sets b_1 to true and the mandatory part of t is now between $d-1$ and $d+1$. The lower bounds are then updated again and big-step explanations are generated. This process is repeated over the whole branch. A number of k big-step explanations of size $\Omega(k)$ are generated at each node along a branch made of $d-1$ nodes. Thus, the time spent generating big-step explanations is $\Omega(k^2\, d)$.

The number of tasks is $n = 2k+1$, so $k = \Omega(n)$. The size of the time interval is $|\Delta| = 3d-1$, so $d = \Omega(|\Delta|)$. Therefore, the worst-case time complexity of generating big-step explanations is $\Omega(n^2\, |\Delta|)$ along a branch. ◀

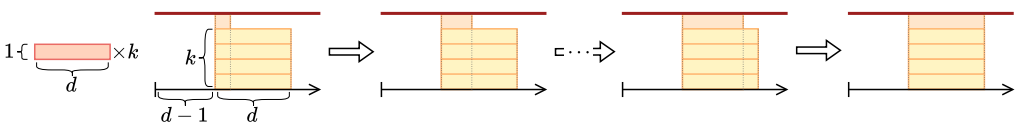

Figure 7 Illustration of the evolution along a branch of the profile of $BS_{k,d}^B$ for arbitrary $k, d \in \mathbb{N}_{>0}$

Proof of the lower bound in Theorem 9. Let $(PW_{k,d})_{k,d\in\mathbb{N}_{>0}}$ be a family of CSPs defined as follows (see Figure 6-right). Let $k, d \in \mathbb{N}_{>0}$, the CSP $PW_{k,d}$ is made of:

- A set F of k fixed tasks starting at time 0, of duration d and height 1;
- A set U of k unfixed tasks starting between 0 and d, of duration 1 and height 1;
- A single constraint $\text{CUMULATIVE}(F \cup U, k)$.

One call of the time-table propagator updates all the lower bounds of the starting times of the tasks in U, setting them to d. For each task u in U, pointwise explanations are generated for all of the $d-1$ time points between 0 and d because u has duration 1. Each one of these explanations is of size $\Omega(k)$. So, the time spent generating pointwise explanations is $\Omega(k^2\, d)$. The number of tasks is $n = 2k$, so $k = \Omega(n)$. The size of the time interval is $|\Delta| = d + 1$, so $d = \Omega(|\Delta|)$. Therefore, the worst-case time complexity of generating pointwise explanations is $\Omega(n^2\,|\Delta|)$ for a single propagator call, and then also $\Omega(n^2\,|\Delta|)$ along a branch. ◀

Proof of the lower bound in Theorem 10. Let $k \in \mathbb{N}_{>0}$, and let BS_k^C be the CSP defined previously. For each rectangle, a factor is created and its $\Omega(k)$ predecessors are computed. Since the number of rectangles is $\Omega(k)$, this requires $\Omega(k^2)$ time. Moreover, we have $k = \Omega(n)$. Therefore, the worst-case time complexity of generating factorised big-step explanations is $\Omega(n^2)$ for a single propagator call. ◀

Proof of the lower bound in Theorem 11. Let $k, d \in \mathbb{N}_{>0}$, and let $PW_{k,d}$ be the CSP defined previously. For each time point between 0 and d, a factor is created and its $\Omega(k)$ predecessors are computed. Since the number of time points is $\Omega(d)$, this requires $\Omega(k\,d)$ time. Moreover, we have $k = \Omega(n)$ and $d = \Omega(|\Delta|)$. Therefore, the worst-case time complexity of generating factorised pointwise explanations is $\Omega(n\,|\Delta|)$ for a single propagator call. ◀

D Stronger Factorisation of Pointwise Explanations

▶ **Theorem 12.** *The maximum number of arcs generated by strongly factorised pointwise explanations is $O(n\,|\Delta|)$ along a branch of the search tree.*

Proof. Let us consider the subgraph of the implication graph induced by the strongly factorised pointwise explanations. To bound the number of arcs, we consider a transformed graph in which each literal is split into two sub-literals depending on whether it appears in part *(a)* or part *(b)* of an explanation. Each original literal has exactly either zero or two predecessors, and so do the sub-literals. This split ensures that each sub-literal appears in the predecessor set of at most one node, by definition of the strong factorisation. Since splitting literals can only increase the number of arcs, bounding the number of arcs in this transformed graph also bounds the number of arcs in the original one.

We distinguish four types of arcs: (A1) sub-literal → sub-literal, (A2) sub-literal → factor, (A3) factor → sub-literal, and (A4) factor → factor. Because the number of factors created is $O(n\,|\Delta|)$, the number of nodes in the transformed graph is $O(n\,|\Delta|)$. From the previous comments about sub-literals, the total numbers of predecessors and successors of sub-literals are $O(n\,|\Delta|)$. Thus, the number of arcs of types (A1)–(A3) is $O(n\,|\Delta|)$.

Moreover, each factor has at most one predecessor factor and at most one successor factor, which are the ones representing the same time point in a previous and next propagator call. Since the number of factors is $O(n\,|\Delta|)$, the number of arcs of type (A4) is also $O(n\,|\Delta|)$.

Combining these bounds shows that the total number of arcs in the transformed graph is $O(n\,|\Delta|)$. Since the transformed graph contains at least as many arcs as the original one, the same bound holds for the original graph. ◀