Constraint-directed search for all-interval series

Constraints manuscript No.
(will be inserted by the editor)

Constraint-Directed Search for All-Interval Series

Md Masbaul Alam Polash · M A Hakim Newton · Abdul Sattar

Received: date / Accepted: date

Abstract All-interval series is a standard benchmark problem for constraint satisfaction search. An all-interval series of size n is a permutation of integers $[0, n)$ such that the differences between adjacent integers are a permutation of $[1, n)$. Generating each such all-interval series of size n is an interesting challenge for constraint community. The problem is very difficult in terms of the size of the search space. Different approaches have been used to date to generate all the solutions of AIS but the search space that must be explored still remains huge. In this paper, we present a constraint-directed backtracking-based tree search algorithm that performs efficient *lazy checking* rather than *immediate constraint propagation*. Moreover, we prove several key properties of all-interval series that help prune the search space significantly. The reduced search space essentially results into fewer backtracking. We also present scalable parallel versions of our algorithm that can exploit the advantage of having multi-core processors and even multiple computer systems. Our new algorithm generates all the solutions of size up to 27 while a satisfiability-based state-of-the-art approach generates all solutions up to size 24.

Keywords All-Interval Series · Backtracking Search · Constraint Programming · Lazy Constraint Propagation

1 Introduction

The all-interval series (AIS) problem was first presented in [13] and was later added to the CSPLIB (www.csplib.org) constraint satisfaction problem library as prob007. Since then it is often used as a standard benchmark problem

Md. Masbaul Alam Polash · M A Hakim Newton · Abdul Sattar
Institute for Integrated and Intelligent Systems
Griffith University, Australia
E-mail: mdmasbaulalam.polash@griffithuni.edu.au, mahakim.newton@griffith.edu.au, a.sattar@griffith.edu.au

for constraint satisfaction search (e.g. [25,27,6]). This problem was originally inspired by a well-known problem occurring in serial musical composition [9], a problem that has been extensively studied and used by Ernest Krenek, a famous modern music composer [15]. Later, the problem of finding such series in the context of constraint solving was formulated in [13].

An *all-interval series* of *size* n, denoted by AIS(n), is a permutation of integers $[0, n)$ such that the differences between adjacent integers are a permutation of $[1, n)$. The AIS problem is a straightforward example of the graceful graphs problem [19] in which the graph is a path. The AIS problem poses a significant challenge to combinatorial search. However, finding a single solution is no longer a challenge because it has been shown in [23] that one particular solution can be found without any search. Now the challenge remains in finding all the solutions of size n. In fact, there are two distinct challenges: One is to count the number of solutions of size n and the other is to generate all the solutions of size n. The counting version does not necessarily require all the solutions to be generated explicitly and hence is not as difficult as the generation version. Using a problem specific idea of equivalence classes of partial solutions without explicitly assigning specific values to specific variables, a non-constraint-based breadth-first search approach in [3] addressed the counting version and reported the numbers of solutions of sizes up to 40.

To deal with the challenge of the generation version, different approaches have been used to date. These include local search (LS) for satisfiability (SAT) [13,22,4] and constraint programming (CP) [23,20,12,7]. Using a *reformulated definition* [12], a very recent SAT-based approach in [18] can generate all solutions of size up to 24 given one week time. Since the search space to be explored for the AIS problem is of factorial size and the total number of solutions increases about 2 to 3 times with 1 increase in n [3], constraint-directed search algorithms developed on generic platforms do not scale well. As a result, generating all the unique solutions beyond size 24 practically appears to be very hard. The challenges are to find useful AIS properties that might be used as auxiliary constraints and to find efficient ways to propagate the constraints to effectively prune the search space. Also, designing parallel algorithms that run on multi-core processors and multiple computers are essential in solving such problems that require enormous processing time. For example, a parallel discrepancy-based search (PDS) in [16] preserves the node visit ordering of limited discrepancy-based search algorithm and scales to thousands of workers. It shows the great potential of constraint programming in a massively parallel environment for which good search strategies are known.

In this paper, our focus is to generate all the unique solutions for AIS of size n and hence we are dealing with the generation version of the AIS problem. We prove several new properties of the reformulated AIS problem to help prune the search space significantly resulting into fewer backtracking. We show the effectiveness of the new properties by using them on top of the CP model used in [12]. We develop a constraint-directed backtracking-based tree search algorithm incorporating these new properties. We also develop a very efficient *lazy checking* technique that performs significantly better in

finding reformulated AIS than the typical *immediate propagation* technique used in constraint solvers. Moreover, we present scalable parallel versions of our tree-search algorithm that solve large problems using multiple processes and multiple computers. Our algorithms now generate all AIS solutions of size up to 27 along with significant speed up in problems of size up to 24. We also use our lazy constraint checking in the non-constraint-based breadth-first search of [3] to prune useless search nodes and to count the numbers of solutions up to size 48. Moreover, we save about 10–17% time to generate all solutions of sizes up to 29 by using a non-constraint-based depth-first search that uses the AIS representation used in the counting version in [3].

In the rest of the paper, we cover all-interval series preliminaries, and then present our algorithms, results and conclusions.

2 All-Interval Series

Given the all-interval series problem of size n, the solution that we define below can be constructed without search. Actually, it was shown in [23] by using a symmetric version of this solution.

Definition 1 (Default Solution) Given a size n, the sequence $(0, n-1, 1, n-2, 2, n-3, \ldots, \lfloor n/2 \rfloor)$ producing the differences $(n-1, n-2, \ldots, 1)$ is an AIS.

Example 1 For size 6, the following sequence gives the differences consecutively from 5 to 1. Thus it is known as the default solution of size 6.

$$0 \;\; 5 \;\; 1 \;\; 4 \;\; 2 \;\; 3$$

□

Definition 2 (Regular Symmetry) Assume two AIS of size n be $s = (s_0, s_1, \ldots, s_{n-1})$ and $s' = (s'_0, s'_1, \ldots, s'_{n-1})$. AIS s and s' are (regular) symmetric to each other if s'_k is s_{n-1-k} or $n-1-s_k$ or $n-1-s_{n-1-k}$. Notice that the third one actually combines the first two.

Example 2 Consider the default solution of size 6.

$$0 \;\; 5 \;\; 1 \;\; 4 \;\; 2 \;\; 3$$

The following three sequences are regular symmetric to the above.

$$3 \;\; 2 \;\; 4 \;\; 1 \;\; 5 \;\; 0$$
$$5 \;\; 0 \;\; 4 \;\; 1 \;\; 3 \;\; 2$$
$$2 \;\; 3 \;\; 1 \;\; 4 \;\; 0 \;\; 5$$

□

These three regular symmetries can be eliminated by using two constraints $d_1 < d_{n-1}$ and $s_0 < s_1$ where $d_i = |s_i - s_{i-1}|$ separately or jointly. Along with these *regular symmetries*, another kind of symmetry named *conditional symmetry* was identified in [12].

Definition 3 (Conditional Symmetry) Let $s = (s_0, \ldots, s_{k-1}, s_k, \ldots, s_{n-1})$ be an AIS such that $|s_k - s_{k-1}| = |s_{n-1} - s_0|$, then $(s_k, \ldots, s_{n-1}, s_0, \ldots, s_{k-1})$ is also an AIS of given size n.

Example 3 Consider the default solution of size 6:

$$0 \quad 5 \quad 1 \quad 4 \quad 2 \quad 3$$

The difference between the first and last number is 3. Also the difference between 1 and 4 is 3. Thus by splitting the sequence between 1 and 4 and joining the rest of the sequence on to the start, one can get another solution:

$$4 \quad 2 \quad 3 \quad 0 \quad 5 \quad 1$$

This sequence is known to be conditionally symmetric to the original one. □

Combining regular and conditional symmetries together, there are 8-way symmetries in AIS. To handle all these symmetries simultaneously, a *reformulated definition* of AIS was also presented in [12]. In the rest of this paper, we mainly focus on the reformulated AIS rather than the original AIS.

Definition 4 (Reformulated Definition) Given integers $[0, n)$ with $n > 3$, assume $s = (s_0, \ldots, s_{n-1})$ be a permutation with $s_0 = 0, s_1 = n-1, s_2 = 1$ and $d = (d_1, \ldots, d_n)$ be the differences with $d_n = |s_0 - s_{n-1}|$ and $d_k = |s_k - s_{k-1}|$ for $(1 \leq k < n)$. If d is a permutation of $[1, n)$ with exactly one number appearing twice then s is called a *Reformulated AIS* (RAIS).

Example 4 Consider the following sequence:

$$0 \quad 5 \quad 1 \quad 2 \quad 4 \quad 3$$

It has the differences $(5, 4, 1, 2, 1, 3)$ with 1 appearing twice. Thus this sequence is known as the RAIS of size 6. □

Notice that the RAIS takes the difference between first and last number into consideration. So, the RAIS deals with n differences whereas the AIS deals with $n - 1$ differences. The important property of RAIS is that it produces two conditionally symmetric AIS [12] as shown in the following lemma.

Lemma 1 (RAIS to AIS) *Assume $d_i = d_j$ is the* repeated difference *in the RAIS $(s_0, \ldots, s_{n-1})$ where $i < j$. Clearly, sequences $(s_i, \ldots, s_{n-1}, s_0, s_{i-1})$ and $(s_j, \ldots, s_{n-1}, s_0, s_{j-1})$ are the two conditionally symmetric AIS.*

Example 5 Consider the following RAIS:

$$0 \;\; 5 \;\; 1 \;\; 2 \;\; 4 \;\; 3$$

Here the differences are $(5, 4, 1, 2, 1, 3)$ with 1 appearing twice. By splitting the sequence between 1 and 2 or between 4 and 3, one can get the following two AIS which are conditionally symmetric to one another.

$$\begin{array}{cccccc} 2 & 4 & 3 & 0 & 5 & 1 \\ 3 & 0 & 5 & 1 & 2 & 4 \end{array}$$

□

Note that the fixed values of s_0, s_1, s_2 handle the regular symmetries. In RAIS there is no requirement that the repeated difference must be between the first and the last number. The following lemma in [12] shows not all integer differences could be repeated in a RAIS.

Lemma 2 (Repeated Parity) *The repeated difference in a RAIS is even iff n is $4k$ or $4k + 1$, and odd otherwise.*

Since for a given size, finding one solution to the RAIS (and hence AIS) problem, namely the default solution, requires no search, several approaches have been proposed to find a second solution. These include ant colony optimization meta-heuristics based Ant-P solver [24] and local search based adaptive search techniques [8,26]. However, the main challenge, as mentioned before, is to generate all the solutions for a given size.

Early LS for SAT faced great difficulties even with size 12 of the generation version of the problem [13,22]. Using *AllDiff* and *cycle* [5] constraints, a CP method [23] later solved up to size 12. Another CP method [20] solved up to size 14 using *AllDiff* constraint with bound consistency and/or arc consistency. Yet another CP method in [7] solved up to size 14 using *AllDiff* and *interval* constraint with arc consistency. Another LS for SAT using an efficient SAT encoding for AIS along with few redundant constraints solved problems up to size 18 [4]. The recent state-of-the-art results for generating all solutions are however from another CP and a SAT based approaches. Below we briefly describe the CP- and SAT-based approaches.

CP-Based Approach. To encode RAIS, the CP approach in [12], uses an *AllDiff* and *AtLeastOne* constraint on the integers and differences respectively. It did not however use any constraint on the parity of repeated difference. In this approach, the 'first-fail' and 'indomain-min' strategies are used respectively as variable and value selection heuristics. As in [18], for experiments, we use the CP solver opturion CPX [2] although the ILog solver [1] was used in [12]. The CP approach could generate all RAIS of size up to 20.

SAT-Based Approach. Using *AtMostOne* and *AtLeastOne* constraints, the SAT-based approach in [18], *denoted by SAT here*, provides an efficient SAT encoding of the RAIS problem. It then uses the SAT solver named Clasp [11]. The SAT based approach generates all RAIS of size up to 24.

3 Our RAIS Properties

Characteristics of the default solution, which is at the same time both an AIS and a RAIS of size n, play a key role in *RAIS construction*. The default solution starts at 0 and places the next integer in the series so that the next largest difference required is obtained. In this process, the last integer $\lfloor n/2 \rfloor$ becomes the repeated difference, which is once created inside the series between two integers and then again from the difference of the first and the last integers.

It is worth mentioning here that the next difference obtained in the RAIS construction process is basically such that it cannot be obtained only from the integers that have not been used yet. We call such differences the *mandatory differences*, since we have to consider obtaining them immediately.

Example 6 Consider the following RAIS:

0 5 1 4 2 3

Here integer 4 is placed after 1 to obtain the mandatory difference 3, because difference 3 cannot be obtained later from any of the pairs of the so-far unused integers 2, 3, and 4. □

Definition 5 (Mandatory Difference) Given a size $n > 3$ and a RAIS $s = (s_0, \ldots, s_{k-1}, s_k, \ldots, s_{n-1})$, difference $d_k = |s_k - s_{k-1}|$ is a *mandatory difference* at position $0 < k < n$ if $|s_i - s_j| \neq d_k$ for any $k \leq i < j < n$.

If a mandatory difference is not obtained at its position, of course there is another way to obtain it; place the integer equal to the difference at the end so that with the starting integer 0, it can still produce that difference.

Example 7 Consider the following RAIS:

0 5 1 2 4 3

Here difference 3 is mandatory after 1, but it has been obtained by placing the integer 3 at the end. Since the mandatory difference is missing at its position, we call difference 3 the *missing difference*. In this way, obtaining difference 3 is actually deferred until the end of the series. Note that 3 is not a missing difference in the default solution $(0, 5, 1, 4, 2, 3)$; it is obtained at its position, even if it appears at the end, too.

Definition 6 (Missing Difference) Given a size $n > 3$ and a RAIS $s = (s_0, \ldots, s_{k-1}, s_k, \ldots, s_{n-1})$, difference $d_n = |s_0 - s_{n-1}|$ is a *missing difference* at position k if $d_n \neq |s_k - s_{k-1}|$ and $|s_i - s_j| \neq d_n$ for any $k \leq i < j < n$.

Note that only one missing difference is possible in the entire series because only one integer can be placed at the end. The mandatory difference may, however, be possible at each position of the series. The missing difference paves the way to defer a mandatory difference, but only for once. Once a difference is marked missing, any mandatory difference arisen later must be obtained immediately. If no difference is mandatory at one position, one can choose to obtain any unobtained difference or even a previously obtained difference, if no difference is yet repeated. RAIS, by definition, allows only one repetition.

Lemma 3 (Last Difference) *Difference $d_n = |s_0 - s_{n-1}|$ is not repeated in any RAIS of size n except the default one.*

Proof In the default solution, differences are obtained consecutively from the largest to the smallest. Moreover, each difference is obtained only once, missing no difference on the way. Therefore, difference d_n which equals the last integer $\lfloor \frac{n}{2} \rfloor$ is repeated. In all other RAIS, there will be a missing difference which is later obtained only from the difference between the first and the last integers. □

Lemma 4 (Third Difference) *If $(0, n-1, 1)$ is not immediately followed by integer $n-2$ then the integers 2 and $n-2$ in either order will appear in successive positions in the RAIS.*

Proof After $(0, n-1, 1)$ in the RAIS, difference $n-3$ becomes mandatory. If it is not obtained immediately by placing integer $n-2$ next to 1, then integer $n-3$ must be placed at the end of the RAIS. Since 1 cannot be next to $n-3$ any more, difference $n-4$ now can be obtained in only one way; by placing integers 2 and $n-2$ next to each other.

Before proving further properties of RAIS, we prove two lemmas on the maximum sum of differences in a regular (circular) permutation.

Lemma 5 (Maximum DiffSum) *Let $s = (s_0, \ldots, s_{n-1})$ be an arbitrary permutation of n integers $[0, n)$ and $d = (d_1, \ldots, d_n)$ be the differences between successive numbers meaning $d_n = |s_0 - s_{n-1}|$ and $d_k = |s_k - s_{k-1}|$ for $(1 \leq k < n)$. The sum of such differences is at most $n(n-1)/2 + \lfloor n/2 \rfloor$.*

Proof We need to prove that the sum of the differences is $n^2/2$ for even n and to $(n^2-1)/2$ for odd n. Each difference d_k is either $s_k - s_{k-1}$ or $s_{k-1} - s_k$. Therefore, among the $2n$ numbers in the sum of the differences $\sum_{1 \leq k \leq n} d_k$, we have exactly n positive and n negative numbers. Now, the sum will be maximised if the larger s_ks give positive and smaller s_ks give negative numbers. Assuming n to be even or odd, we get the following two sums:

$$S_{even} = 2((n-1) + \ldots + \frac{n}{2}) - 2((\frac{n}{2} - 1) + \ldots + 0) = \frac{n^2}{2}$$

$$S_{odd} = 2((n-1) + \ldots + (\frac{n-1}{2} + 1)) - 2((\frac{n-1}{2} - 1) + \ldots + 0) = \frac{n^2-1}{2}$$

Notice that in the expression of S_{odd}, the middle number $(n-1)/2$, being neither strictly small nor strictly large, gets omitted as its net contribution $+(n-1)/2 - (n-1)/2$ in the maximisation process will be 0.

□

Lemma 6 (Conditional DiffSum) *Assume $s = (s_0, \ldots, s_{n-1})$ be a permutation of integers $[0, n)$ and the differences be $d = (d_1, \ldots, d_n)$ with $d_n = |s_0 - s_{n-1}|$ and $d_k = |s_k - s_{k-1}|$ for $(1 \leq k < n)$. Also, assume (j, j') or (j', j) are respectively two integers $(s_i, s_{i'})$ such that $0 \leq i < n$ and $i' = (i+1) \mod n$. If $0 \leq j' < j < \lceil (n-1)/2 \rceil$ then the sum of the differences will be at most $n(n-1)/2 + \lfloor n/2 \rfloor + 2j - 2\lceil (n-1)/2 \rceil$. Similarly, if $\lfloor (n-1)/2 \rfloor < j < j' \leq n-1$ then the sum of the differences will be at most $n(n-1)/2 + \lfloor n/2 \rfloor + 2\lfloor (n-1)/2 \rfloor - 2j$.*

Proof For convenience, let us consider s_0 and s_{n-1} to be next to each other. Given the proof of Lemma 5, we know the larger s_ks give positive and the smaller s_ks give negative numbers in the maximum sum of differences $S = n(n-1)/2 + \lfloor n/2 \rfloor$. This means the large numbers have small numbers at both of their left and right, and vice versa; moreover, when n is odd, the middle number has a large number at one side and a small number at the other side. Below we show the small and large numbers for even and odd sizes.

$$0 \ldots (\lceil (n-1)/2 \rceil - 1) \qquad\qquad (\lceil (n-1)/2 \rceil) \quad \ldots (n-1)$$

$$0 \ldots (\lceil (n-1)/2 \rceil - 1) \qquad \lceil (n-1)/2 \rceil \qquad (\lceil (n-1)/2 \rceil + 1) \ldots (n-1)$$

For small integers $(0 \leq j' < j < \lceil (n-1)/2 \rceil)$, as per the other conditions, when j and j' are next to each other, j can no longer contribute negatively to S because of j' being smaller than j. However, a positive contribution from j means a change of $+2j$ in S. Since j's one side is no longer available, there will be two non-small numbers $\lceil (n-1)/2 \rceil \leq j'' < j''' \leq n-1$, which will be bound to be next to each other. However, because of this j'' will no longer be able to contribute positively to S resulting into a change of $-2j''$ in S. Since we want to maximise S, we need the least value of j'', which is $\lceil (n-1)/2 \rceil$. So the maximum possible sum will be $n(n-1)/2 + \lfloor n/2 \rfloor + 2j - 2\lceil (n-1)/2 \rceil$.

$$0 \ldots \quad (\lfloor (n-1)/2 \rfloor) \qquad\qquad (\lfloor (n-1)/2 \rfloor + 1) \ldots (n-1)$$

$$0 \ldots (\lfloor (n-1)/2 \rfloor - 1) \qquad \lfloor (n-1)/2 \rfloor \qquad (\lfloor (n-1)/2 \rfloor + 1) \ldots (n-1)$$

To keep the expressions symmetric, we can also write the small and large numbers for even and odd sizes in the above way. Now, for the large integers $\lfloor (n-1)/2 \rfloor < j < j' \leq n-1$, the maximum possible sum of differences is $n(n-1)/2 + \lfloor n/2 \rfloor + 2\lfloor (n-1)/2 \rfloor - 2j$ and the proof, from the opposite perspective, is very similar to that for the small integers.

□

Lemma 7 below restricts the choices for a repeated difference. It is a key to significantly prune the search space.

Lemma 7 (Repeated Range) *The repeated difference in a RAIS of size n cannot be greater than $\lfloor n/2 \rfloor$.*

Proof Given a size n, the default solution $(0, n-1, 1, n-2, 2, n-3, \ldots, \lfloor n/2 \rfloor)$ is both an AIS and RAIS. The default solution produces the differences $(n-1, n-2, \ldots, 1)$, whose sum is $n(n-1)/2 + \lfloor n/2 \rfloor$. Moreover, each RAIS is also a regular permutation of $[0, n)$. Therefore, using Lemma 5, the maximum possible sum of differences of any RAIS is also $n(n-1)/2+\lfloor n/2 \rfloor$. By definition, each RAIS has all the difference $[1, n)$ at least once with exactly one of them repeated again. Therefore, the repeated difference in a RAIS is at most $\lfloor n/2 \rfloor$.
□

Corollary 1 (Repeated Difference) *Combining Lemma 2 and 7, the repeated difference in a RAIS of size n is* at most *$\lfloor n/2 \rfloor$ and is even iff n is $4k$ or $4k+1$, and odd otherwise. This also means the repeated difference is* at least *2 iff n is $4k$ or $4k+1$, and 1 otherwise.*

Corollary 2 (RAIS DiffSum) *As an extension of Corollary 1, the sum of the differences of a RAIS is at least $n(n-1)/2 + P$ and at most $n(n-1)/2 + \lfloor n/2 \rfloor$ where P is 2 for $(n \mod 4) < 2$ and 1 for $(n \mod 4) \geq 2$.*

Combining Lemma 6 with Corollary 2, the following lemma allows us to cut down many search branches, whose upper bound of the sum of the differences is less than the least possible sum of the differences for an RAIS.

Lemma 8 (DiffSum Pruning) *Assume s_0 and s_{n-1} of a RAIS are next to each other. There exist no two integers next to each other in a RAIS such that both are either in $[0, H)$ or in $(L, n-1]$, where $H = (\lfloor n/2 \rfloor + P)/2$, $L = n - 1 - H$, and P is 2 for $(n \mod 4) < 2$ and 1 for $(n \mod 4) \geq 2$.*

Proof We know $\lceil (n-1)/2 \rceil = \lfloor n/2 \rfloor$. From Lemma 6, we know for two small integers $0 \leq j' < j < \lfloor n/2 \rfloor$ next to each other, the maximum possible sum of differences in a permutation is $n(n-1)/2+\lfloor n/2 \rfloor - 2\lfloor n/2 \rfloor + 2j = n(n-1)/2 - \lfloor n/2 \rfloor + 2j$. However, as per the Corollary 2, the least possible sum of differences for an RAIS is $n(n-1)/2 + P$ where P is constant for a given n. Note that each RAIS is also a regular permutation. Now from $n(n-1)/2 - \lfloor n/2 \rfloor + 2j = n(n-1)/2 + P$ and then simplifying $j = (\lfloor n/2 \rfloor + P)/2$, we can find H, the least value of j for which the least possible sum of differences of RAIS is met. For any $j \in [0, H)$, the least possible sum cannot be met and so no such RAIS exist. In a similar fashion, from the opposite perspective, we can show no RAIS exist for two large integers in $(L, n-1]$ next to each other.

4 Our Auxiliary Constraints

As is normally the case, the constraints capturing the RAIS definition are sufficient to separate a valid RAIS sequence from an invalid RAIS sequence. However, auxiliary constraints that capture various provable properties could help prune the search space. In order to test this hypothesis, we implemented Corollary 1 repeated difference (RD), Lemma 3 last difference (LD), Lemma 4 third difference (TD), and Lemma 8 DiffSum (SD) pruning as auxiliary constraints within the CP approach of [12].

Given series $s = (s_0, \ldots, s_{n-1})$ and differences $d = (d_1, \ldots, d_n)$ where $d_n = |s_0 - s_{n-1}|$ and $d_k = |s_k - s_{k-1}|$ for $(1 \leq k < n)$, we model the new properties of RAIS in the following ways. Notice that here we present the contrapositive of the actual constraint for Lemma 3 and 4. That's because we tried different ways to model the new properties and these contrapositive versions perform better than others.

Lemma 3 constraint:

$$\exists_{j \in 3..n}(n \neq j \wedge d_i = d_j) \Rightarrow \forall_{i \in 1..n-1}(d_i = n - i)$$

Lemma 4 constraint:

$$(s_3 \neq n - 2) \Rightarrow (s_{n-1} = n - 3)$$

$$\neg\exists_{i \in 3..n-2}(d_i = n - 4) \Rightarrow (s_3 = n - 2)$$

Lemma 8 constraint:

$$l_1 = (\lfloor n/2 \rfloor + P)/2, \quad l_2 = n - 1 - l_1$$

$$P = 2 \text{ if } (n \mod 4) < 2 \text{ else } 1$$

$$l(i, j) = (i < l_1 \wedge j < l_1) \vee (i > l_2 \wedge j > l_2)$$

$$\forall_{i \in 2..n-2}(l(s_i, s_{i+1}) = 0)$$

Corollary 1 constraint:

$$\exists_{i \in 1..n}((d_i \leq \frac{n}{2}) \wedge \exists_{j \in 3..n}(n \neq j \wedge d_i = d_j) \wedge (n \bmod 4 < 2) = (d_i \bmod 2 = 0))$$

In Figure 1, we report performance of the CP approach when various combinations of the above four properties (RD, LD, TD and SD) are included. The combinations include only adding one property (e.g. SD, LD, TD, RD) and two properties (e.g. LDSD, TDSD, LDRD, TDLD, TDRD, RDSD), three properties (e.g. TDLDSD, LDRDSD, TDRDSD, TDLDRD), all properties and no properties (no lemmas NL). The NL version is actually the one used in [18]. We use the CP solver opturion CPX [2] to run these versions of the CP approach. Overall the version that include LD made significant improvements in search time and versions that include SD degrade.

In Table 1, we further compare the best results obtained by the versions listed above with that of the SAT approach [18]. Notice that the best results

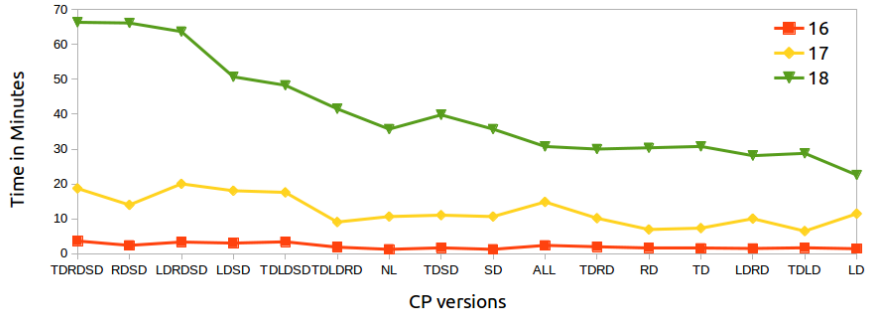


Fig. 1 Performance of different CP versions with auxiliary constraints

Table 1 Execution time comparison (time in minutes)

Size	17	18	19	20	21
CP	10.6	37.4	198.6	980.4	
aCP	6.5	28.8	110.2	571.8	3729.3
SAT	0.3	1.7	4.7	18.8	57.7

obtained from the CP versions with auxiliary constraints (denoted by aCP) overall are significantly worse than the SAT approach. However, the aCP version is significantly faster than the original CP version (i.e. NL). Moreover, the aCP can solve size 21 while the CP cannot. None of the new versions can solve size 22 in one week time.

5 Our Constraint-Driven Backtracking Search

Overall what we found from [18] is that a CP approach using a generic solver does not perform well compared to the SAT approach. Moreover, in our experiments described above with our new RAIS properties as auxiliary constraints within the CP approach, we observed some improvement but still cannot outperform the SAT approach. Given the observations, we decided to investigate whether it is the reason that the new properties are not really very useful in terms of search pruning or the generic CP solver could not exploit the auxiliary constraints efficiently. For this we have implemented a constraint-driven backtracking-based tree search in C++ language. In our implementation, the constraint propagation part is conceptually kept separate and the backtracking search framework is such that addition of auxiliary constraints becomes very easy. In the following two sections, we first describe the search framework using immediate propagation and then we improve it using lazy checking.

6 Our Immediate Propagation

Since a typical backtracking approach does not perform well, we then make an attempt to restrict the branches at each choice points. So at each position of the RAIS, we determine the mandatory difference (if any) and explore the branch with that difference. By obtaining all the mandatory differences one after another, one can easily get the default solution. For example, in Figure 2, after placing $0, 6, 1$ as the root, difference 4 becomes mandatory. So, 5 is placed after 1 in the leftmost branch of the tree. Then difference 3 becomes mandatory. In this way, we go deep into the branch recursively and reach the default solution (0 6 1 5 2 4 3). Now we backtrack to find other solutions. During backtracking, we mark the mandatory difference as missing and explore branches with all the non-mandatory differences. In Figure 2, when we backtrack from (0 6 1 5 2) to (0 6 1 5), 3 is marked as missing and differences $1, 2$ becomes non-mandatory. Since according to definition 6, the integer equal to the missing difference has to be placed at the end of the series, we only explore the branch with difference 1. In all the solutions other than the default one, we also consider, for the entire series, obtaining one repeated difference i.e. again using a difference that has already been used in the path from the root to the current node of the search tree.

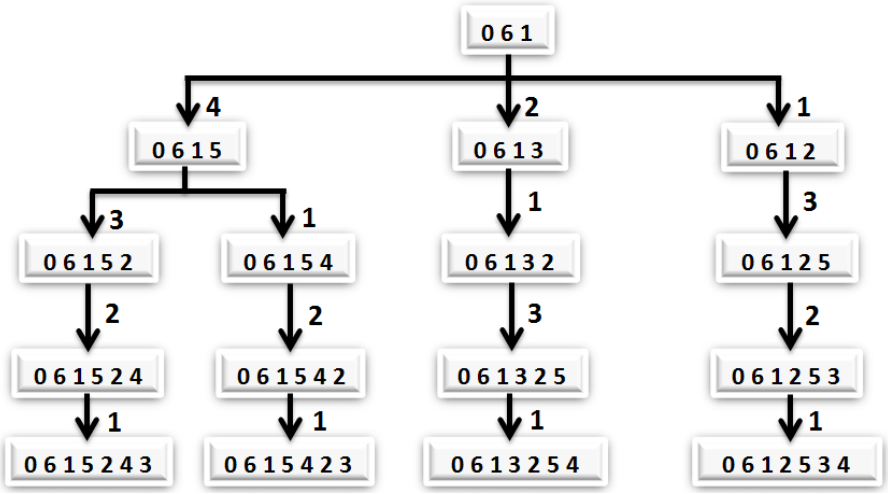


Fig. 2 Backtracking-based tree search for RAIS of size 7

6.1 Data Structures

We now describe the data structures required to implement the immediate propagation based backtracking approach described above. Notice that at each

level of the search tree, when we append the next integer from left to right to the series, we need to know the mandatory difference (if any) at that position of the RAIS. If a mandatory difference exists but is not used, then it becomes a missing difference and we have to keep track of the missing difference. We also need to know which difference is already used and which one is repeated. Moreover, we need to know which integers are available for future use.

bool ai$[0 : n-1]$ = true	//	available integers
ai$[0]$ = ai$[1]$ = ai$[n-1]$ = false	//	first three are fixed
int aiL $= 1$, aiR $= n-2$	//	exclusive left, inclusive right
int ud$[1 : n-1] = 0$	//	#times a diff is used
int ad$[1 : n-1] = 0$	//	#ways a diff can be obtained
for $(1 \leq k \leq n-3)$ ad$[k] = n-3-k$	//	initialisation of avail diffs
int adL $= 0$, adR $= n-3$	//	exclusive left, inclusive right
int missd $= 0$	//	the missing diff if $\neq 0$

In the above list of variables, Boolean array ai keeps track of integers which are still available for future use. Notice that integers 0, 1, and $n-1$ are not available from the beginning. During search index pair (aiL, aiR] encloses the available integers, although few used integers may be included in the range. An integer array named *used differences* ud keeps track of the number of times a difference is already used in the part of the series that is already constructed. Note that a difference can appear only once in a RAIS except one difference that could appear twice. A difference that is not used is available for future use. For a given position of the series, the mandatory difference can be identified by maintaining a typical *immediate propagation* data structure.

This immediate propagation data structure keeps track of the differences that can only be obtained from the unused integers. Note that an unused difference that cannot be obtained only from the unused integers is a mandatory difference. For this, we keep an array, namely *available differences* ad, which holds the number of ways difference d can be obtained from the unused integers. Initially, unused integers are [2, n-2] and just by using them each difference $1 \leq k \leq n-3$ can be obtained in $n-3-k$ ways. Using the example below, we explain the way the arrays ad and ud are maintained during search. However, for convenience, index pair (adL, adR] encloses the available differences although few used differences could be included in the range. Note that arrays ud and ad can be merged into one array and in our experiment we did that. We use another variable missd that holds the missing difference when it is non-zero. Note that for convenience of implementation, missd when chosen as the missing difference is treated as used, but in the computation of array ad, it is to be counted as unused.

Example 8 To find RAIS of size 7, according to definition 4, the first three numbers are fixed to 0, 6, 1. Then using the remaining unused integers $(2, 3, 4, 5)$, differences $(1, 2, 3, 4)$ can be obtained in $(3, 2, 1, 0)$ ways respectively (shown in Figure 3(a)). Here 0 indicates that difference 4 cannot be obtained only from the unused integers. So, difference 4 is mandatory at this position. Thus by generating difference 4, differences $(1, 2, 3)$ can now be obtained from unused integers $(2, 3, 4)$ in $(2, 1, 0)$ ways (shown in Figure 3(b)). If the mandatory difference 4 is marked as missing at this position, then another difference (say 3)

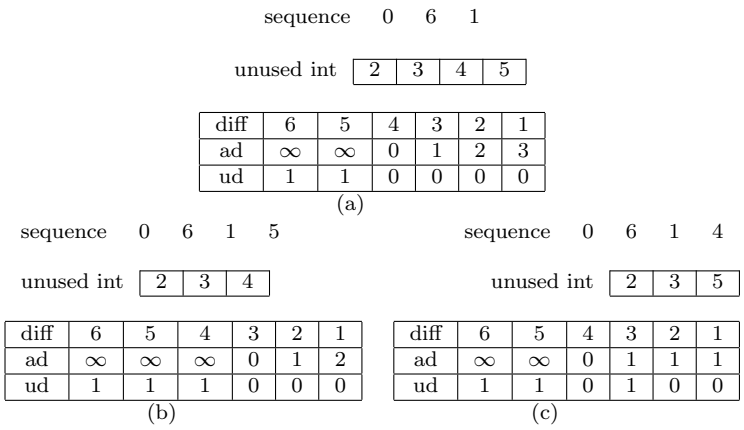


Fig. 3 Maintaining ud and ad arrays in immediate propagation technique

needs to be obtained. By obtaining difference 3, differences $(1, 2, 3)$ can now be obtained from integers $(2, 3, 5)$ in $(1, 1, 1)$ ways (shown in Figure 3(c)). □

6.2 Required Procedures

We now describe the procedures that are required to modify the data structures described above both when an integer (and so a difference) is used and as part of the backtracking that use is later revoked. Procedures rewind and unwind below update the arrays ai, ad and ud whenever an integer k and a difference d are respectively made used or unused in the series during recursion rewinding or unwinding. Moreover, function mandatory returns whether a difference d is mandatory at that point of RAIS construction. Notice that in Procedure rewind, for the newly used integer k, the changes are made to ad assuming that k can no longer produce any differences with the still unused integers and in Procedure unwind, this modification is just reversed.

```
proc rewind(integer k, difference d)   // before recursion
    ai[k] = false, ++ud[d]             // k, d not available
    if(missd != d)       // missing diff is not being decided
        foreach(aiR ≥ k' > aiL)      // update array ad
            if (ai[k'] and ¬ud[|k − k'|])
                −−ad[|k − k'|]    // k can't produce any diffs
        if (missd and ¬ud[|k−missd|])   // integer missd
            −−ad[|k−missd|]   // treated used, actually not

proc unwind(integer k, difference d)   // after recursion
    if(missd != d)       // missing diff is not being decided
        if (missd and¬ud[|k−missd|])   // integer missd
            ++ad[|k−missd|]   // treated used, actually not
        foreach(aiR ≥ k' > aiL)      // restore array ad
            if (ai[k'] and ¬ud[|k − k'|])
                ++ad[|k − k'|]   // k can again produce diffs
```

```
    −−ud[d], ai[k] = true      // k, d available again

func bool mandatory(difference d)
    return ad[d] ≤ 0    // not possible from unused integers
```

Below we present our algorithm called backtracking. In the main procedure, the first three integers in the series are $(0, n-1, 1)$. Also an array $l[i][j]$ is initialised to perform DiffSum pruning (Lemma 8). We then call the recursive Procedure widen to explore the search branches at the current node of the search tree. Procedure widen calls Procedure deepen to go one level down in the search tree; Lemma 8 is considered at this stage to prune search branches. Backtracking occurs when the search returns back from Procedure deepen and at that point Procedure widen considers the next branch to explore. The algorithm has four global variables: one array s is to hold the series, an integer cs to keep track of the size so far constructed, another integer repeatd to denote whether a difference is already repeated and another array l for Lemma 8. Note that in this algorithm, we embedded all our new RAIS properties. However, we run experiments with those new RAIS properties on and off both.

```
        algo backtracking

int s[1 : n]              // series containing n integers
int cs = 0                // current size of the series
int repeatd = 0           // repeated diff if non-zero
bool l[1 : n]             // static for Lemma 8

proc main(size n)         // execution starts here
    s[1] = 0, s[2] = n − 1, s[3] = 1, cs = 3
    l1 = (n/2 + (n  mod 4 < 2 ? 2 : 1))/2
    l2 = n − 1 − l1
    for (i = 1 . . . n, j = 1 . . . n)
        l[i][j] = (i < l1 ∧ j < l1) ∨ (i > l2 ∧ j > l2)
    if (n > 3) widen()    // explore branches

proc widen()      // creates branches
01  if (cs == n − 1)
02      if (!missd) s[++cs] = ⌊n/2⌋ // default solution
03      else if (¬repeatd xor ¬ud[|s[cs] - missd|])
04          s[++cs] = missd
05      print s; return
06  if (missd == n − 3) // Lemma 4
07      if (s[cs] = n − 2 and ai[2])
08          deepen(2, n − 4), return
09      if (s[cs] = 2 and ai[n − 2])
10          deepen(n − 2, n − 4), return
11  update adL, adR, aiL, aiR as needed
12      they are restored before each return
13  foreach (adR ≥ d > adL)   // mandatory diffs
14      if (ud[d] or ¬mandatory(d)) continue
15      foreach (k ∈ {s[cs] + d, s[cs] - d})
16          if (ai[k]) deepen(k, d)
17      if (¬missd and ai[d])
18          missd = d, rewind(d, d)
19          widen()
20          unwind(d, d), missd = 0
21      return       // only one mandatory diff handled
```

```
22  foreach (adR ≥ d > adL)    // non-mandatory diffs
23      if (ud[d]) continue
24      foreach (k ∈ {s[cs] + d, s[cs] − d})
25          if (ai[k]) deepen(k, d)
26  if (repeatd) continue    // repeat only once below
27  for (d = n/2; d ≥ 1; d = d − 2)    // Corollary 1
28      if (d == missdiff) continue    // Lemma 3
29      if (¬ud[d]) continue    // must be used before
30      foreach (k ∈ {s[cs] + d, s[cs] − d} and ai[k])
31          repeatd = d, deepen(k, d), repeatd = 0

proc deepen(integer k, difference d)    // goes down
    if (l[k][s[cs]]) return // Lemma 8
    s[++cs] = k, rewind(k, d)
    widen()
    unwind(k, d), −−cs
```

In Procedure widen, if the current size cs is $n-1$ (Lines 01-05), then for the default solution that has no missing difference, we append $\lfloor n/2 \rfloor$. For other RAIS, we append the missing difference if either no repeated difference exists or the new difference to be obtained is not used before. For other values of cs, we apply Lemma 4 (Lines 06-10) which implies that if $n-3$ is the missing difference, then integers 2 and $n-2$ will be next to each other. Procedure deepen(k, d) appends integer k to the series and rewinds the recursion by calling Procedure rewind first and Procedure widen again; the recursion is later unwinded by calling Procedure unwind.

In Procedure widen, we then narrow the (aiL, aiR] and (adL, adR] ranges down (Line 11). Lines 13-21 deal with the mandatory differences. For a mandatory difference, we determine the candidate integers to be appended to the series (Lines 15-16). Alternatively, we can consider the mandatory difference as a missing difference (which can be done only once on each path) and go into further recursion (Lines 17-20). Lines 22-25 deal with the non-mandatory differences. We then deal with the repeated difference (Lines 26-31). If a difference is not yet repeated, we then try to repeat each difference suggested by Corollary 1. However, according to Lemma 3, a missing difference cannot be repeated. Also, a difference not yet used is not considered here (Line 29). Lastly, for each available to-be-repeated difference, we go into further recursion.

Theorem 1 (Algorithm Properties) *Our algorithm* backtracking *correctly finds all the RAIS of a given size n.*

Proof Our algorithm is based on standard backtracking-style tree search where the differences are used to create search branches. The purpose of the immediate propagation procedure is to use fail-first approach in value selection while the variable ordering is fixed before search. The other pruning decisions are based on properties that are proven in Lemma 3, 4, 8 and Corollary 1.

□

6.3 Experimental Results

We implemented our backtracking algorithm with immediate propagation in C++. In the analysis of the results, we denote our algorithm by IP. We ran all experiments reported in this paper on the same high performance computing cluster *Gowonda* at *Griffith University*. Each node of the cluster is equipped with Intel Xeon CPU E5-2650 processors @2.60 GHz, FDR 4x InfiniBand Interconnect, having system peak performance 18949.2 Gflops.

Table 2 Effect of new properties in immediate propagation

Size	#Nodes in Billions					Time in Minutes				
	NL	**RD**	**SD**	**RDSD**	**%Imp**	**NL**	**RD**	**SD**	**RDSD**	**%Imp**
20	0.5	0.5	0.3	0.3	**40**	1.8	1.7	1.3	1.2	**33.3**
21	2.9	2.9	2.1	2.1	**27.6**	15.1	14.5	11.3	10.7	**29.1**
22	20.2	19.8	15.0	14.7	**27.2**	81.2	79.3	62.1	59.9	**26.2**
23	128.7	124.7	99.2	95.9	**25.5**	509.3	491.8	401.5	384.2	**24.6**

(%) Imp = 100% × (NL - RDSD) / NL

We analyse the performance improvements due to our new RAIS properties within our IP algorithm. Table 2 shows the effect of using the new properties, reported in Corollary 1 and Lemma 8, in our IP algorithm. Here, Column NL denotes the baseline version that does not use these two properties. Column RD denotes a version using only Corollary 1 that restricts the range of the repeated difference. Column SD denotes a version using only Lemma 8 that applies DiffSum pruning technique. Lastly, Column RDSD denotes a version using both these properties. The new properties exhibit significant improvements in the number of nodes visited and the execution time. The combined improvement is overall about 24–33%. We do not show the effect of Lemma 3 and Lemma 4 in this paper since the improvements achieved from them are comparatively much less than Corollary 1 and Lemma 8.

Table 3 Execution time comparison (Time in minutes). The total number of AIS solutions can be obtained by multiplying #Sols by 8

Size	CP	SAT	IP	(%)Imp	#Sols
18	37.40	1.76	**0.07**	96.02	63,837
19	198.60	4.73	**0.30**	93.65	181,412
20	980.40	18.81	**1.26**	93.30	437,168
21		57.73	**10.75**	81.38	1,306,478
22		246.60	**59.98**	75.68	4,821,338
23		946.20	**384.16**	59.40	14,864,374
24		2575.80	**1868.87**	27.45	39,404,484

(%) Imp = 100% × (SAT - IP) / SAT

We compare the performance of our IP algorithm that uses all of our new RAIS properties (i.e. Lemma 3, 4, 8 and Corollary 1) with that of two most recent state-of-the-art algorithms: the CP-based algorithm in [12] and the SAT-based algorithm in [18]. We obtained their implementation from the authors of [18]. Note that encoding the RAIS using *AllDiff* and *AtLeastOne* constraints, the CP approach in [12] could find all RAIS of size up to 20. The SAT approach in [18] using the definition finds all RAIS of size up to 24. Along with our IP algorithm, we ran these solvers on our computers and gave one week time cut-off to all algorithms for all runs. From our experimental results, Table 3 shows that IP performs much better than CP and SAT for up to size 24. Also, notice that the ratio of the time taken by IP and SAT are increasing steadily meaning the advantage of using IP algorithm reduces with greater sizes. We therefore need more efficient techniques even within the dedicated algorithm. Nevertheless, none of the algorithms could solve larger instances that have sizes 25 or above. However, our immediate propagation method significantly reduces failures during search. Table 4 shows that over the CP approach, our IP approach achieves 39-74% failure reductions. The number of failures for the CP approach shown are collected from [12].

Table 4 Search failure comparison

Size	#Fails		#Fails/Sol		(%) Failure
	CP	IP	CP	IP	Reduction
15	91,695	55,202	28.66	17.26	**39.80**
16	389,142	182,548	55.67	26.11	**53.09**
17	2,093,203	908,251	116.95	50.74	**56.61**
18	13,447,654	5,198,103	210.65	81.43	**61.34**
19	79,270,906	28,127,135	436.94	155.04	**64.52**
20	435,374,856	110,130,322	995.90	251.92	**74.70**

(%) Reduction = 100% × (CP#Fails - IP#Fails) / CP#Fails

7 Lazy Checking

Profiling our C++ program implementing the IP algorithm, we observed that the most time consuming parts of the algorithm are the rewind and unwind procedures. When an integer is used, Procedure rewind immediately updates the available differences array ad and upon backtracking, Procedure unwind immediately restores the previous ad values. These immediate update and restore operations performed as part of the typical backtracking and standard immediate propagation mechanisms become very time consuming. Note that the available differences are used in determining the mandatory difference at a given position. As an alternative to all these, in this paper, we present a very efficient *lazy checking* approach that performs on demand checking at the

point where the information is used. Instead of maintaining the number of ways a difference can be obtained, a lazy checking is rather made on demand to determine whether a difference can be obtained from the unused integers.

Lazy computation has been using in local search and constraint-based local search for quite long time [14,17,28,10]. Lazy approach compute values only when the values are actually needed. The values that are not needed right now need not be recomputed. Constraint solvers generally spend enormous time in constraint propagation that is done immediately when a choice is made during search and is later revoked on backtracking. This propagation overhead perhaps could be reduced to a large extent if performed lazily. Lazy propagation is however not studied well in the constraint programming paradigm. In this paper, we show the difference a lazy approach could bring when it replaces a typically used immediate propagation approach.

7.1 Data Structures

For the lazy checking approach, we just keep an array ud where ud[d] holds the number of times difference d is *used so far* in the series. Thus, we omit the ad array (and so the index pair adL and adR) of Figure 3 to implement the idea of lazy checking. All other data structures required in immediate propagation approach are also required here.

7.2 Required Procedures

For the lazy checking approach to work, we need to replace the Procedures rewind, unwind, and mandatory described for the immediate propagation approach with their new versions adapted to the lazy checking approach. The main constraint-directed backtracking algorithm described for the immediate propagation approach will however be exactly used in the lazy checking approach except that it will only call the new versions of the three procedures. Below we present the lazy checking versions of the three procedures.

```
proc rewind(integer k, difference d)
    ai[k] = false, ++ud[d]

proc unwind(integer k, difference d)
    --ud[d], ai[k] = true

func bool mandatory(difference d)
01  int maxd = aiR - aiL - 1 // max diff possible
//  Adjust maxd based on missing differences
02  if (missd and missd > aiR)
03      maxd = missd - aiL - 1
04  else if (missd and missd < aiL + 1)
05      maxd = aiR - missd
//  Check if d is possible from available integers
06  if (d > maxd) return true
07  if (ai[missd + d] or (missd and ai[missd - d]))
```

```
08      return false
09  foreach (aiR ≥ k > aiL + d)
10      if (ai[k] and ai[k − d]) break
11  return (k ≤ aiL + d)
```

Procedures rewind and unwind update ud[d] whenever a difference d is respectively made used and unused during recursion rewinding and unwinding. Lastly, function mandatory returns whether a difference d is mandatory at that point of RAIS construction. This is done first by computing the maximum difference maxd from available integers (Line 01). However, maxd is adjusted based on missd (Lines 02-05). This is because for convenience of implementation, missd when chosen as the missing difference is treated as used, but in computing the differences could-be obtained, it is to be counted as unused. A difference d is mandatory if it is larger than maxd (Line 06) or neither can be obtained (Lines 07-08) by placing an integer next to the to-be last integer (which is missd) nor from the available integers (Lines 09-11).

7.3 Experimental Results

For convenience, we denote our lazy checking approach by LC. The implementation of LC is also done in C++ and is run on the same computers as described in the experiments with IP.

Table 5 Effect of new properties in Lazy checking

Size	#Nodes in Billions					Time in Minutes				
	NL	**RD**	**SD**	**RDSD**	**%Imp**	**NL**	**RD**	**SD**	**RDSD**	**%Imp**
20	0.5	0.5	0.3	0.3	**40**	0.7	0.6	0.5	0.5	**28.6**
21	2.9	2.9	2.1	2.1	**27.6**	3.9	3.7	3.2	3.0	**23.1**
22	20.2	19.8	15.0	14.7	**27.2**	26.7	25.5	22.4	21.6	**19.1**
23	128.7	124.7	99.2	95.9	**25.5**	173	164	149.6	142.4	**17.7**

(%) Imp = 100% × (NL - RDSD) / NL

Like with IP, we observe the effect of our new RAIS properties particularly Corollary 1 and Lemma 8 within the LC approach. In Table 5, Column NL denotes the baseline version that does not use these two properties. Column RD denotes a version using only Corollary 1 that restricts the range of the repeated difference. Column SD denotes a version using only Lemma 8 that applies DiffSum Pruning technique. Lastly, Column RDSD denotes a version using both these properties. The new properties exhibit significant improvements in the number of nodes visited and the execution time. The combined improvement is about 17–28%.

We compare the performance of our LC algorithm that uses all of our new RAIS properties with that of the similar version of IP and also with that of the CP-based algorithm in [12] and the SAT-based algorithm in [18]. All algorithms are given one week time cut-off in all runs. Table 6 presents the

Table 6 Execution time comparison (Time in minutes). The total number of AIS solutions can be obtained by multiplying #Sols by 8

Size	CP	SAT	IP	LC	(%)Imp	#Sols
18	37.40	1.76	0.07	**0.03**	57.1	63,837
19	198.60	4.73	0.30	**0.12**	60.0	181,412
20	980.40	18.81	1.26	**0.51**	59.5	437,168
21		57.73	10.75	**3.01**	72.0	1,306,478
22		246.60	59.98	**21.58**	64.1	4,821,338
23		946.20	384.16	**142.41**	62.9	14,864,374
24		2575.80	1868.87	**695.81**	62.8	39,404,484
25				**5297.77**		**126,853,094**

(%) Imp = 100% × (IP- LC) / IP

experimental results. Clearly, LC significantly outperforms all CP, SAT and IP algorithms up to size 24 and finds all solutions of size 25, which none of other three algorithms could find. Overall LC runs almost 3 times faster than IP. We do not show the search failures for LC separately. This is because IP and LC essentially explore the same search nodes, the only distinction is in the constraint propagation i.e. the way the mandatory difference is computed. While IP immediately maintains `ad` the array of available differences, LC performs on demand lazy checking to find the mandatory difference. This clearly makes a significant improvement in the run time performance. Nevertheless, none of the algorithms could generate all solutions of size 26.

Given the performance of our LC algorithm, it is a challenge for generic CP solvers to support lazy constraint propagation and to find ways to effectively utilise auxiliary constraints within the search procedure.

8 Parallel Algorithms

Given that a very efficient dedicated algorithm such as our lazy checking approach could not generate all solutions of size 26, we implement parallel versions of our LC algorithm. On one hand, multi-core processors and computer clusters are easily available now-a-days. On the other hand, large-sized problems naturally demand enormous computing time. Algorithms to solve these large problems should therefore take the advantage of advanced hardware availability. We have implemented two parallel versions of our algorithm: multi-process and client-server versions. Our parallel versions use the *producer-consumer* architecture presented in [21] and *client-server model* presented in [29].

8.1 Multi-Process Version

Initially, our algorithm is run up to a given recursion depth and all the partial RAIS (henceforth referred to by *task*) are stored in a file called *input file*.

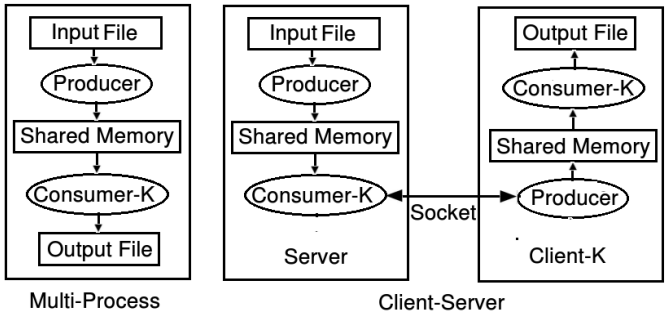


Fig. 4 Parallel architectures: multi-process and client-server

We then create a producer and a given number of consumer processes as shown in Figure 4. The producer uses the input file and gradually supplies the tasks from the input file to the consumers through a shared memory. Consumers using each partial RAIS as a root perform the backtracking search to find all complete RAIS at the search subtree. Consumers take one task at a time from the shared memory while the producer puts a given number of them in the shared memory at once.

```
semaphore mutex = 1
semaphore fillCount = 0
semaphore emptyCount = SharedMemSize

proc producer()
    while true
        item = readFromFile()
        lock(emptyCount)
            lock(mutex)
                writeItemIntoSharedmemory()
            unlock(mutex)
        unlock(fillCount)


proc consumer()
    while true
        lock(fillCount)
            lock(mutex)
                readItemFromSharedmemory()
            unlock(mutex)
        unlock(emptyCount)
```

We use semaphore to synchronise the processes and ensure mutual exclusion. For this, we use three semaphores, fillCount, emptyCount and mutex.

Semaphore fillCount denotes the number of items already in the shared memory and available to be read, while emptyCount denotes the number of available spaces in the shared memory where items could be written. fillCount is incremented and emptyCount is decremented when a new item is written into the shared memory. If the producer tries to decrement emptyCount when its value is zero, the producer is put to sleep. The next time an item is consumed, emptyCount is incremented and fillCount is decremented. When all the items are consumed, fillCount becomes zero and the producer wakes up. The consumer works analogously. Finally, semaphore mutex ensures that only one consumer is accessing the shared memory at a time.

8.2 Client-Server Version

In order to use multiple computer systems within a cluster or in a network, we implement a client-server version as shown in Figure 4. This version uses sockets for communication between processes running on different computers.

How server works. When the server is up and running, like the multi-process version, it stores all tasks up to a certain recursion depth in a file. A producer reads these tasks from the input file and supplies them to the consumers through a shared memory. The consumers do not themselves perform the search. They rather pass these tasks to the producer on the client computers. When the producer reaches *end-of-file*, it writes NULL to the shared memory. The consumer then reads and sends NULL to the client indicating that there is no further task available. After getting an acknowledgement from the client that it has received NULL, the consumer becomes dead. When all the consumers are dead, server terminates.

How client works. The producer on a client computer gets the task from the consumer on the server side and supplies these tasks to the consumers on the same computer through a shared memory. The consumers on the client computers actually perform the search. When the producer receives NULL, it writes it to the shared memory and sends a termination acknowledgement to the server. Whenever a consumer reads NULL, it terminates. When all the consumers are dead, client terminates.

8.3 Experimental Results

The parallel versions are also implemented in c++ and run on the same computers that we use to run IP and LC algorithms. We do not run the problems of size 21 and below. This is because these problems are too small to be suitable candidate for the parallel versions. Our main results are however shown on size 24 and above, where enormous computing effort is required.

Table 7 Performance of parallel versions of lazy checking (Time in Hours). The total number of AIS solutions can be obtained by multiplying #Sols by 8

Size	Execution Time		Total Time		#Solutions
	MP	**SC**	**MP**	**SC**	
24	1.53	0.7	11.39	11.98	39,404,484
25	10.07	3.29	80.48	77.22	**126,853,094**
26	75.62	13.81	584.93	551.42	**495,460,562**
27		106.35		4248.14	**1,645,766,269**

MP:Multi-Process SC:Server-Client #client 5 #process/client 8 task-level 3

Table 7 shows the performance of the parallel versions of our LC algorithm. The multi-process version runs on one computer while client-server version runs on 5 computers at a time, each one having 8 processes simultaneously. Each process was given a *task of level* 3 meaning a partial RAIS of size 6: first 3 integers are fixed initially and then 3 more integers are found by recursion up to depth 3. Here the *execution time* denotes the maximum time taken by a single process over all simultaneously running processes while the *total time* is the summation of their CPU-times. Notice that our multi-process and client-server versions respectively generate all RAIS of size 26 and 27 in about 551 CPU hours (23 days) and 4248 CPU hours (177 days). Our parallel versions allowed us to obtain these results within 5 real days.

We observe that the total time in client-server version is less than the multi-process version which takes less total time than the single process version. The reason is the well-known obvious inherent parallelism in the hardware that in general allows the delay in memory IO operation to be better utilised by the processor to execute some other process, in this case a related process.

Table 8 Varying the task level in client-server version

Size	Execution Time (mins)			Total Time (mins)		
	2	**3**	**4**	**2**	**3**	**4**
22	0.89	0.40	0.40	21.89	16.98	17.49
23	5.45	4.19	4.71	144.92	145.08	154.97
24	20.29	17.89	19.90	573.65	572.26	627.53

Table 9 Varying the number of clients in client-server version

Size	Execution Time (mins)			Total Time (mins)		
	3	**4**	**5**	**3**	**4**	**5**
22	0.91	0.56	0.40	21.89	17.54	16.98
23	6.04	4.78	4.19	144.82	150.67	145.08
24	25.95	26.64	17.89	621.92	706.91	572.26

Table 10 Varying the number of process per client in client-server version

Size	Execution Time (mins)			Total Time (mins)		
5.45	**4**	**6**	**8**	**4**	**6**	**8**
22	1.30	0.77	0.40	20.61	18.41	16.98
23	8.32	5.33	4.19	145.48	145.25	145.08
24	35.51	25.06	17.89	609.31	581.03	572.26

Table 8-10 show the execution time and total time taken by different configurations of our client-server version using the LC algorithm. By varying the task-level for sizes 22-24, we found that the performance is optimum when the task-level is 3. This is due to the trade-off between the benefit of parallelism and the overhead of communication. For task-level above 3, because of the increased recursion depth, the number of task increases drastically but each task needs less search time. This eventually causes enormous communication overhead. Hence, for problems of size 25 and above, where total times required are enormous, we run the client-server version with task-level 3. Having discussed this, we still do not report the overhead times in this paper. This is because we found for task-level 3, the overhead time measured is negligible compared to the total time required to solve the problem. Moreover, the overhead of using producer processes is in general found to be very negligible as well.

However, varying the number of clients and the number of processes per client almost proportionately result into varying execution time. As noted before, more parallelism however leads to reduced total time. Currently, due to hardware availability, we can run only up to 5 clients each having at most 8 processes. Beyond these points, we therefore cannot investigate existence of any practically optimum configuration.

9 Adding New Constraints to Our Algorithm

Our search algorithm for RAIS uses a constraint-driven approach, which offers the flexibility to add new RAIS constraints easily. For example, from the pseudocode, one can easily see how our auxiliary constraints are added to the algorithm. Moreover, we can show another example of adding a new constraint to our algorithm. In RAIS we observe the following conjecture to hold for all results up to size 48. If used as a constraint, the conjecture improves the time performance of the search by 8-10%. Since we can not prove it in general, we leave it as an open question and do not include in our results.

Conjecture 1 (Fourth Integer) For $n > 5$, the minimum integer possible immediately after $(0, n-1, 1)$ is $(n+1)/3$

Assuming the conjecture to hold just for the sake of exposition, we just need to add the following line in proc deepen() of algo backtracking.

$$\mathsf{if}(\mathsf{cs} == 3 \text{ and } k < (n+1)/3) \text{ return};$$

If any new RAIS properties are identified later, the related constraints could thus be incorporated in the algorithm very easily.

10 Improving Non-Constraint-Based Search

A non-constraint-based breadth-first search algorithm is presented in [3] to count the graceful n-permutations for paths, which is actually equivalent to solving the AIS counting version. The breadth-first search runs for levels $n-1$ to 1. At each level l, partial solutions (also called nodes) encompassing differences $n-1$ to $l+1$ from the previous level are taken and all possible ways of encompassing difference l are explored. In the example shown below, a partial solution $\langle 5,0,6,2\rangle\langle 1,4\rangle$ encompasses differences 6, 5, 4, and 3. Notice that a partial solution comprises a number of strings e.g. $\langle 5,0,6,2\rangle$ and $\langle 1,4\rangle$. Now we can obtain difference 2 in three ways: *i*) putting 2 and 4 one beside another and thus we get $\langle 5,0,6,2,4,1\rangle$; *ii*) putting 3 before 1 and thus we get $\langle 5,0,6,2\rangle\langle 3,1,4\rangle$; and *iii*) putting 3 before 5 and thus we get $\langle 3,5,0,6,2\rangle\langle 1,4\rangle$.

$$\langle 5,0,6,2\rangle\langle 1,4\rangle$$

$$\langle 5,0,6,2,4,1\rangle \qquad \langle 5,0,6,2\rangle\langle 3,1,4\rangle \qquad \langle 3,5,0,6,2\rangle\langle 1,4\rangle$$

Now, we can see that among the newly obtained partial solutions, the two $\langle 5,0,6,2\rangle\langle 3,1,4\rangle$ and $\langle 3,5,0,6,2\rangle\langle 1,4\rangle$ could be considered equivalent. This is because using Definition 2, we can find $\langle 3,5,0,6,2\rangle\langle 1,4\rangle$ to be equivalent to $\langle 3,1,6,0,4\rangle\langle 5,2\rangle$. Then, comparing $\langle 5,0,6,2\rangle\langle 3,1,4\rangle$ with $\langle 3,1,6,0,4\rangle\langle 5,2\rangle$, we see that both sides of 0, 6, and 1 are occupied in the two partial solutions; and one sides of 2, 5, 3, 4 are still available to create any other differences; and 2, 5 cannot be one beside another with the same being true for 3, 4. The conditions necessary and sufficient for such equivalence are defined in [3]. When two such partial solutions are equivalent, only one could be kept. Moreover, we need to remember the number of such equivalent classes so that the number of complete solutions obtained from this partial solution is then multiplied by that number to obtain the total number of solutions found for the entire equivalence class. Two arrays Free and Forb help determine which numbers are yet free and which pairs of numbers cannot be one beside another. These two arrays are the only information that is used to represent the equivalence classes and no information is kept to remember which number is exactly beside which others. Below we show the pseudocode of the algorithm in [3].

```
        algo Counting

proc countAIS(Size)
    solnCount = 0, nodeCount = 0
    hashTab1.add(⟨0, n − 1, 1⟩)
    while (hashTab1.Count != 0)
        nodeCount += hashTab1.Count
        foreach (node n ∈ hashTab1)
            l = n.level-1 // difference to create
            foreach (Size − l > j ≥ 0)
```

```
                if (!n.Free[j]) continue; // j not free
                k = j + l // the other number to use
                if (!n.Free[k]) continue; // k not free
                if (n.Forb[k] ==j) continue;
                if (l == 1)
                    solnCount += n.EquiCount
                else
                    expand(j, k, n, hashTab2)
        hashTab1 = hashTab2, hashTab2.reset()
proc expand(i, j, n, hashTab2)
    // put i, j one beside another
    −−Free[i], −−Free[j]
    f1 = Forb[i], f2 = Forb[j]
    Forb[f1] = f2, Forb[f2] = f1
    if (n is equivalent to n' ∈ hashTab2)
        n'.EquiCount += n.EquiCount
    else // a new partial solution
        hashTab2.add(n) // for next level
```

As we can see the algorithm is not a constraint-based approach rather it is a straightforward tree search procedure that constructs members of the equivalence classes representing partial or complete solutions. To make it clearer, the algorithm does not deal with the concept of variables and assigning values to the variables satisfying the constraints. It only handles with values and constructs members of the equivalence classes where the members comprise a number of strings encompassing the higher differences so far obtained. From the representation of the members of an equivalence class, it is not clear exactly which value will be eventually be assigned to which position of the complete AIS solution. Moreover, no information is kept about which numbers appear beside which number. Thus generation of the actual AIS from the members of equivalence classes is not straightforward and requires further search. Nevertheless, using the AIS counting algorithm mentioned above, the numbers of AIS are reported up to size 40 in the file https://oeis.org/A006967/b006967.txt.

Although our main focus in this paper is rather to generate all unique AIS solutions than just to count them, we further try to improve the efficiency of the counting algorithm in [3] in terms of the nodes generated or expanded by the breadth-first search. Note that generating the solutions i.e. assignment of values to variables is normally the focus of the constraint community as well. Nevertheless, we tried to adopt our new RAIS properties e.g. Lemma 4, Lemma 8, Corollary 1 to AIS but since the algorithm in [3] is a non-constraint-based algorithm, it was very difficult to do so. We only achieved some improvement by using the following lemma which mainly comes from the definition of AIS. Application of the lemma in the way shown is actually inspired by the lazy checking that we perform in our constraint-based search for RAIS.

Lemma 9 *After generating a partial solution at level l, if it is not possible to generate all the other differences from $l-1$ to 1 using the free numbers and satisfying their mutual forbidding relations, then the partial solution can be discarded at that level. It will eventually produce no AIS solution.*

Proof All difference from $n-1$ to l are already present in the partial solution. So the remaining difference $l-1$ to 1 must be obtained. □

Lemma 9 is incorporated in proc expand below. A search node is first checked whether it satisfies Lemma 9 or not and if yes then the equivalence checking or adding to the hash table is performed.

```
proc expand(i, j, n, hashTab2)
    // put i, j one beside another
    −−Free[i], −−Free[j]
    f1 = Forb[i], f2 = Forb[j]
    Forb[f1] = f2, Forb[f2] = f1
    // Lemma 9 below
    if (n.Level < α)
        foreach (n.Level−1 > l > 0)
            foreach (Size −l > j ≥ 0)
                if (!n.Free[j]) continue;
                k = j + l
                if (!n.Free[k]) continue;
                if n.Forb[k] == j) continue;
                break
            if (j ≠ size − l) return
    // Lemma 9 above
    if (n is equivalent to n′ ∈ hashTab2)
        n′.EquiCount += n.EquiCount
    else // a new partial solution
        hashTab2.add(n) // for next level
```

In our experiments, we found that up to a certain level, the search does not discard any partial solutions. After that level (represented by α in the pseudocode), search starts to discard significant numbers of partial solutions. Thus to reduce the overhead, we look for an empirically better value of α. So we tried different values for α and as Table 11 shows the algorithm is the most efficient when α is set to 4. So this value is used in further experiments.

Table 11 parameter tuning for α; Time to run the algorithm is shown here in seconds

Size	0	3	4	5	6	7	$n/2$	n
35	4.09	4.08	**4.07**	4.09	4.15	4.20	4.44	4.38
36	4.61	4.59	**4.50**	4.60	4.68	4.78	5.13	5.05
37	6.18	6.18	**6.17**	6.21	6.31	6.46	6.98	6.79
38	7.72	7.71	**7.69**	7.77	7.91	8.05	8.76	8.62
39	11.25	11.2	**11.2**	11.3	11.5	11.7	12.7	12.6
40	15.33	15.2	**15.2**	15.3	15.6	15.8	17.4	17.04
41	29.08	29.02	**28.9**	29.01	29.3	29.7	31.7	31.42
42	30.97	30.8	**30.7**	30.96	31.4	31.9	35.1	34.45

0 means without our heuristics; n: size of the series

Using $\alpha = 4$, we then run experiments for size 40–48 and Table 12 shows the results. As we see from the table, the reduction in the numbers of nodes using the lemma is 14-50 thousand. Although the lemma does not lead to a very significant improvement in time efficiency of the counting version of the

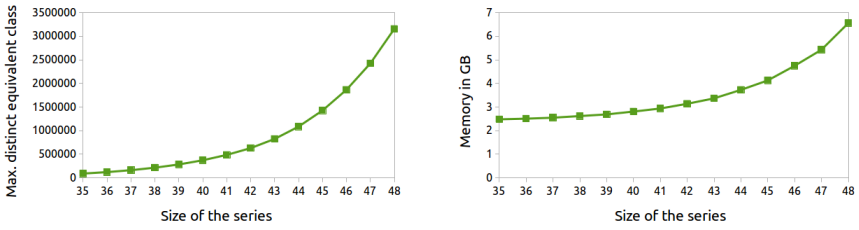
AIS problem, we will later show that it does save significant time in solving the generation version. Nevertheless, note that the numbers of solutions for sizes 41-48 are reported here for the first time in the literature.

Table 12 Effectiveness of our heuristics in counting solutions. The #AIS Solutions is obtained by multiplying solnCount of algo Counting by 4.

Size	#AIS Solutions	#Visited nodes			Time in sec	
		UL	**NL**	**NR**	**UL**	**NL**
40	257148148674392752	2293453	2308298	**14845**	16.00	16.13
41	1032009807743958000	3050401	3067838	**17437**	24.56	24.86
42	4776537381650913456	4046772	4067020	**20248**	29.83	30.14
43	19563451130134939776	5353260	5377041	**23781**	46.41	46.96
44	71216650440776894752	7065860	7093970	**28110**	62.53	62.93
45	224367172460076921168	9308469	9341054	**32585**	96.39	97.54
46	468624231739441973664	12241960	12279271	**37311**	128.38	128.92
47	700989664194333458560	16051693	16094895	**43202**	212.62	214.58
48	754293465264018719232	21014722	21065113	**50391**	298.80	301.98

UL: using Lemma 9; NL: not using Lemma 9; NR: #Node reduction

Memory is always a concern for breadth-first search. In the AIS counting algorithm described above, all nodes at the same level of the search tree are stored in a hash table for quick checking of any potential equivalence (if any). A node whose equivalent is already in the hash table is not added again, rather the equivalence count of the node already in the hash table is incremented. Fig. 5 shows the maximum numbers of nodes to hold at one level and the required memory to run the algorithm. As we can see, both the numbers of distinct equivalent classes and memory requirement increases steadily with the increase in size of the series; the rate of increase grows further after 40.



(a) (b)

Fig. 5 For each size of the series, (a) maximum number of partial solutions kept in memory and expanded at a time (b) required memory to run the algorithm

Since our goal is to solve the generation version of the AIS problem and thus generate all the solutions of size n, we also extend the algorithm of [3] to do so. As noted before, the counting algorithm of [3] still requires search to generate the solutions of an AIS sequence if an equivalent class is given. This is because from one member of an equivalent class, there is no straightforward

way to generate all the other solutions of that class. Also the drawback of the algorithm in [3] is that it can not eliminate conditional symmetry. Since it deals with AIS not RAIS, the repeated difference only appears between first and last number of a sequence. Thus, the conditional symmetry can only be detected after generating the solution but not during search. For this reason, this approach generates all the conditionally symmetric redundant solutions as well. Since in solving the generation version of the problem, we have to explore all the members in an equivalence class meaning no equivalence checking is needed, we have implemented our non-constraint-based AIS generation algorithm using the depth-first approach. For the generation version, besides the arrays Free and Forb, we have to store at each node the information of which value is beside which others so that the actual solution could be computed. The algorithm to generate all the solutions is shown in algo Generating. As we can see, it requires two more arrays to hold the partial solutions and then to expand them. As expected, to implement the depth-first search for generating all the solutions, a stack is used in proc genAIS. The only modification in proc expand is the addition of proc place(). This new procedure finds the strings having i and j at their end point and put them one beside another in a single string, and thus generating an AIS solution.

algo Generating

```
int length = 0;
string holdSoln[];//holds the solution strings
int strPos[];//indicates the position of a partial solution string
proc genAIS(Size)
    solnCount = 0, nodeCount = 0
    stack1.add(⟨0, n − 1, 1⟩)//add will increase the value of length by 1
    while (length != 0)
        node n = stack1.node[−−length])
        l = n.level-1 // difference to create
        foreach (Size − l > j ≥ 0)
            if (!n.Free[j]) continue; // j not free
            k = j + l // the other number to use
            if (!n.Free[k]) continue; // k not free
            if (n.Forb[k] ==j) continue;
            if (l == 1)
                ++solnCount;
                place(j,k);
            else
                expand(j, k, n)
                ++nodeCount;
proc expand(i, j, n, hashTab2)
    // put i, j one beside another
    −−Free[i], −−Free[j]
    f1 = Forb[i], f2 = Forb[j]
    Forb[f1] = f2, Forb[f2] = f1
    place(i, j);
    put Lemma 9 as in algo Counting
    stack1.add(n) // for next level
proc place(i, j)
    si = holdSoln[strPos[i]]
```

```
    sj = holdSoln[strPos[j]]
  if (si.begin == i)                    else if (si.end == i)
    if (sj.begin == j)                    if (sj.end == j)
      strPos[sj.end] = strPos[i];           strPos[si.begin] = strPos[j];
      reverse(si);                          reverse(si);
      si.append(sj);                        sj.append(si);
      return strPos[i];                     return strPos[j];
    else if (sj.end == j)                 else if (sj.begin == j)
      strPos[si.end] = strPos[j];           strPos[sj.end] = strPos[i];
      sj.append(si);                        si.append(sj);
      return strPos[j];                     return strPos[i];
```

Table 13 shows the effectiveness of Lemma 9 in generating all the solutions. It shows that using Lemma 9, we have to visit 4-5% fewer nodes than not using the lemma. This saving in the nodes eventually gives 11-17% improvement in time. We run this non-constraint-based algorithm to generate all AIS solutions of sizes up to 29 while the known results in the literature for the generation version is up to size 24 and that is by using a SAT-based approach.

Table 13 Effectiveness of Lemma 9 in generating all the solutions

Size	#Visited nodes			Time in sec		
	UL	**NL**	**Imp(%)**	**UL**	**NL**	**Imp(%)**
22	24635396	25687181	**4.09**	27.24	30.79	**11.53**
23	78531568	82165849	**4.49**	89.17	103.55	**13.89**
24	259159314	270718477	**4.26**	269.48	311.11	**13.38**
25	259159314	270718477	**4.26**	930.57	1066.7	**12.76**
26	2819839451	2978057657	**5.31**	3097.23	3569.31	**13.23**
27	9603488795	10173133374	**5.60**	10434.1	12782.4	**18.37**
28	33803325796	35662581184	**5.21**	37302.8	43514.5	**14.27**
29	118854828050	125698700875	**5.45**	127146	153825	**17.34**

UL: Using Lemma 9; NL: Not using Lemma 9

11 Conclusion

This paper presents a constraint-based approach to solve the all-interval series problem. It exhibits new properties of the reformulated all-interval series which prune the search space significantly. It provides a backtracking-based tree search algorithm which incorporates these new properties. It presents a very efficient lazy checking technique that performs significantly better in finding reformulated AIS than the typical immediate propagation technique used in constraint solvers. It also presents scalable parallel versions of this algorithm that take the advantage of multi-core processors and even multiple computer systems. Finally, our algorithm generates all the unique solutions of the all-interval series up to size 27. To solve the AIS problem for greater sizes using constraint-based approaches, one might need to find more efficient pruning techniques. Despite the use of our new properties, empirical analysis shows that search goes deep down a branch before realising that there is no

solution in that branch. Thus pruning techniques that can cut a branch with no solution at the start of that branch will be very effective. So, the future direction to solve the AIS problem with constraint-based approach will be to find some other properties that can help to prune the useless branches.

References

1. Ilog solver. http://www.cs.cornell.edu/w8/iisi/ilog/cp11/usrsolver/usrsolverpreface.html
2. Minizinc challenge 2013 results. https://www.minizinc.org/challenge2013/results2013.html
3. Adamaszek, M.: Efficient enumeration of graceful permutations. arXiv preprint math/0608513 (2006)
4. Alsinet, T., Bejar, R., Cabiscol, A., Fernandez, C., Manya, F.: Minimal and redundant SAT encodings for the all-interval series problem. In: Topics in Artificial Intelligence, 5th Catalonian Conference on AI (CCIA), LNCS, vol. 2504, pp. 139–144 (2002)
5. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. Journal of Mathematical and Computer Modelling **20**, 97–123 (1994)
6. Bjar, R., Many, F., Cabiscol, A., Fernndez, C., Gomes, C.: Regular-sat: A many-valued approach to solving combinatorial problems. Discrete Applied Mathematics **155**(12), 1613–1626 (2007)
7. Choi, C., Lee, J.: On the pruning behaviour of minimal combined models for permutation csps. In International Workshop on Reformulating Constraint Satisfaction Problems (2002)
8. Codognet, P., Diaz, D.: Yet another local search method for constraint solving. In: Proceedings of the International Symposium on Stochastic Algorithms: Foundations and Applications, SAGA '01, pp. 73–90. Springer-Verlag (2001)
9. Colles, H.: Grove's dictionary of music and musicians. The MacMillan Company, New York (1940)
10. Dent, M.J., Mercer, R.E.: Minimal forward checking. In: Proc. of the 6th international conference on tools with Artificial Intelligence, pp. 432–438. IEEE (1994). DOI 10.1109/TAI.1994.346460
11. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver Clasp. Lecture Notes in Computer Science, Springer **5753**, 509–514 (2009)
12. Gent, I.P., McDonald, I., Smith, B.M.: Conditional symmetry in the all-interval series problem. In: Proceedings of the 3rd International Workshop on Symmetry in Constraint Satisfaction Problems, vol. 3, pp. 55–65 (2003)
13. Hoos, H.H.: Stochastic local search - methods, models, applications. Ph.D. thesis, Department of Computer Science, Technical University of Darmstadt, Germany (1998)
14. Hudson, S.: Incremental attribute evaluation: A flexible algorithm for lazy update. ACM Transactions on Programming Language and Systems **13**(3), 315–341 (1991). DOI 10.1145/117009.117012
15. Krenek, E.: Horizons circled-reflections on my music. University of California Press, Berkeley (1974)
16. Moisan, T., Gaudreault, J., Quimper, C.G.: Parallel discrepancy-based search. In: International Conference on Principles and Practice of Constraint Programming, pp. 30–46. Springer (2013)
17. Newton, M., Pham, D., Sattar, A., Maher, M.: Kangaroo: An efficient constraint-based local search system using lazy propagation. CP. LNCS, Springer, Heidelberg **6876**, 645–659 (2011)
18. Nguyen, V., Son, M.: Solving the all-interval series problem: SAT vs CP. In: Proceedings of the 5th symposium on Information and Communication Technology, pp. 65–74 (2014)
19. Petrie, K.E., Smith, B.M.: Symmetry breaking in graceful graphs. In: Proceddings of Principles and Practice of Constraint Programming (F.Rossi ed.). Springer (2003)
20. Puget, J., Regin, J.: Solving the all-interval problem. http://www.cs.st-andrews.ac.uk/ ianm/CSPLib/prob/prob007/puget.pdf
21. Remzi, H.A.D., Andrea, C.A.D.: Operating Systems: Three easy steps. Arpaci-Dusseau books (2014)

22. Schuurmans, D., Southey, F.: Local search characteristics of incomplete SAT procedures. In: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), Austin/TX, USA, pp. 297–302. AAAI Press (2000)
23. Simonis, H., Beldiceanu, N., Sa, C.: A note on CSPLIB prob007. Tech. rep., Normale (1998)
24. Solnon, C.: Solving permutation constraint satisfaction problems with artificial ants. In: Proceedings of ECAI'2000, pp. 118–122. IOS Press (2000)
25. Toro, M., Rueda, C., Agon, C., Assayag, G.: Gelisp: A library to represent musical csps and search strategies. Computing Research Repository **abs/1510.02828** (2015)
26. Truchet, C., Richoux, F., Codognet, P.: Prediction of parallel speed-ups for las vegas algorithms. CoRR **abs/1212.4287** (2012)
27. T.Walsh: Parameterized complexity results in symmetry breaking. In: Proceedings of 5th Internatinal symposium, IPEC, Chennai, India, LNCS, vol. 6478, pp. 4–13. Springer (2010)
28. Worister, M., Steinlechner, H., Maierhofer, S., Tobler, R.: Lazy incremental computation for efficient scene graph rendering. In: Proc. of the 5th High-Performance Graphics Conference, pp. 53–62. ACM, New York, NY, USA (2013). DOI 10.1145/2492045.2492051
29. Yadav, S.C., Singh, S.K.: An introduction to Client/Server Computing. New Age International Publishers Limited (2009)