A Bitwise GAC Algorithm for Alldifferent Constraints

Zhe Li[1] , **Yaohua Wang**[1*] , **Zhanshan Li**[2*]
[1]National University of Defense Technology, Changsha, China
[2]Jilin University, Changchun, China
leezear@163.com, nudtyh@foxmail.com, lizs@jlu.edu.cn

Abstract

The generalized arc consistency (GAC) algorithm is the prevailing solution for alldifferent constraint problems. The core part of GAC for alldifferent constraints is excavating and enumerating all the strongly connected components (SCCs) of the graph model. This causes a large amount of complex data structures to maintain the node information, leading to a large overhead both in time and memory space. More critically, the complexity of the data structures further precludes the coordination of different optimization schemes for GAC. To solve this problem, the key observation of this paper is that the GAC algorithm only cares whether a node of the graph model is in an SCC or not, rather than which SCCs it belongs to. Based on this observation, we propose Alldiff$^{\text{bit}}$, which employs bitwise data structures and operations to efficiently determine if a node is in an SCC. This greatly reduces the corresponding overhead, and enhances the ability to incorporate existing optimizations to work in a synergistic way. Our experiments show that Alldiff$^{\text{bit}}$ outperforms the state-of-the-art GAC algorithms over 60%.

1 Introduction

In constraint programming (CP), many problems can be modeled using the alldifferent constraint. This constraint forces every constrained variable to take distinct values [Lauriere, 1978; Bessiere *et al.*, 2009]. Constraint propagation is a core process for filtering fruitless search space to solve CP problems. At present, there are many filtering algorithms for propagating alldifferent constraint. They can be classified into bound consistency (BC) [López-Ortiz *et al.*, 2003], range consistency (RC) [Leconte, 1996] and generalized arc consistency (GAC) [Régin, 1994] according to their pruning ability (consistency) from weak to strong. Among them, GAC is more more suitable for solving hard problem instances due to its stronger pruning ability via more complex inference.

The classic GAC filtering algorithm for the alldifferent constraint was presented by Régin [Régin, 1994] (hereinafter, called Régin algorithm). It computes a maximum matching, enumerates strongly connected components (SCCs), and then removes the unmatched edges between independent SCCs (i.e., redundant edges) in the residual graph of the constraint, which is based on a corollary of Berge [Berge, 1973]. The most time-consuming part of the this algorithm is the excavation and enumeration of SCCs, which is commonly solved via the Tarjan algorithm [Tarjan, 1972].

To improve the performance of the Tarjan algorithm, many studies have been carried out to reduce the number of SCCs that need to be computed. Examples include: 1) SCC-splitting [Gent *et al.*, 2008], which only computes an SCC where the domain of its variable has been changed during the backtracking search, thus reducing the range of variables required to compute SCCs. 2) Bitwise operation [Van Kessel and Quimper, 2012], which employs a bitwise approach to accelerate propagation based on the WordRam model and have always been an important means of accelerating consistency algorithms [Lecoutre and Vion, 2008; Wang *et al.*, 2016; Demeulenaere *et al.*, 2016; Li *et al.*, 2021]. 3) Matching optimization [Zhang *et al.*, 2018], which proves that redundant edges can be divided into two classes, one of them can be removed directly without computing the corresponding SCCs. 4) Early detection [Zhang *et al.*, 2020], which avoids the computation of the unimportant edges whose deletion will not split the current SCCs.

However, all of the optimizations mentioned above involve excavating all strongly connected components (SCCs) and maintaining corresponding information using complex data structures such as queues, stacks, traversal orders, and SCC IDs for each node. The GAC algorithm is embedded in the entire backtracking search solution framework and is invoked iteratively, frequently updating and resetting these complex data structures. This causes a significant overhead in both execution time and memory space. Furthermore, due to the complexity of the data structures, it is difficult to integrate different optimization schemes in a synergistic way, resulting in lower overall efficiency.

In fact, our analysis shows that the essential task of the GAC algorithm is to delete inconsistent nodes. Therefore, it only cares whether a node in the graph is in an SCC or not, rather than which SCC it belongs to. Based on this key observation, we transform a 'which' problem into a 'yes-or-no' one. This enables us to propose a bitwise GAC al-

*Corresponding authors

gorithm called Alldiff^{bit}. Alldiff^{bit} employs complete bit-wise data structures and performs the 'yes-or-no' problem via simple bitwise 'OR' operations during the traversal of the bit-adjacency matrix of the graph model. Additionally, the bitwise data structures and operations also enable Alldiff^{bit} to easily incorporate existing optimizations, including: 1) SCC-Splitting by improving the SCC partition representation mechanism; 2) matching optimization via bit-BFS of adjacency bit-matrix; and 3) early detection by prioritizing the checking of the connectivity of deleted edges. These existing optimizations can work organically when integrated into Alldiff^{bit}.

Our experiments on large numbers of constraint problems (CPs) show that Alldiff^{bit} is both efficient and stable. Compared to the GAC algorithms implemented by state-of-the-art CP solver [Prud'homme *et al.*, 2017], Alldiff^{bit} achieves an average performance speedup of 60.65% per group across 18 series instances. We believe that Alldiff^{bit} is a powerful contender against existing algorithms.

2 Background and Preliminaries

A constraint satisfaction problem (CSP) $\mathcal{P}$ is a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, in which $\mathcal{X}$ is a set of n variables $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, $\mathcal{D}$ is a set of domains $\mathcal{D} = \{D(x_1), D(x_2), ..., D(x_n)\}$, where $D(x_i)$ is a finite set of possible values for variable x_i, and $\mathcal{C}$ is a set of e constraints $\mathcal{C} = \{c_1, c_2, ..., c_e\}$. A constraint c restricts an ordered set of variables $X(c) = \{x_1, x_2, ..., x_r\}$, and also restricts a subset of the Cartesian product $D(x_1) \times D(x_2) \times ... \times D(x_r)$, which specifies the allowed combinations of values for $X(c)$. The domain of c is the union of domain, which is denoted by $D(c) = \bigcup_{x \in X(c)} D(x)$. In addition, we use $B_c(a)$ to denote the set of variables in $X(c)$ whose domain contains a (i.e., $B_c(a) = \{x \mid x \in X(c), a \in D(x)\}$). A solution to a CSP is an assignment of a value to each variable, so that all the constraints are satisfied.

Definition 1 (Generalized Arc Consistency, GAC)**.** *Given a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$:*

- *A constraint c is GAC iff $\forall\ x \in X(c)$ and $\forall\ a \in D(x)$, there exists a support for (x, a) on c.*
- *A constraint network $\mathcal{P}$ is GAC iff all constraints in $\mathcal{C}$ are GAC.*

An alldifferent constraint c: $\{x_1, ..., x_r\}$ $(r > 1)$ is an r-arity constraint that ensures that the values assigned to constrained variables are all distinct. The GAC algorithms are based on Berge's graph theory, particularly on maximum matching and strongly connected components. A matching is a set of edges that do not have common vertices. In this paper, if variable x is matched to value a, it can be written as $x = M(a)$ and $a = M(x)$, otherwise $M(a) = \perp$ and $M(x) = \perp$. A maximum matching is one that has maximum cardinality. If a node is connected with a matched edge, it is called a matched node; otherwise, it is called a free node. An alternating path is a path whose edges alternate between matched and unmatched nodes. An augmenting path is a path whose starting and ending nodes are both free nodes.

Definition 2 (Residual Graph)**.** *The residual digraph is defined as $R = \langle V_R, E_R \rangle$, where $V_R = X(c) \cup D(c) \cup \{t\}$, t is a sink node linked to $D(c)$, $E_R = E_M \cup E_U \cup E_{t_1} \cup E_{t_2}$. Specifically, the matching edges E_M connect variables to their respective matching values: $E_M = \{x \mapsto a \mid x \in X(c), a = M(x)\}$. The unmatched edges E_U connect values to their respective unmatching variables: $E_U = \{a \mapsto x \mid a \in D(c), x \in B_c(a) \setminus \{M(a)\}\}$. The edges in E_{t_1} connect matching value nodes to t: $E_{t_1} = \{a \mapsto t \mid M(a) \in X(c)\}$. The edges in E_{t_2} connect t to unmatching value nodes (i.e., free nodes): $E_{t_2} = \{t \mapsto a \mid M(a) \notin X(c)\}$.*

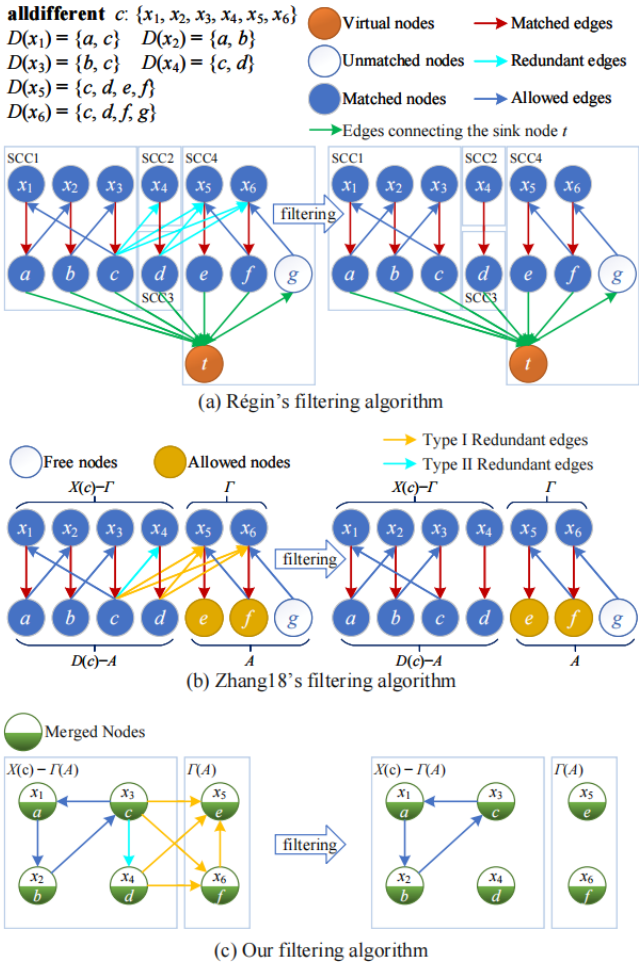

Figure 1: Illustration of arc consistency filtering algorithms.

Each edge $e \in E_M \cup E_U$ represents a variable-value pair in constraint c. A redundant edge is one that does not appear in any maximum matching. An allowed edge is one belonging to some but not all maximum matchings. A node is allowed, iff, for an arbitrary maximum matching M, it can be reached by an even alternating path that begins at a free node [Zhang *et al.*, 2017]. The set of variable nodes in alternating paths is denoted by Γ, and its value nodes set is denoted by A.

Definition 3 (Redundant Edge)**.** *For an arbitrary maximum matching, an unmatching edge $e_{a \mapsto x}$ is redundant iff (1) $x \in \Gamma$ and $a \in D(c) - A$, or (2) $x \in X(c) - \Gamma$, $a \in D(c) - A$ and $e_{a \mapsto x}$ does not belong to any alternating cycle [Zhang* et al.*, 2018].*

In this paper, we call these two type of redundant edges as "type I" and "type II" redundant edges, respectively. Zhang18 algorithm (as shown in Figure 1(b)) develops the graph theory of Régin algorithm (as shown in Figure 1(a)) by removing type I redundant edge without computing SCCs. For a residual graph $R = \langle V_R, E_R \rangle$ and a set of (deleted) edges $DE = e_1, e_2, ..., e_k$, where $e_i \in E_M \cup E_U$,$(1 \leq i \leq k)$, $R' = \langle V_R, E_R \setminus DE \rangle$ is consistent, iff, for every start-end node pair (x_i, y_i) of e_i in DE, there exists at least two arc-disjoint alternating paths connecting x_i and y_i [Zhang *et al.*, 2020]. These edges are called unimportant edges. To be specific, Zhang18's algorithm develops Régin's algorithm by adding the following steps 3 and 4: (1) Find a maximum matching M from variables to distinct values. If it fails[1], the constraint is inconsistent. (2) Construct the residual graph R (see Definition 2). (3) Find all alternating paths from all free nodes in R, and put the variable (resp. value) nodes of the alternating paths into Γ (resp. A). (4) Remove all type I redundant edges without finding SCCs. (5) Find all SCCs of R and remove all redundant edges. If R does not have free nodes, steps 3 and 4 will not be executed, and this algorithm will be degraded to Régin's algorithm. Zhang20's algorithm improves step 5 by detecting whether each deleted edge e is surrounded by a cycle during the DFS for finding SCCs. If so, it is an unimportant edge. If all edges are unimportant, this propagation can be exited early.

3 The AllDiff$^{\text{bit}}$ Algorithm

In this section, we propose the Alldiff$^{\text{bit}}$ algorithm. Instead of enumerating SCCs to check whether the variable values satisfy GAC, Alldiff$^{\text{bit}}$ directly checks their connectivity in the adjacency bit-matrix of the graph. This simplifies the data structures and comprehensively bit-vectorizes them. Furthermore, it enables the synergistic integration of existing optimizations. Specifically, Alldiff$^{\text{bit}}$ first reduces the graph model by merging matched nodes and uses the adjacency bit-matrix to represent the merged graph (as depicted in Figure 1(c) and Table 1). Then, it checks the consistency of each variable-value by checking the connectivity of its corresponding edge in a bit-BFS manner. Furthermore, we extend Alldiff$^{\text{bit}}$ to accommodate various existing improvements. For SCC Splitting [Gent *et al.*, 2008], our algorithm gives priority to checking the edges that are mutually connected by using a new edge checking order. This order ensures that a complete traversal of an entire SCC is performed when the traversal stops. For matching optimization [Zhang *et al.*, 2018], we also bitwise vectorize the A and Γ sets and quickly calculate them by bit-BFS. For early detection [Zhang *et al.*, 2020], we preferentially traverse the deleted edges, and the propagation can quit early if they all pass the connectivity check.

Bitwise representation of merged graph. When computing SCCs of the reduced residual graph R, the existing algorithms alternately traverse matching and unmatching edges. To eliminate matching edges, the variables and values that are matched can be merged. This results in a merged node that inherits all incoming edges of the original variable node and all outgoing edges of its matched value node. As a result, we obtain a smaller directed graph $G = \langle V, E \rangle$, where $V = X(c)$, $E = \{x \mapsto y \mid x \in X(c), y \in B_c(M(x)) \setminus \{x\}\}$ (see Figure 1(c)). This reduction helps to decrease the size and order of the graph. Based on the above, we introduce the adjacency bit-matrix $\boldsymbol{G}$ to represent the merged graph[2]. If node x can reach node y in the merged graph, then $G[x][y] = 1b$, otherwise $G[x][y] = 0b$. The adjacency bit-matrix $\boldsymbol{G_F}$ represents the frontier nodes to be expanded during BFS. If x needs to extend y then $\boldsymbol{G_F}[x][y] = 1b$, otherwise $\boldsymbol{G_F}[x][y] = 0b$. If a value (x, a) is valid (i.e., $a \in D(x)$ and $x \in B_c(a)$), then $\boldsymbol{B}[a][x] = 1b$, otherwise $\boldsymbol{B}[a][x] = 0b$. Table 1 gives the bitwise representation of Figure 1(c). Method *mergeMatchedNodes* of algorithm 1 initializes the above fields.

	$x_1 ... x_6$
$\boldsymbol{B}[a]$	110000
$\boldsymbol{B}[b]$	011000
$\boldsymbol{B}[c]$	101111
$\boldsymbol{B}[d]$	000111
$\boldsymbol{B}[e]$	000010
$\boldsymbol{B}[f]$	000011
$\boldsymbol{B}[g]$	000001

(a) $\boldsymbol{B}$

	$x_1 ... x_4$	x_5 x_6
$\boldsymbol{G}[x_1]$	0100	00
$\boldsymbol{G}[x_2]$	0010	00
$\boldsymbol{G}[x_3]$	1001	11
$\boldsymbol{G}[x_4]$	0000	11
$\boldsymbol{G}[x_5]$	0000	00
$\boldsymbol{G}[x_6]$	0000	10

(b) $\boldsymbol{G}$

Table 1: Bitwise representation of Figure 1(c).

Proposition 1. *(Correctness of Merging Nodes) After merging matched nodes, a reduced residual graph $R = \langle V_R, E_R \rangle$ can be transformed to a directed graph $G = \langle V, E \rangle$, where $V = X(c)$, $E = \{x \mapsto y \mid x \in X(c), y \in B_c(M(x)) \setminus \{x\}\}$.*

Proof. For reduced residual graph R of an alldifferent constraint c, the matching is a mapping from $X(c)$ to $D(c)$, denoted as f: $X(c) \mapsto D(c)$, and the unmatching is a mapping from $D(c)$ to $X(c)$, denoted as g: $D(c) \mapsto X(c)$. Trivially, f $\neq$ g. We combine f and g to eliminate V to get the composite mapping g $\circ$ f: $X(c) \mapsto X(c)$, which means $x \in X(c)$ gets its neighbor from variables whose domain contains the matching of x. Hence, it corresponds to the directed graph $G = \langle V, E \rangle$, $V = X(c)$, $E = \text{g} \circ \text{f} = \{x \mapsto y \mid x \in X(c), y \in B_c(M(x)) \setminus \{x\}\}$. □

Check the connectivity of edges. Our algorithm differs from Tarjan's algorithm in that it does not compute all SCCs at once. Instead, it is driven by the connectivity of edges in the reduced graph, and only checks variable values of an SCC when needed. As described in Definition 3, checking whether an unmatching edge $e_{a \mapsto x}$ is redundant is equivalent to checking whether $e_{M(a) \mapsto x}$ belongs to any alternating cycle. If there exists a path from x to $M(a)$, then the path $x \mapsto M(a)$ and edge $e_{M(a) \mapsto x}$ can form a cycle. For a valid variable value (x, a), its corresponding node $M(a)$ can be connected to node x. Checking whether it is in an SCC requires checking the connectivity from node x to node $M(a)$. Method *check*(x,a) in Algorithm 1 checks the connectivity

[1]This means that there is a variable node without a matching value node.

[2]All of the bitwise data structures are represented in bold font in this paper.

Algorithm 1: FIND SCCS

```
   Input: s: current SCC; DE: deleted edges in s
1  Method findSCCs(s, DE):
2      mergeMatchedNodes(s)
3      if ¬ matchingOpt(s) ∧ ¬ earlyDetect(s, DE) then
4          filterDomains(s)
5  Method mergeMatchedNodes(s):
6      Initial B s.t. B[a][x] = 1b iff x ∈ B_c(a)
7      foreach matched variable x ∈ s do
8          G[x] ← B[M(x)]
9          G[x][x] ← 0b
10         G_F[x] ← G[x]
11 Method matchingOpt(s):
12     Γ ← 0; A ← ∅
13     foreach free node a do
14         Γ ← Γ|B[a]; Γ_F ← Γ_F|B[a]; A[a] ← 1b
15     while x ∈ s s.t. Γ_F[x] = 1b do
16         Γ_F ← Γ_F | (B[M(x)]& ∼ Γ); Γ_F[x] ← 0b
17         Γ ← Γ | B[M(x)]; A[M(x)] ← 1b
18     Remove Type I redundant edges
19     Remove Γ from s; add Γ as an independent SCC
20     return |s| = |Γ|
21 Method earlyDetect(s,DE):
22     foreach (x, a) ∈ DE do
23         if ¬(check(x, a)∧check(M(a), M(x))) then
24             return false
25     return true
26 Method check(x, a):
27     while y ∈ s s.t. G_F[x][y] = 1b do
28         G_F[x] ← G_F[x] | (G[y]& ∼ G[x])
29         G_F[x][y] ← 0b
30         G[x] ← G[x] | G[y]
31         if G[x][M(a)] = 1b then
32             return true;
33     return false
34 Method filterDomains(s):
35     s_u ← s; s' ← s; s_c ← ∅; s_m ← ∅
36     while s' ≠ ∅ do
37         pop x' from s'
38         if s_c ≠ ∅ then
39             add s_c as a independent SCC
40         s_c = {x'}; s_u = (s_u ∪ s_m) \ {x'}; s_m = ∅
41         foreach x ∈ s_c do
42             foreach unmatched value a ∈ D(x) do
43                 if M(a) ∈ s_u then
44                     if check(x, a) then
45                         move M(a) to s' and s_c
46                     else
47                         remove a from D(x); move
                           M(a) to s_m
48                 else if M(a) ∈ s_m ∨ M(a) ∉ s then
49                     remove a from D(x);
```

of node x to node a. It uses the bit-BFS method proposed in [Cheriyan and Mehlhorn, 1996]. The method takes an extendable node y of $\boldsymbol{G}_F[x]$ (line 27). The connectable nodes of y that have not been visited by x (i.e., $\boldsymbol{G}[y]\& \sim \boldsymbol{G}[x]$) are added to the frontier $\boldsymbol{G}_F[x]$ by bitwise operations (line 28), and then $\boldsymbol{G}_F[x]$ clears the bit of extended node y (line 29). Next, all the nodes connected to x are recorded by bitwise anding $\boldsymbol{G}[y]$ to $\boldsymbol{G}[x]$ (line 30). The method continues traversing and returns true until $\boldsymbol{G}[x][M(a)] = 1b$ is detected, which means that x can connect to $M(a)$. If $\boldsymbol{G}[x][M(a)] = 0b$ when the BFS is completed, the method returns false. All search information is stored in $\boldsymbol{G}[x]$ and $\boldsymbol{G}_F[x]$ for future calls.

The aforementioned improvements are the core idea of Alldiff$^{\text{bit}}$. Additionally, our algorithm is highly scalable and can be easily extended and integrated to optimize all the related improvements for alldifferent constraints. We will describe these implementations in the following subsections.

Integration of SCC splitting. Since our algorithm does not directly compute SCCs, the mechanism of SCC Splitting in Gent algorithm needs to be slightly modified. Suppose s is the current variables partition that needs to compute SCC. We divide it into three subsets: connected, unknown and moved, denoted by s_c, s_u and s_m, respectively, where s_c denotes the set of nodes (i.e., variables) that are explicitly in the same SCC, s_m denotes the set of nodes that are explicitly not in current SCC with s_c, the rest of the nodes which need to be checked the connectivity are stored in s_u, and s' is a queue of variables to be extended during iteration in method *filterDomains*. This method initializes s_u and s' to s. It also resets both s_c and s_m to empty set, indicating that the connectivity of all nodes is initially unknown. The method then pushes them into queue s' to wait for checking if they are in SCC. The outer loop (line 36) iteratively checks the connectivity of each node in s'. The main loop (lines 41-49) gives priority to traversing the connected node in s_c. It moves newly connected nodes to s_c and unconnected ones to s_m. Since new connected nodes may be added to s_c, the main loop continues until all nodes of the current SCC have been fully explored. Once the traversal of the main loop is completed, s_c is transformed independently into a new SCC (line 39). This results in the previously traversed s_m becoming outdated and reclassified into s_u (line 40). Then, the outer loop pops a new node x' from s' to continue the traversal (line 37). Lines 43 to 49 are the core part for checking connectivity. For a variable-value pair (x, a), the method *check*(x,a) is invoked only if $M(a) \in s_u$. If $M(a)$ is in the moved sub-partition or does not belong to the current SCC s, it can be deleted directly (lines 48-49).

Integration of matching optimizing. Differing from DFS in Zhang18 algorithm, method *matchingOpt(s)* perform a traversal starting from free nodes in bit-BFS fashion. As $\boldsymbol{G}$ and $\boldsymbol{G}_F$, the method uses $\boldsymbol{\Gamma}$ to denoted the set Γ and $\boldsymbol{\Gamma}_F$ to denoted the set of nodes which needs to be traversed. The procedure for computing set Γ and A similarly to computing SCC as described above. It first iterates over each free node to initialize $\boldsymbol{\Gamma}$ and $\boldsymbol{\Gamma}_F$ by bitwise or, and adds the free nodes to $\boldsymbol{A}$ in lines 13 to 14. Then, it iteratively traverses

each node x from $\boldsymbol{\Gamma}_F$ and adds new untraversed nodes of x (i.e., $\boldsymbol{B}[M(x)]\& \sim \boldsymbol{\Gamma}$) to $\boldsymbol{\Gamma}_F$ for further traversal. Next, the method extends $\boldsymbol{\Gamma}$ and clears the newly traversed node x at statement $\boldsymbol{\Gamma}_F[x] \leftarrow 0b$. Please note that the same traversal method is used to extend $\boldsymbol{G}$ and $\boldsymbol{G}_F$ in method *check*(x, a). Finally, $M(x)$ is added to $\boldsymbol{A}$ at statement $\boldsymbol{A}[M(x)] \leftarrow 1b$ (lines 15-17). After bit-BFS, the type I redundant edges is directly delete in line 18. $\boldsymbol{\Gamma}$ is removed from current SCC s and added as a independent SCC. According to [Zhang *et al.*, 2018], if $\Gamma = s$, all variables in s do not need to be filtered, method *filterDomains* can be skipped directly.

Integration of early detection. Our algorithm uses value-driven SCC detection, which makes it more naturally compatible with early detection than the Zhang20 algorithm. As described in [Zhang *et al.*, 2020], the *earlyDetect* method checks whether the two paths from x to $M(a)$ and $M(a)$ to x exist (line 23) to detect whether a deleted edge (x, a) is an unimportant edge. If (x, a) is in the current SCC s, then $\neg(\text{check}(x, a) \wedge \text{check}(M(a), M(x)))$ returns true, indicating that (x, a) is an unimportant edge. If a deletion is detected to be important, the algorithm skips the detection and directly calls the *filterDomains*(s) method. Conversely, if all edges in DE pass the detection, the call of method *filterDomains* can be skipped.

In summary, the algorithm executes the *filterDomains* method only if the *matchingOpt* and *earlyDetect* methods pass, indicating that there are still variables to be filtered. Otherwise, the method is skipped (lines 3-4).

Proposition 2. *(Soundness) If (x, a) corresponds to a type II redundant edge on the reduced graph, it cannot pass method check(x, a).*

Proof. Suppose (x, a) corresponds to a type II redundant edges. So its corresponding edge is $e_{M(a)\mapsto x}$. If it can pass method *check*(x, a). i.e., we get $G[x][M(a)] = 1b$ after BFS at line 41. This means that there exists a path from x to $M(a)$, and this path and $e_{M(a)\mapsto x}$ make up a alternating cycle. $e_{M(a)\mapsto x}$ clearly does not satisfy Definition 3, i.e., (x, a) is not a type II redundant edge. This contradicts the hypothesis, so *check*(x, a) returns $false$ (i.e., $G[x][M(a)] = 0b$) if (x, a) is a type II redundant edge. □

Proposition 3. *(Completeness) All redundant edges can be removed in filterDomains().*

Proof. All unmatched nodes need to iteratively check whether they are redundant edges at lines 24-25 in Algorithm 1. We know from [Zhang *et al.*, 2018] that Γ and A are correctly partitioned after matching optimizing. Hence all type I redundant edges can be removed at line 14 in Algorithm 1. And all the type II redundant edges can be removed at line 37 and line 39 in Algorithm 1. Therefore, all redundant edges can be removed in *findSCC*(). □

Proposition 4. *(Correctness) Our filtering algorithm is correct on the reduced graph.*

Proof. Correctness immediately follows from Propositions 1, 2 and 3. □

Complexity. In this paper, we use a bit version FF-BFS [Ford and Fulkerson, 1956; Van Kessel and Quimper, 2012] to perform incremental matching in $O(km)$, where k and m are the numbers of unmatched and total edges in the variable–value graph, respectively. Although our lazy-computing SCCs mechanism does not change algorithm's complexity, it significantly reduces the size of the graph after node merging. The time complexity of computing SCCs via Régin's algorithm is $O(|V_R|+|E_R|) = O(r+2d+1+dr) = O(dr)$. That of Zhang18's algorithm is $O(|V_R| + |E_R|) = O(r + d + dr) = O(dr)$, and it can be reduced to $O(r^2)$ with matching optimizing. And that of our algorithm is $O(|V| + |E|) = O(r + r^2) = O(r^2)$. Apparently, for alldifferent constraint, we have $r \leq d$, so our algorithm has a lower time complexity than Régin's algorithm. Although our algorithm has the same complexity as Zhang18's algorithm, it requires less computational cost in practice due to the reduced digraph. In addition, using bit representation can further reduce memory consumption.

4 Experimental Results

We evaluated the performance of our algorithm by comparing it to backtracking search algorithm that embed arc consistency algorithms: Gent, WordRam, Zhang18 and Zhang20.[3] WordRam algorithm uses bitwise FF-BFS and bitwise Tarjan algorithm to compute maximum matching and SCCs, respectively. The default incremental matching algorithm for GAC algorithms are FF-BFS for non bit algorithms and bit FF-BFS for WordRam and our algorithm. We evaluate the performance of these embedded GAC algorithms by comparing search resolution algorithms. To conduct a comprehensive evaluation, we used various alldifferent CP instances from the XCSP3 website [Boussemart *et al.*, 2016][4]. All series names are listed in Table 2 and we tested over 800 instances. Our experiments were conducted on a PC with an AMD Ryzen 9 7950X CPU @ 4.5GHz, 32 GB RAM, and 64-bit Windows 11. The timeout was set to 1200 seconds for each instance. The number of non-timeout instances (and total instances) for each series is given below the series name after "#=". All the algorithms were implemented in the Java-based CP solver, Choco [Prud'homme *et al.*, 2017][5] using OpenJDK 19. To ensure fairness, We used binary branching search with DOM [Dechter and Meiri, 1994] as the variable ordering heuristic, and "min value" as the value ordering heuristic. This ensured that all algorithms generated the same shape of search trees when solving, avoiding interference with experimental results due to differences in their shapes. For a straightforward impression of the algorithms' efficacy, we first compared the solving time per series. While most CPs contain alldifferent constraints, they may also contain other types of constraints. To provide a microscopic observation,

[3]The source code and dataset are available at https://github.com/leezear2022/alldiff-choco. Our testing includes many academic and practical instances that require hashing of domains during propagation. However, this may decrease the efficiency of WordRAM and Alldiff^{bit}.

[4]http://xcsp.org/

[5]https://choco-solver.org/

		Gent	WordRam	Zhang18	Zhang20	Alldiff^{bit}
AllInterval	S	10.5646	9.4971	17.0013	19.5266	**5.1543**
#=22(22)	F	7.9763	4.0751	11.216	11.3596	**2.3148**
ColouredQueens	S	0.0281	0.023	0.0284	0.0285	0.0144
#=5(5)	F	0.0138	0.0095	0.0185	0.0148	**0.0037**
CostasArray	S	50.4423	54.3546	61.5451	61.9975	28.7757
#=9(9)	F	22.3688	18.2497	30.6049	26.0048	**4.1603**
CryptoPuzzle	S	0.0043	0.0024	0.0022	0.0024	**0.0020**
#=10(10)	F	0.0007	0.0005	0.0005	0.0006	**0.0003**
GolombRuler	S	44.9784	60.2597	42.3241	43.4504	**16.5973**
#=12(15)	F	28.6827	33.3267	19.091	21.3516	**2.0128**
GracefulGraph	S	129.2112	134.6306	162.6763	156.2065	103.2885
#=6(12)	F	30.5368	20.3289	57.2696	48.0534	**8.984**
Kakuro	S	5.4934	5.4074	5.9387	7.2000	3.4506
#=535(551)	F	2.2918	1.7664	2.7593	2.9282	**0.2988**
LatinSquare	S	8.7373	9.8025	13.378	9.4055	4.7786
#=215(248)	F	3.4382	3.793	7.9452	4.2036	**1.1698**
MagicSquare	S	2.5444	2.6016	3.9199	2.9171	1.7404
#=1(7)	F	1.0565	0.7561	2.333	1.2975	**0.3958**
NumberPartitioning	S	36.2312	35.9082	41.8748	36.0007	30.7444
#=25(27)	F	5.1889	4.1094	10.9782	5.3246	**2.5597**
OpenStacks	S	0.789	0.7997	0.8142	0.774	0.7515
#=74(77)	F	0.0204	0.0192	0.0479	0.0243	**0.0061**
Ortholatin	S	5.1972	5.3861	6.6632	5.814	2.8006
#=5(6)	F	2.5924	2.2569	4.1145	2.7694	**0.5722**
QuadraticAssignment	S	0.0728	0.0677	0.0714	0.0662	0.0602
#=23(120)	F	0.0042	0.003	0.0081	0.0044	**0.0011**
QuasiGroups	S	11.1594	11.1074	11.9626	11.6889	10.3804
#=6(6)	F	0.8937	0.6984	1.6201	1.1269	**0.255**
Scheduling	S	0.0174	0.0164	0.0164	0.0172	0.0158
#=22(41)	F	0.0001	0.0001	0.0002	0.0002	0.0002
SchurrLemma	S	28.4222	28.7055	39.8903	47.2946	18.4717
#=6(6)	F	13.8733	11.583	23.0209	21.6662	**2.9145**
SportsScheduling	S	80.0827	71.7606	74.1604	90.3811	53.6116
#=6(7)	F	21.1234	15.3616	19.9731	20.2245	**4.5165**
Sudoku	S	0.001	0.0006	0.0009	0.0008	0.0006
#=46(46)	F	0.0002	0.0002	0.0006	0.0003	0.0002

Table 2: Results of comparing filtering algorithms on series instances

we also compared the filtering time for solved instances. Additionally, we presented the proportion of different improvement points in the propagation process.

Table 2 presents the efficiency of the algorithm for each instance. Instances that timed out during testing are excluded from statistical averaging. The average CPU time (in seconds) for solving the problem and computing SCCs for the remaining instances are presented in the "S" and "F" rows, respectively. Gent serves as the baseline algorithm in Table 2. The results of Alldiff^{bit} with a 50% to 100% increase in efficiency are underlined, while those above 100% are bolded. We first compare the average solving time of the instances. Our algorithm solves the instances in the shortest time on all 18 groups of instances. This is because our algorithm integrates and improves all the improvement points. In particular, our algorithm achieves a lead of 50% to 100% on 7 series and even more than 100% on 3 series. In short, our algorithm's solving efficiency is improved by an average of 60.65% across all series. It is worth noting that our algorithm also achieves the best results in instances where Zhang18 or Zhang20 algorithm are better, such as CryptoPuzzle and GolombRuler for Zhang18 algorithm, and Kakuro, Openstacks, and QuadraticAssignment for Zhang20 algorithm. This is due to our algorithm's integration and improvement of these enhancements. We continue to compare the time it takes for each algorithm to compute SCCs. To better analyze the improvements, we are blocking out other distracting factors, such as computation of matching, propagation of other types of constraints, search algorithms, etc. Similar to the trend of solving time, our algorithm is the most efficient in calculating SCCs. Moreover, it achieves a higher acceleration ratio compared to the other algorithms. In fact, our algorithm won and achieved more than 100% acceleration in 16 series of instances, and even more than 5 times acceleration in 2

series.

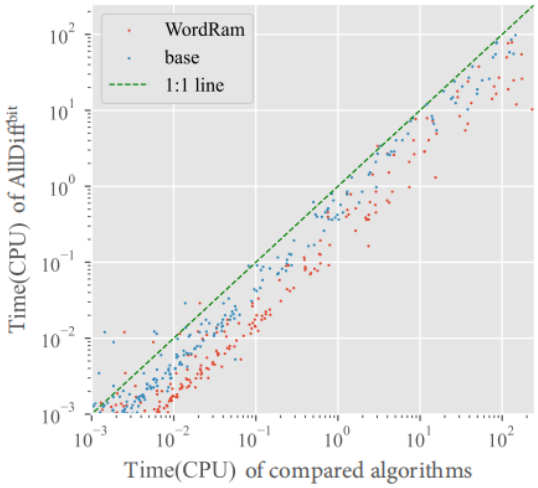

Figure 2: Runtime comparison of method *findSCCs* for bitwise GAC algorithms.

To compare the performance of the core idea presented in this paper with our integration algorithm – Alldiff^{bit} – we implemented a simple algorithm "base" with bitwise representation and "Check the Connectivity of Edges", which we provisionally refer to as "base." Figure 2 shows a scatter plot that allows us to compare the efficiency of three bitwise GAC algorithms when executing the method *findSCC* for each instance. The vertical and horizontal axes represent the CPU time (in seconds) used by Alldiff^{bit} and other bitwise GAC algorithms to compute SCCs, respectively. In general, as the time increases, most of the dots are located at the bottom right of the diagonal. This indicates that Alldiff^{bit} is more efficient for easy to hard instances. It is worth noting that the main difference between the two bit-based algorithms, WordRam and Alldiff^{bit}, is the traversal method.

The Wordram algorithm uses bit-DFS, which requires maintaining a bit vector during traversal to record the nodes to be extended at each propagation. Due to the need for complex data structures to enumerate SCCs, they are difficult to vectorize, which increases maintenance costs. In contrast, our algorithm only retains $\boldsymbol{G}$ and $\boldsymbol{G}_F$ for bit-BFS since no prior enumeration of SCCs is required. As a result, their maintenance is lightweight, which significantly improves the efficiency of the overall method execution. As Alldiff^{bit} is an integrated algorithm, we observe that the dots of the base algorithm cluster on an "oblique line" below the diagonal. This is because Alldiff^{bit} is a direct improvement of the base algorithm, implying that our algorithm has a more stable improvement. Nevertheless, the base algorithm is more efficient than the WordRam algorithm. In summary, the solution efficiency of Alldiff^{bit} improves by an average of 60.65% per group across all series.

Finally, in order to observe the impact of the various improvement points, we counted the proportion of their occurrence in computing SCCs (referred to the propagation type) during the solution process of the solved instances. If the type

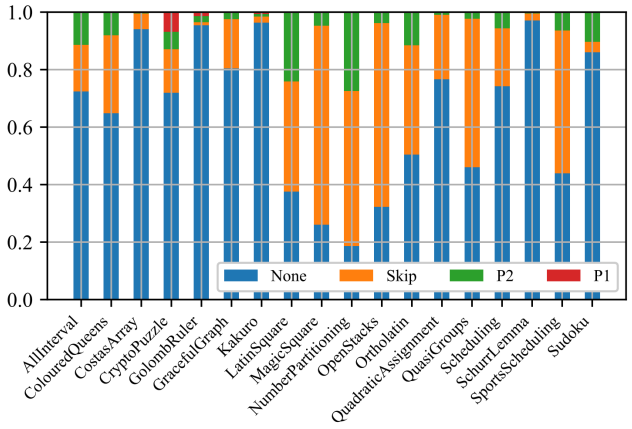

Figure 3: The proportion of propagation types.

I (resp. II) redundant edges are deleted, we call it the type I (resp. II) deletion, and denote it as "P1" (resp. "P2"). And if it is skipped due to early detection, it is denoted as "Skip," otherwise, it is denoted as "None." As shown in Figure 3, for series CryptoPuzzle, GolumbRuler, and Kakuro, the type I deletion occupies a particular proportion. This is consistent with the results in Table 2, which show that the Zhang18 has better performance on these series compared to the baseline algorithm. While for other series, the type II deletion still accounts for a large proportion. This is because if there are no free nodes, no match optimization is performed on line 13 of algorithm 1. For other series, however, the type II deletion still accounts for a large proportion. This is because if there are no free nodes, no match optimization is performed on line 13 of algorithm 1. The early detection identifies a large amount of redundant propagation. While this may incur additional overhead in some degree of presentation, it substantially reduces the number of computed SCCs on many series instances. Therefore, considering the overhead, Zhang20 performs better on some series with a higher proportion of partial early detection than the baseline algorithm.

5 Conclusion

In this paper, we present Alldiff^{bit}, an improved GAC filtering algorithm for alldifferent constraints. Our main idea is based on the observation that it is unnecessary to enumerate strongly connected components (SCCs) in order to determine whether a variable-value satisfies GAC. Instead, only the edges of the graph model need to be checked for strong connectivity. This allows the data structures used by graph traversal to be simplified as fully bit vectorized, thus substantially reducing the overhead of graph traversal. Additionally, Alldiff^{bit} integrates the SCC splitting of Gent, the matching optimization of Zhang18, and the early detection method of Zhang20 algorithm, making it strongly extensible. Our algorithm further improves upon these algorithms. Extensive experiments show that Alldiff^{bit} significantly outperforms existing GAC algorithms. All of these results illustrate that Alldiff^{bit} completes a staged integration and optimization of the GAC algorithm for alldifferent constraints.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work is supported by the NSF of Hunan Province No.2022JJ10066, the National Nature Science Foundation of China No.62272477 and 62276060.

References

[Berge, 1973] Claude Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[Bessiere *et al.*, 2009] Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decompositions of all different, global cardinality and related constraints. In *International Joint Conference on Artificial Intelligence*, pages 419–424, 2009.

[Boussemart *et al.*, 2016] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

[Cheriyan and Mehlhorn, 1996] Joseph Cheriyan and Kurt Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[Dechter and Meiri, 1994] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68(2):211–241, 1994.

[Demeulenaere *et al.*, 2016] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*, pages 207–223. Springer, 2016.

[Ford and Fulkerson, 1956] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[Gent *et al.*, 2008] 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.

[Lauriere, 1978] Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, 10(1):29–127, 1978.

[Leconte, 1996] Michel Leconte. A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning*, pages 19–28, 1996.

[Lecoutre and Vion, 2008] Christophe Lecoutre and Julien Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters (CPL)*, 2:21–35, 2008.

[Li *et al.*, 2021] Zhe Li, Zhezhou Yu, Hongbo Li, Jinsong Guo, and Zhanshan Li. Revisiting the efficacy of weak consistencies: a study of forward checking. *Science China Information Sciences*, 64:1–3, 2021.

[López-Ortiz *et al.*, 2003] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter Van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *International Joint Conference on Artificial Intelligence*, pages 245–250, 2003.

[Prud'homme *et al.*, 2017] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[Régin, 1994] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 94, pages 362–367, 1994.

[Tarjan, 1972] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[Van Kessel and Quimper, 2012] Philippe Van Kessel and Claude-Guy Quimper. Filtering algorithms based on the word-ram model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, 2012.

[Wang *et al.*, 2016] Ruiwei Wang, Wei Xia, Roland HC Yap, and Zhanshan Li. Optimizing simple tabular reduction with a bitwise representation. In *International Joint Conference on Artificial Intelligence*, pages 787–795, 2016.

[Zhang *et al.*, 2017] Xizhe Zhang, Jianfei Han, and Weixiong Zhang. An efficient algorithm for finding all possible input nodes for controlling complex networks. *Scientific Reports*, 7(1):10677, 2017.

[Zhang *et al.*, 2018] Xizhe Zhang, Qian Li, and Weixiong Zhang. A fast algorithm for generalized arc consistency of the alldifferent constraint. In *International Joint Conference on Artificial Intelligence*, pages 1398–1403, 2018.

[Zhang *et al.*, 2020] Xizhe Zhang, Jian Gao, Yizhi Lv, and Weixiong Zhang. Early and efficient identification of useless constraint propagation for alldifferent constraints. In *International Joint Conference on Artificial Intelligence*, pages 1126–1133, 2020.