Bimodal Depth-First Search for Scalable GAC for AllDifferent

Sulian Le Bozec-Chiffoleau[1], Nicolas Beldiceanu[1], Charles Prud'homme[1],
Gilles Simonin[1], Xavier Lorca[2]

[1] IMT Atlantique, LS2N, UMR CNRS 6004, F-44307 Nantes, France
sulian.le-bozec-chiffoleau@imt-atlantique.fr
[2] Centre Génie Industriel, IMT Mines Albi, Université de Toulouse, Albi

Keywords : *depth-first search, constraint programming, global constraint, alldifferent, partially-complemented graph*

1 Introduction

Constraint Programming (CP) [9] models problems using decision variables with domains and constraints. A CP solver searches for assignments satisfying all constraints, combining *search* – systematically exploring and backtracking when conflicts arise – and *propagation*, which prunes inconsistent values to reduce the search space. Because complete pruning is NP-hard, CP solvers balance propagation and search. *Global constraints* capture interactions among multiple variables and enable stronger filtering. For instance, the ALLDIFFERENT constraint [8] enforces pairwise distinct values. Such constraints exploit problem structure to achieve high levels of consistency, such as *Generalised Arc Consistency* (GAC), which removes all inconsistent values, or *Bound Consistency* (BC), which relaxes domains to intervals, enabling faster filtering but less pruning. Filtering global constraints often relies on graph algorithms, particularly *Depth-First Search* (DFS) on the *Variable–Value Graph* (VVG), where variables and values form a bipartite graph. However, DFS becomes costly on dense graphs, as it requires $\Theta(n+m)$ time for n vertices and m edges. To accelerate DFS, [2] introduced the *partially complemented* (p-c) graph representation, which stores for each vertex either its successors or its non-successors, whichever is smaller. Its size is $n + \tilde{m}$, with $\tilde{m}$ summing these minima over all vertices; algorithms can then run in $\tilde{m}$-based times rather than m-based times. Since VVGs may be initially dense, later sparse, or unevenly structured, $\tilde{m}$-based time complexities are attractive. However, CP domain representations prevent direct use of p-c techniques on VVGs. This motivates our *bimodal DFS*, inspired by p-c DFS but adapted to CP internal data structures. It exploits two common solver capabilities: (A1) efficient iteration over a variable's domain (successors in the VVG) and (A2) efficient membership tests for values (edge existence).

2 Background and Motivations

2.1 The ALLDIFFERENT constraint

The ALLDIFFERENT constraint is a key constraint in Constraint Programming, used in many applications. It ensures all variables take a different value:

$$\text{ALLDIFFERENT}(x_0, \ldots, x_{n-1}) \equiv x_i \neq x_j, \text{ for all } (i,j) \in \{0, \ldots, n-1\}^2 \text{ such that } i \neq j$$

An efficient GAC algorithm removing all inconsistent values with respect to the constraint was proposed by [8] (Figure 1). First, it finds a maximum matching in the VVG, a bipartite graph where one set of vertices represents variables and the other set corresponds to possible values. Edges link variables to their feasible values. Then, it constructs the *Residual Graph* (RG), a directed version of

This is a summary of [5], presented at the 34th International Joint Conference on Artificial Intelligence (IJCAI 2025)

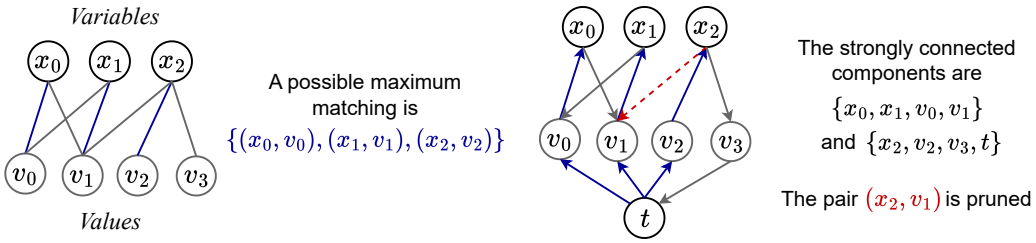

FIG. 1: Illustration of Régin's algorithm, with the variable-value graph (left) and the residual graph (right)

the VVG and the matching. Arcs are directed from variables to values, except for the pairs in the matching for which arcs are directed from values to variables. It also contains an artificial vertex t, with arcs from t to all matched values, and arcs from all unmatched values to t. Finally, variable-value pairs that do not belong to the same *strongly connected component* (SCC) in the RG are pruned.

Problem: The **practical bottleneck** is the computation of SCCs by Tarjan's algorithm [10], which is based on DFS and runs in $\Theta(n+m)$ time for a graph with n vertices and m arcs.

2.2 Partially-Complemented Representation of Graphs

A number of studies in the literature have developped techniques to reduce the time complexity for constructing a BFS-forest or a DFS-forest of a graph. Indeed, the size of a forest is bounded by the number of vertices n, so one would ideally like to construct it in $O(n)$ time.

In particular, [2] introduced the *Partially Complemented Representation* (p-c representation) of a directed graph $G = (V, A)$. It consists, for each vertex $v \in V$, in either storing its *successors* $N^+(v) = \{w \in V : (v, w) \in A\}$ or its *non-successors* $\overline{N}^+(v) = \{w \in V : (v, w) \notin A\}$ (v is then said to be *complemented*). We can obtain a minimum-sized p-c representation $\tilde{G}$ of G in $O(n+m)$ time, where $n = |V|$ and $m = |A|$. Indeed, with $d^+(v) = |N^+(v)|$, for each $v \in V$: if $d^+(v) < n - d^+(v)$ then $N^+(v)$ is stored, otherwise $\overline{N}^+(v)$ is stored. Then, with $\tilde{m} = \sum_{v \in V} \min(d^+(v), n - d^+(v))$, the size of the minimum-sized p-c representation $\tilde{G}$ is $n + \tilde{m}$. See Figure 2.

Some graph algorithms, including BFS and DFS, can be adapted to the p-c representation. This results in time complexities that depend on $\tilde{m}$ instead of m. This is a very interesting property when $\tilde{m} \ll m$, which is the case, for example, when the graph is very dense.

Question: Can we use such p-c algorithms within constraint programming solvers ?

2.3 Graph Representations in Constraint Programming

In constraint programming, graphs are often derived from the domain of integer variables, notably when studying global constraints. For time and space-saving purposes, the graph G is not explicitly constructed but inferred from the domains. Hence, graph operations over G are applied through domain operations: $N^+(x) = D(x)$ and $(x, v) \in G \Leftrightarrow v \in D(x)$. Where $N^+(x)$ denotes the successors of vertex x, $D(x)$ is the domain of variable x, and v is a value of the universe.

To answer the question asked in the previous section, we must look at the different integer domain representations that are used in constraint solvers and check whether they allow efficient iteration

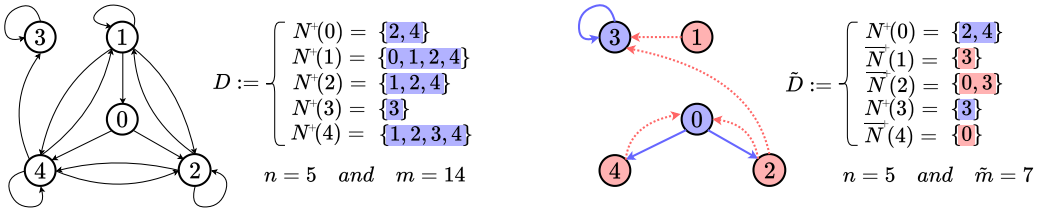

FIG. 2: Illustration of the adjacency list (left) and minimum-sized p-c (right) representations of the same graph

over the domain and over the non-domain. Three predominant data structures are *sparse set* [3], *successor vector* (doubly-linked list) [11] and *bitset*. Among the three, only sparse set enables an efficient iteration over the non-domain $\overline{D}(x)$ of a variable x. Whereas all three allow a constant time check for the presence of a given value v. Furthermore, the non-domain of a variable does not necessarily represent the whole set of non-successors in the corresponding VVG, as all variables may not have the same initial universe of values.

For these reasons, we can not directly apply the p-c algorithms to graphs derived from the domain of integer variables. However, we can reasonably make the following assumptions about such graphs:

- $(A1)$ Iterating over the successors $N^+(v)$ of a vertex v is done in $O(d^+(v))$ time
- $(A2)$ Checking the existence of an arc (u, v) in the graph is done in $O(1)$ time

Indeed, these assumptions hold for both sparse set and successor vector, while bitset does not comply with assumption $(A1)$ only.

Objective: Obtaining $\tilde{m}$-based time complexities under assumptions $(A1)$ and $(A2)$.

3 The Bimodal Graph Traversal Algorithms

Let us consider a directed graph $G = (V, A)$. In this section, we describe our bimodal graph traversal algorithms that enable to construct a BFS-forest and a DFS-forest of G in $O(n + \tilde{m})$ time under assumptions $(A1)$ and $(A2)$. We will also assume the following:

- $(A3)$ Accessing the degree $d^+(v)$ of a vertex v is done in $O(1)$ time

This assumption is not necessary to get the targetted complexity but allows for a simpler design of the algorithms. Note that $(A3)$ is satisfied by domain representations as they dynamically update the size of the domain.

3.1 Bimodal Breadth-First Search

To perform a bimodal BFS, we use a doubly linked list to represent the set of unvisited vertices. This allows us to iterate over the set of unvisited vertices in linear time in their number, and to check if a vertex is already visited in constant time because an element e from the initial universe is present in the list if and only if $\texttt{prev}(\texttt{next}(e)) = e$. In other words, it satisfies the complexities mentioned in assumptions $(A1)$ and $(A2)$.

The bimodal BFS is fairly simple and works as follows. When exploring from a given vertex v, one want to add all unvisited successors of v to the queue of vertices to explore from, and mark them as visited. The point is then to find the intersection between $N^+(v)$, the set of successors of v, and U, the set of unvisited vertices. Given that we can iterate on both sets in linear time in their size, and check the presence of a vertex in constant time, we can therefore find their intersection in $O(min(d^+(v), |U|))$ time. At this point of the exploration, the vertices in U are either successors of v in G and then successors of v in the resulting BFS-forest, or non-successors of v whose number is bounded by $n - d^+(v)$. Because the size of a BFS-forest is bounded by $n - 1$, this leads to an $O(n + \tilde{m})$ time complexity for constructing a BFS-forest of G.

3.2 Bimodal Depth-First Search

Difficulties arise when considering DFS because of the backtracking behaviour and the fact that each vertex may be visited several times. When exploring from a vertex v, one can choose between two modes: iterate over the successors $N^+(v)$ or the unvisited vertices U, depending on the degree $d^+(v)$. That is where the name *bimodal* comes from. If one choose to iterate over $N^+(v)$, it corresponds to the existing DFS and leads to $O(d^+(v))$ time spent for exploring from vertex v. If one choose to iterate over U, finding the intersection between $N^+(v)$ and U at each visit of v would be too costly, as only one unvisited successor is required. So, the idea is to iterate over U until we find a successor

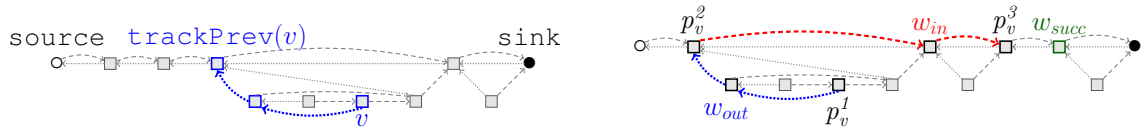

FIG. 3: Left: Illustration of the `trackPrev` function. The nodes at the top are still in the list, while those at the bottom were removed. The thick-blue nodes are the positions successively taken by *track* when calling `trackPrev`(v). Right: Illustration of the sketch of the proof of Theorem 1.

Algorithm 1 Bimodal Depth-First Search (Bi-DFS)

Require: A directed graph $G = (V, A)$
Ensure: Explore a DFS-forest of G

```
1:  U ← V;                                  ▷ Initialise a tracking list of unvisited vertices U with all the vertices V
2:  while U ≠ ∅ do EXPLORE(U.next(U.source));
3:  procedure EXPLORE(v)
4:      U.remove(v);
5:      if d⁺(v) < n − d⁺(v) then                          ▷ iterate over the successors N⁺(v)
6:          for w ∈ N⁺(v) : U.present(w) do EXPLORE(w);
7:      else                                               ▷ iterate over the unvisited vertices U
8:          p_v ← U.source;
9:          while U.hasNext(p_v) do
                ①
10:             p_v ← U.trackPrev(p_v);
                ②
11:             while U.hasNext(p_v) ∧ U.next(p_v) ∉ N⁺(v) do
12:                 p_v ← U.next(p_v);
                ③
13:             if U.hasNext(p_v) then EXPLORE(U.next(p_v));
```

w of v. We store with a pointer p_v our last position in the list before finding w, so that during the next visit of v we can start iterating from this position. The vertex w is then removed from U and the exploration continues from w.

The problem is, the list of unvisited vertices does change between two visits of v, and the position taken by p_v may be removed from U during this period. We would therefore have no choice but to start iterating over U from scratch during the next visit of v, which would result in a time complexity exceeding $O(n + \tilde{m})$.

To address this issue, we introduce the *tracking list*, a doubly linked list with an extra function `trackPrev`. When a node is removed, we keep the information of its `prev` and `next` pointers. The `source` and `sink` nodes are artificial nodes added at the beginning and at the end of the list.

Definition 1 (Tracking List)**.** *A* tracking list *is a doubly linked list to which we add the following elementary function (see Figure 3-left):*

function TRACKPREV(v) ▷ *returns the first present node by following the* `prev` *pointers from* v*:*
  $track \leftarrow v$;
  while ¬`present`$(track)$ **do** $track \leftarrow$ `prev`$(track)$;
  return $track$;

Algorithm 1 describes our bimodal Depth-First Search, Bi-DFS.

Theorem 1. *Bi-DFS goes through a DFS-forest of G, and, under assumptions $(A1) - (A3)$, the time spent exploring from a vertex $v \in V$ is:*

- $O(d^+(v))$ *if* $d^+(v) < n - d^+(v)$;
- $O(n - d^+(v) + d^+_{DFS}(v))$ *if* $d^+(v) \geq n - d^+(v)$.

Where $d^+_{DFS}(v)$ denotes the number of successors of v in the resulting DFS-forest.

Strategy	Condition to iterate over U	Comments
CLASSIC	Never	Does Régin's algorithm with the classical BFS and DFS
COMP	Always	Leads to time complexities based on the complement graph's size
PARTIAL	$\|U\| < \|D(x)\|$	Leads to the $\tilde{m}$-based time complexities
TUNED	$\sqrt{\|U\|} < \|D(x)\|$	Optimised for better practical performances

TAB. 1: Four different strategies for choosing the exploration mode in the bimodal algorithms

Proof : The proof outline is provided here. The main difficulty lies in proving that p_v does not traverse too many nodes in U over all visits of v. Consider an iteration of the loop at line 9, which represents a single visit to vertex v. In Bi-DFS we distinguish three positions for p_v: (*i*) just before line 10 at ①, denoted p_v^1; (*ii*) just before lines 11–12 at ②, denoted p_v^2; (*iii*) just after lines 11–12 at ③, denoted p_v^3. We can prove that (*a*) every node traversed by p_v is a non-successor of v, and that (*b*) every node traversed by p_v is traversed at most once between p_v^1 and p_v^2 (e.g. w_{out} in Fig. 3-right), and at most once between p_v^2 and p_v^3 (e.g. w_{in} in Fig. 3-right) over all visits of v.

Corollary 1. *Under assumptions* $(A1) - (A3)$*, a DFS-forest for a directed graph* G *can be constructed in* $O(n + \tilde{m})$ *time.*

Proof : By Theorem 1, definition of $\tilde{m}$, and because $\sum_{v \in V} d^+_{DFS}(v) \leq n$.

4 Experiments on ALLDIFFERENT using Choco-solver

The ALLDIFFERENT constraint being our original motivation, we extend our bimodal approach to the whole filtering procedure described by Régin's algorithm. To compute the maximum matching in the VVG, we use Kuhn's algorithm [4] that finds augmenting paths with BFS. We replace each BFS by our bimodal BFS. To compute the SCCs, we extend our complexity result for bimodal DFS to Tarjan's algorithm.

Our bimodal DFS (Bi-DFS) is configurable in the sense that one can change the way to choose the exploration mode in order to get better practical performances. In the experiments, we considered the four strategies detailed in Table 1 for choosing the exploration mode. We compared with three built-in algorithms in CHOCO-SOLVER [7]: REGIN, ZHANG (a recent optimisation of Régin's algorithm [12]) and BC (a bound consistency propagator for ALLDIFFERENT that filters less but faster [6]).

Among others, we considered two problems mainly carried by ALLDIFFERENT for which we can gradually increase the size: N-Queens and Latin-Squares. We used `Choco-solver` to solve the instances of the two problems with the mentioned algorithms for ALLDIFFERENT (Figure 4).

Our experiments show that the bimodal approach, particularly when paired with the TUNED or COMP strategies, significantly improves performance when maintaining GAC for ALLDIFFERENT constraints with large sizes. The speedup even increases with constraint size. Finally, the filtering

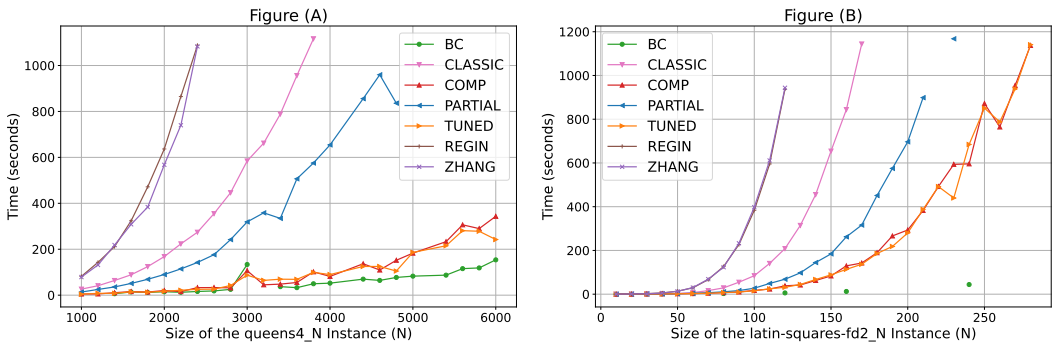

FIG. 4: Time to find the first solution on N-Queens (A) and Latin-Squares (B) instances

speed of our GAC algorithms is comparable to that of a BC filtering algorithm, but enables to solve more instances. Indeed, BC did not solve sizes 1200 and 3200 for N-Queens nor many sizes for Latin-Squares due to the lack of filtering strength.

5 Discussion and Perspectives

We proposed the bimodal graph traversal algorithms that achieve the same time complexities than the partially-complemented algorithms, but on graphs derived from integer domain variables in constraint programming. We integrated our approach into `Choco-solver` and experimentally showed its significant benefits. In conclusion, we believe that GAC could now be chosen over BC to filter ALLDIFFERENT as a default behaviour in CP solvers.

The bimodal approach can be extended to other global constraints. In particular, those based on Dumalge-Mendelsohn decomposition [1] or constraints over graph variables. It could also be useful outside CP, as long as assumptions $(A1)$ and $(A2)$ hold.

References

[1] Radoslaw Cymer. Dulmage-mendelsohn canonical decomposition as a generic pruning technique. *Constraints An Int. J.*, 17(3):234–272, 2012.

[2] Elias Dahlhaus, Jens Gustedt, and Ross M. McConnell. Partially complemented representations of digraphs. *Discrete Mathematics & Theoretical Computer Science*, 5, 2002.

[3] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

[4] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 52(1):7–21, 2004.

[5] Sulian Le Bozec-Chiffoleau, Nicolas Beldiceanu, Charles Prud'Homme, Gilles Simonin, and Xavier Lorca. Bimodal depth-first search for scalable gac for alldifferent. In *IJCAI*, pages 2610–2618, 2025.

[6] 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 *IJCAI*, volume 3, pages 245–250, 2003.

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

[8] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367, 1994.

[9] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[10] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[11] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial intelligence*, 57(2-3):291–321, 1992.

[12] Xizhe Zhang, Qian Li, and Weixiong Zhang. A fast algorithm for generalized arc consistency of the alldifferent constraint. In *IJCAI*, pages 1398–1403, 2018.