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

Extending SMT solvers with support for finite domain `alldifferent` constraint[⋆]

Milan Banković

the date of receipt and acceptance should be inserted later

Abstract In this paper we consider integration of SMT solvers with the filtering algorithms for the finite domain `alldifferent` constraint. Such integration makes SMT solvers suitable for solving constraint satisfaction problems with the `alldifferent` constraint involved. First, we present a novel algorithm for explaining inconsistencies and propagations in the `alldifferent` constraint. We compare it to Katsirelos' algorithm and flow-based algorithms that are commonly used for that purpose. Then we describe our DPLL(**T**)-compliant SMT theory solver for constraint satisfaction problems that include `alldifferent` constraints. We also provide an experimental evaluation of our approach.

Keywords SMT solving · CSP solving · `alldifferent` constraint · explanation algorithms

1 Introduction

Satisfiability modulo theories (SMT) ([2]) is an intensively studied area of research in recent years. It extends *boolean satisfiability* (SAT) technologies to support checking for satisfiability of first order formulae with respect to some first order theory. In this way, the powerful general purpose techniques used in SAT (clause learning, backjumping, efficient search space exploration, etc.) are combined with specific decision procedures that enable better reasoning about some theories of interest. The theories that usually appear in SMT are mainly chosen for their applicability in industry — predominately in *software verification* (these include arithmetics, bitvectors, theories of arrays and lists, theory of uninterpreted functions, etc.). Defining new theories

[⋆] The final publication is available at Springer via http://dx.doi.org/10.1007/s10601-015-9232-8

Milan Banković
Faculty of Mathematics, University of Belgrade
Studentski Trg 16, 11000 Belgrade, Serbia
E-mail: milan@matf.bg.ac.rs

and incorporating appropriate decision procedures into SMT may extend applicability of SMT solvers to other areas. One such possibility arises from combining SMT with techniques and algorithms developed within *constraint programming* (CP) ([33]). Equipped with CP algorithms, SMT solvers would be able to solve *constraint satisfaction problems* (CSP) more efficiently than they can do now. The possibility of such integration has been already suggested in [26] and [27].

With these facts in mind, in this paper we consider extending an SMT solver with support for the finite domain `alldifferent` constraint which constrains its variables to take pairwise distinct values (from their finite domains). This is one of the most famous global constraints, and its applications vary from puzzle solving, scheduling, timetabling, etc. Because of its applicability, it has been studied intensively over the last few decades, and several different filtering algorithms have been developed. The most famous among them is certainly *Regin's filtering algorithm* ([30]) that is based on the matching problem in bipartite graphs and that enforces *hyper-arc consistency*. The algorithm can be easily incorporated into SMT solvers, provided that it is extended with *explanation* capabilities: it must be able to explain *inconsistencies* (*conflicts* in SMT terminology) as well as *prunings* and *assignments* (*theory propagations* in SMT terminology), which is needed for *conflict analysis*. Since many modern constraint solvers also support learning and non-chronological backtracking, the need for explanations also exists in CP world, so the first such algorithms were developed within constraint solvers. The most notable explaining algorithms are the one described by Katsirelos ([21]), and those based on the minimal cuts in the flow networks ([32], [9]). In this paper we consider an alternative explaining algorithm, based on *minimal obstacle set* (MOS), that was preliminarily described in [1]. This paper is the extension of [1], and its main contributions are the following:

– we describe our MOS-based algorithm for explaining inconsistencies and propagations for the `alldifferent` constraint in more details. We also provide a comparison with Katsirelos' and flow-based algorithms, giving an insight of why our algorithm might be better (Section 3).
– we explain how the integration of the `alldifferent` algorithms and SMT solver is achieved (i.e. how these algorithms are adapted to work within SMT environment). We describe the implementation of our SMT solver for solving CSPs and provide an usage example (Section 4).
– we present experimental results in order to evaluate our approach (Section 5).

Related work. A survey on `alldifferent` constraint can be found in [18]. It presents filtering algorithms that enforce different types of local consistency (including already mentioned Regin's algorithm [30]). When explanation algorithms are concerned, the reference work is the work of Katsirelos ([21]). It is further extended in [25], where the same algorithm is presented, but the explanations are now generated lazily, when needed. An approach to explaining based on the flow networks ([11]) is considered in [32] and [9]. The explanations obtained in [32] are similar to ours, but may include redundant literals that our algorithm efficiently eliminates. The explanations obtained in [9] are the same as Katsirelos', except that the proposed algorithm can also be used for explaining propagated assignments, while Katsirelos' algorithm explains only prunings (our algorithm also explains both assignments and prunings in a uniform fashion). In [8], the same authors give a more detailed discussion on explaining `alldifferent`. In this work, both bound and hyper-arc consistency of the `alldifferent` constraint are considered and the explaining algorithms are proposed in both cases. In the case of hyper-arc consistency, the proposed algorithm is the same as in [9], so it is equivalent to Katsirelos' work, with additional assignments explaining capability. Just like our explaining algorithm, the algorithm proposed in [9] and [8] can generate explanations lazily.

The use of SAT and SMT technologies in solving CSPs is also a vivid research area. There is a lot of work on encoding CSP problems into SAT ([40], [29], [38], [19]). These approaches are eager, i.e. the problem is completely encoded into SAT and fed to the SAT solver. The SMT approach is also used in solving CSPs ([5], [20]), where the CSP problems are encoded in SMT-LIB ([3]) language, but only the standard SMT theories are used in the encoding. To the author's best knowledge, there is no published work on developing specific SMT theories for global constraints and incorporating filtering algorithms for such constraints into SMT (although we are aware of some work in progress on this topic ([27])).

One interesting approach in using SAT technologies in solving CSPs is so-called *lazy clause generation* (LCG) ([28]). In this approach, the SAT solver is integrated with propagators for global constraints. When a propagator wants to make a propagation, it generates a clause that represents the inference. The clause is then sent to the SAT solver, which unit-propagates the corresponding literal. The approach is quite similar to our approach, since it combines SAT technologies with filtering algorithms for global constraints. The main difference is in the propagation mechanism: we do not generate clauses that trigger unit propagations, our propagators work as SMT theory solvers that propagate inferred literals themselves, and later provide the explanations when needed. In recent years, LCG solvers also converged to explaining propagations lazily, as suggested in [39]. In particular, this is the case with `alldifferent` explaining, as stated in [8]. The SAT literals are also introduced lazily, just like in our solver. However, up to our knowledge, the details still remain unpublished.

2 Background

In this paper we assume the standard syntax and semantics of first order logic *with equality*, adopting the terminology used in [2]. We also assume that all considered first order formulae are *ground*, i.e. do not contain variables. A formula F over some fixed signature Σ is *satisfiable* if there exists an interpretation M of the function and predicate symbols of Σ such that formula F evaluates to *true* in M (denoted by $M \vDash F$). In practice, we are usually restricted to some particular set $\mathbf{T}$ of interpretations over Σ, called a (first order) *theory*. A formula F is *satisfiable* in $\mathbf{T}$ (or $\mathbf{T}$-*satisfiable*) if exists $M \in \mathbf{T}$ such that $M \vDash F$. The problem of checking $\mathbf{T}$-satisfiability of a first order formula is called *Satisfiability Modulo Theory (SMT) problem*. As a special case, a *propositional formula* is a ground formula containing only predicate symbols of arity 0, called *propositional symbols*, that are interpreted either as *true* or *false*. Problem of checking satisfiability of a propositional formula is known as *SAT problem*.

Procedures for solving SAT and SMT problems are called, respectively, SAT and SMT *solvers*. The most successful modern SAT solvers are CDCL solvers (short for *conflict-driven clause learning* ([24])), based on famous DPLL algorithm ([7]), but with many improvements (both algorithmic and implementational). On the other hand, modern SMT solvers are usually built on top of existing CDCL SAT solvers, following the so-called *lazy approach* ([2]) (the most famous such architecture is known as DPLL(**T**) ([12])). In essence, this means that an SMT solver consists of a SAT engine and decision procedures for supported theories (called *theory solvers*). A theory solver for **T** should be able to check the satisfiability of conjunctions of literals over **T** and to provide an *explanation* of unsatisfiability. Beside this, theory solvers may also have other desirable properties such as *incrementality* and detection (and explaining) of *theory propagations*. The SAT engine searches for a satisfying truth assignment to literals of the formula in the usual fashion, but consulting the theory solver when necessary in order to keep the assignment consistent with the theory.

In this paper we also rely on terminology and concepts introduced in [18]. We only repeat the most important notions here. A Constraint Satisfaction Problem (CSP) is represented by the triplet $(\mathbf{X},\mathbf{D},\mathbf{C})$, where $\mathbf{X} = (x_1,x_2,\ldots,x_n)$ is a finite set of variables, $\mathbf{D} = (D_{x_1},D_{x_2},\ldots,D_{x_n})$ is a set of finite domains, where D_{x_i} is the domain of the variable x_i, $\mathbf{C} = \{C_1,C_2,\ldots,C_m\}$ is a finite set of constraints. A constraint $C \in \mathbf{C}$ over variables $x_{i_1},x_{i_2},\ldots,x_{i_k}$ is some subset of $D_{x_{i_1}} \times D_{x_{i_2}} \times \ldots \times D_{x_{i_k}}$. The number k is called *arity* of the constraint C. A *solution* of CSP will be any n-tuple $(d_1,d_2,\ldots,d_n)$ from $D_{x_1} \times D_{x_2} \times \ldots \times D_{x_n}$ such that for each constraint $C \in \mathbf{C}$ over variables $x_{i_1},x_{i_2},\ldots,x_{i_k}$ k-tuple $(d_{i_1},d_{i_2},\ldots,d_{i_k})$ is in C. A CSP problem is *consistent* if it has a solution, and *inconsistent* otherwise.

A constraint C over variables $x_{i_1},x_{i_2},\ldots,x_{i_k}$ is *hyper-arc consistent* if for each value $d_{i_r} \in D_{x_{i_r}}$ ($r \in \{1,\ldots,k\}$) there are values $d_{i_s} \in D_{x_{i_s}}$ for each $s \in \{1,\ldots,k\} \setminus \{r\}$, such that $(d_{i_1},\ldots,d_{i_k}) \in C$. Assuming the domains are ordered, a constraint C over variables $x_{i_1},x_{i_2},\ldots,x_{i_k}$ is *bound consistent* if for each value $d_{i_r} \in \{min(D_{x_{i_r}}),max(D_{x_{i_r}})\}$ ($r \in \{1,\ldots,k\}$) there are values $d_{i_s} \in [min(D_{x_{i_s}}),max(D_{x_{i_s}})]$ for each $s \in \{1,\ldots,k\} \setminus \{r\}$, such that $(d_{i_1},\ldots,d_{i_k}) \in C$. A CSP problem is *hyper-arc consistent* (*bound consistent*) if all its constraints are.

Constraints whose arity is greater then two are often called *global* constraints. The one that is especially interesting for us is mentioned `alldifferent` constraint defined as follows:

$$\texttt{alldifferent}(x_{i_1},x_{i_2},\ldots,x_{i_k}) = \{(d_{i_1},d_{i_2},\ldots,d_{i_k}) \mid d_{i_j} \in D_{x_{i_j}}, r \neq s \Rightarrow d_{i_r} \neq d_{i_s}\}$$

3 The `alldifferent` algorithms

In this section we present all the `alldifferent` algorithms used in our solver. The section aims to be self-contained, so we first shortly describe algorithms not contributed by this paper, but important for understanding our novel algorithm. These include Ford-Fulkerson's algorithm ([11]), Regin's algorithm ([30]) and Katsirelos' algorithm ([21]). After that, we provide a detailed description of the MOS problem and propose an algorithm for solving it. Then we discuss the application of the MOS problem for explaining inconsistencies and propagations for the `alldifferent` constraint. We also compare the explanations obtained by MOS with those obtained by Katsirelos' algorithm and the flow-based explaining algorithms.

3.1 Checking for consistency

A *bipartite graph* $B = (U,V,E)$ is an undirected graph with vertices divided into two disjoint subsets U and V, such that for each edge $(u,v) \in E$, u is in U and v is in V. A *matching* in the bipartite graph B is a set of edges $\mathbb{M} \subseteq E$ such that no two edges from $\mathbb{M}$ have a vertex in common. If an edge (u,v) belongs to $\mathbb{M}$, we say that the vertices u and v are *matched*. A vertex is *free* if it is not matched with any other vertex. A matching $\mathbb{M}$ is *optimal* if there is no matching of a greater cardinality (i.e. containing a greater number of edges). An optimal matching is *perfect* if all vertices in U are matched.

We can assign a bipartite graph $B = (U,V,E)$ to a constraint $\texttt{alldifferent}(x_{i_1},\ldots,x_{i_k})$ in the following way: U contains one vertex u^x for each variable $x \in \{x_{i_1},\ldots,x_{i_k}\}$, and V contains one vertex v^d for each value $d \in \bigcup_{j=1}^{k} D_{x_{i_j}}$. The edge (u^x,v^d) belongs to E if and only if $d \in D_x$ (each variable is connected to values from its domain). It is easy to see that the following theorem holds.

Theorem 1 *An assignment $x_{i_1} = d_{i_1},\ldots,x_{i_k} = d_{i_k}$ satisfies the constraint $\texttt{alldifferent}(x_{i_1},\ldots,x_{i_k})$ if and only if $\mathbb{M} = \{(u^{x_{i_j}},v^{d_{i_j}}) \mid 1 \leq j \leq k\}$ is a perfect matching in B. Consequently, the constraint $\texttt{alldifferent}(x_{i_1},\ldots,x_{i_k})$ is satisfiable if and only if there is a perfect matching in B.* $\square$

Theorem 1 implies that the `alldifferent` constraint satisfiability can be reduced to the problem of finding an optimal matching in B — if such matching is not perfect, the constraint is unsatisfiable.

One of the procedures commonly used for finding an optimal matching is the Ford-Fulkerson's algorithm ([11]). The algorithm starts from some existing (non-optimal) matching $\mathbb{M}$ (possibly empty) and incrementally extends it until an optimal matching is obtained. The *residual graph* $G_{B,\mathbb{M}}$ for a bipartite graph B with a matching $\mathbb{M}$ is the directed graph with the set of vertices $U \cup V \cup \{s,t\}$ $(s,t \notin U \cup V)$. Its edges correspond to the edges of B, where edges in $\mathbb{M}$ are oriented from V to U, and edges not in $\mathbb{M}$ are oriented from U to V. The vertex s is connected to all vertices from U — an edge is oriented from s to $u \in U$ if u is not matched, and from u to s otherwise. The vertex t is connected to all vertices from V — an edge is oriented from t to $v \in V$ if v is matched, and from v to t otherwise. It can be proven that the matching $\mathbb{M}$ can be augmented in B if and only if there is a directed path from s to t in $G_{B,\mathbb{M}}$ (a proof of this can be found in [18]). Such paths are called *augmenting paths*. After an augmenting path is discovered (by a simple breadth-first search (BFS) based procedure), the directions of the edges in the path are reversed, and the matching is changed accordingly (i.e. the edges oriented from V to U are added to the matching, and those oriented from U to V are removed from the matching). This way, each augmenting path increases the cardinality of the current matching by one. The runtime

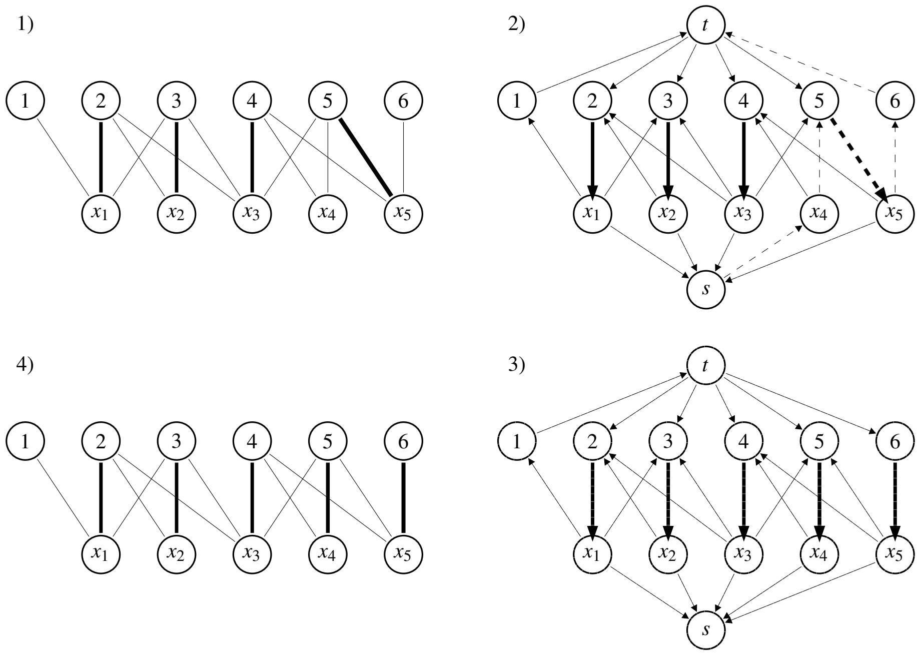

Fig. 1: Ford-Fulkerson's algorithm

of the procedure depends on the difference between the number of edges in the initial matching and in the obtained optimal matching. Therefore, the procedure will run faster if the initial matching is close to optimal. This makes the procedure suitable for incremental applications (such as SMT).

Example 1 Let us look at the example in Figure 1. The first graph shows a bipartite graph assigned to an `alldifferent` constraint (bold edges are in $\mathbb{M}$), and the second graph is the corresponding residual graph (dashed edges represent an augmenting path). The third graph is the residual graph after reversing the directions of the edges in the discovered augmenting path, and the fourth graph shows the new matching in the bipartite graph.

3.2 Enforcing hyper-arc consistency

In order to enforce hyper-arc consistency on a constraint, we must remove all the values from the domains of the constraint's variables that are not part of any assignment satisfying the constraint. The removal of an inconsistent value from a variable domain is called *pruning*. As we have already seen, in case of the `alldifferent` constraint such values correspond to the edges of the graph that are not part of any perfect matching. We call such edges *inconsistent edges*. All such edges should be found and removed (*pruned*) from the graph, and the corresponding prunings should be propagated.

Another important type of edges are *vital edges* — the edges that belong to all perfect matchings. If an edge is vital, it means that the corresponding value is contained in all solutions that satisfy the constraint, so the corresponding assignment should be propagated.

Edges that are neither vital nor inconsistent, i.e. that belong to some, but not all perfect matchings are called *alternating edges*. The following theorem trivially holds.

Theorem 2 *Each edge e in a bipartite graph B with a perfect matching $\mathbb{M}$ is either inconsistent, vital or alternating. Furthermore, if an edge e is not alternating, it is vital if it belongs to the perfect matching $\mathbb{M}$ and it is inconsistent otherwise.* □

According to Theorem 2, if we manage to find all the alternating edges in the graph, then it will be easy to classify the remaining edges as vital or inconsistent, based on their presence in the perfect matching $\mathbb{M}$. This is the main idea of Regin's algorithm ([30]).

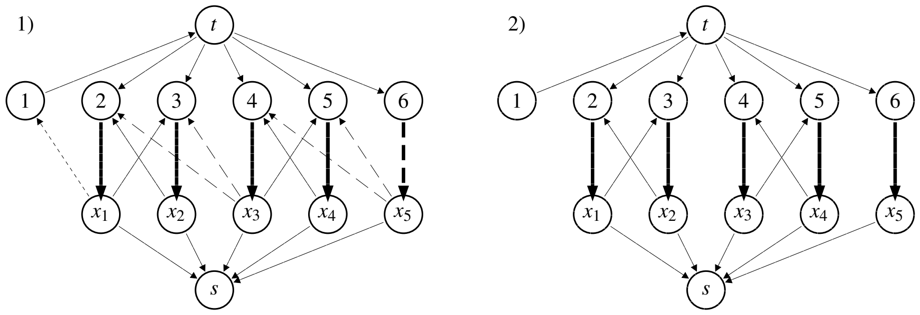

Fig. 2: Regin's algorithm

The following theorem gives us an effective way to find all the alternating edges in a bipartite graph.

Theorem 3 *Let B be a bipartite graph with a perfect matching $\mathbb{M}$. An edge e is alternating in B if and only if it belongs to a directed cycle in the residual graph $G_{B,\mathbb{M}}$.* □

A proof of Theorem 3 can be found in [4]. The edges belonging to directed cycles of $G_{B,\mathbb{M}}$ can be detected by finding strongly connected components of $G_{B,\mathbb{M}}$. A *strongly connected component* (SCC) in a directed graph is a maximal subset of vertices such that its every two vertices are mutually reachable. An edge (u,v) belongs to some directed cycle if and only if the vertices u and v are in the same SCC. SCCs can be found by Tarjan's depth first search (DFS) based algorithm ([41]).

Example 2 Let us look at the example in Figure 2. Assume that $x_1 \neq 1$ is asserted (i.e. the pruning is imposed on the constraint), causing the removal of the edge $(x_1, 1)$ from the graph (dotted edge in the graph 1). When Regin's algorithm is executed, it first finds and marks all alternating edges (by running Tarjan's algorithm). The edges not marked (dashed edges in the graph 1) are either vital (edge $(x_5, 6)$) or inconsistent (other four dashed edges). After the corresponding assignments and prunings are propagated, the inconsistent edges are removed from the graph (shown in the graph 2).

Regin's algorithm can be alternatively described in terms of Hall sets. A *Hall set* S is a set of variables such that $|S| = |T|$, where T is the union of domains of variables in S (the *combined domain*). Since variables in S must take distinct values, it follows that all values in T will be consumed by the variables from S. Therefore, we can prune the values in T from the domains of variables not in S. Regin's algorithm in essence does exactly that: it calls Tarjan's algorithm to find SCCs, and prunes all the edges whose vertices are in distinct SCCs. Since SCCs correspond to Hall sets, an edge crossing between two SCCs actually connects a variable x from one Hall set with a value d that is consumed by another Hall set. Therefore, $x = d$ cannot be part of a satisfying assignment.

3.3 Katsirelos' explanation algorithm

An *explanation* of a propagation (pruning or assignment) l is any subset E of the set R of all prunings that occurred before l such that E implies l. Similarly, an explanation of a conflict (i.e. an inconsistency of the constraint) is any subset E of the set R of all prunings at the time when the conflict occurred such that E is sufficient to cause the inconsistency of the constraint. Naturally, we want the explanations to be as small as possible.

Katsirelos' explanation algorithm ([21]) directly exploits the previously discussed relations between the Regin's algorithm and the Hall sets.

Theorem 4 *Let $x \neq d$ be the pruning that should be explained, and let S and T be, respectively, the Hall set and the corresponding combined domain such that $x \notin S$ and $d \in T$. Let $E = \{x' \neq d' \mid x' \in S,\ d' \notin T\}$, that is, let E be the set of all prunings of values from the domains of variables in S, excluding the values from the combined domain T. Then E is an explanation of the pruning $x \neq d$.*

Proof Notice that the set E is exactly the set of prunings that caused the set S to become a Hall set. Now, according to the previous discussion, the value $d \in T$ cannot be assigned to the variable $x \notin S$, meaning that E implies $x \neq d$. $\square$

Since the original Katsirelos' algorithm produces explanations eagerly, that is, immediately after Regin's algorithm has been invoked, the SCCs are known in that moment, so it is easy to find the SCC that contains a pruned value (i.e. the SCC $S \cup T$ where S is a Hall set and T is its combined domain such that $d \in T$). In order to explain the prunings lazily, the previous state of the graph (at the moment the pruning was made) must be reconstructed (using the information about the chronology of prunings that is maintained during the operation of the solver), and then the Tarjan's algorithm must be executed again in order to rediscover the SCCs ([25], [15]).

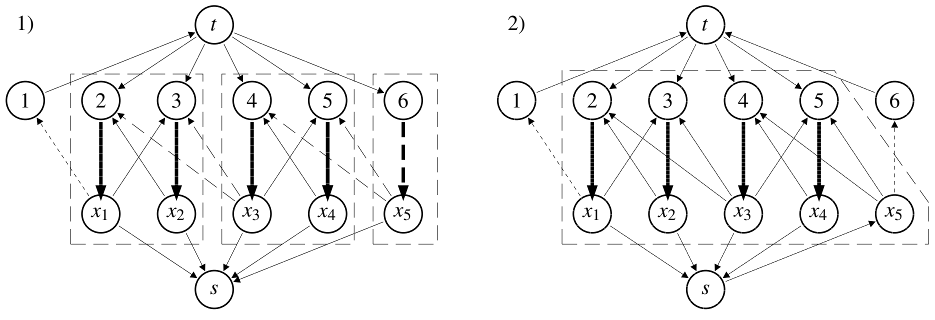

Fig. 3: Katsirelos' explanations

A conflict may be explained in a similar fashion ([8]). An `alldifferent` constraint is inconsistent if and only if there exists a set of variables S such that $|S| > |T|$, i.e. there are more variables in S than values in its combined domain T (the proof of this fact can be found in [18]). The following theorem describes conflict explanations based on the Katsirelos' algorithm.

Theorem 5 *Let B be the bipartite graph assigned to an inconsistent* `alldifferent` *constraint, and let $\mathbb{M}$ be an optimal (but not perfect) matching in B. Let $u \in U$ be an unmatched vertex (that corresponds to an unassigned variable), S be the set of variables corresponding to the vertices from U that are reachable from u in $G_{B,\mathbb{M}}$ and let T be the combined domain of S. The set of prunings $E = \{x' = d' \mid x' \in S,\ d' \notin T\}$ is an explanation for the conflict.*

Proof Notice that the set of vertices reachable from u in $G_{B,\mathbb{M}}$ does not contain an unmatched vertex in V (since the matching is optimal) and it must contain more vertices from U (corresponding to the set of variables S) than vertices from V (corresponding to the combined domain T), since the vertex u is unmatched. The set E is exactly the reason why $|S| > |T|$ and, according to the previous discussion, the reason why the constraint is inconsistent. □

Example 3 Continuing Example 2, after the pruning is done, we identify 3 Hall sets (framed parts of the graph 1 in Figure 3). Two of them correspond to SCCs ($\{x_1,x_2\}$ and $\{x_3,x_4\}$) and the third is a singleton variable set $\{x_5\}$. The explanation of the propagated assignment $x_5 = 6$ consists of pruned edges that leave the frame of the corresponding Hall set ($x_5 \neq 4$ and $x_5 \neq 5$). The explanation of the pruning $x_5 \neq 4$ consists of pruned edges that leave the frame of the Hall set that consumes the value 4 ($x_3 \neq 2$ and $x_3 \neq 3$). In a similar fashion, the explanation of the pruning $x_3 \neq 3$ is the pruning $x_1 \neq 1$.

The graph 2 in Figure 3 is an example of conflict explaining. Let us assume that $x_1 \neq 1$ and $x_5 \neq 6$ are asserted. When we prune the corresponding edges (dotted edges in the graph 2), the vertex x_5 is not matched any more. Starting graph traversal from that vertex, we reach all the vertices within the framed area of the graph. The framed area contains 5 vertices in U (the set of variables $S = \{x_1,x_2,x_3,x_4,x_5\}$) and 4 vertices in V (the combined domain $T = \{2,3,4,5\}$). Notice that $|S| > |T|$. The explanation of the unsatisfiability consists of the pruned edges that leave the frame ($x_1 \neq 1$ and $x_5 \neq 6$).

3.4 Minimal obstacle set problem (MOS)

Let $G = (V,E)$ be a directed graph, with a set of *start vertices* $S \subseteq V$ and a set of *final vertices* $F \subseteq V$. We say that a set of *obstacles* $O \subseteq E$ *separates* S from F if an arbitrary path from any vertex $v \in S$ to any vertex $f \in F$ contains at least one edge from O. We say that the vertex $v \in V$ is *blocked* by the obstacle set O (or O*-blocked*) if O separates the set $\{v\}$ from F, and that it is O*-unblocked* otherwise. Clearly, if O separates S from F, then each $v \in S$ is O-blocked, and each $f \in F$ is O-unblocked. A path that contains no obstacles from O is called an *obstacle-free* or O*-free* path. A set of obstacles O that separates S from F is a *minimal obstacle set* if there is no proper subset O' of O that also separates S from F. The following theorem gives necessary and sufficient conditions for a set of obstacles O to be a minimal obstacle set.

Theorem 6 *Given a graph G, and sets of vertices S and F, a set of obstacles O that separates S from F is minimal if and only if for each obstacle $e = (v,w) \in O$ the vertex v is reachable from some vertex $u \in S$ through some O-free path and the vertex w is O-unblocked.*

Proof Suppose that each edge from O satisfies given conditions. We prove that O is a minimal obstacle set. Assume the opposite, that some O' that is a proper subset of O also separates S from F. Then exists $e = (v,w) \in O \setminus O'$. Since v is reachable from some $u \in S$ through an O-free path, and w is O-unblocked which means that some vertex $f \in F$ is reachable from w through an O-free path, and since $e \notin O'$, there is an O'-free path from u to f and that is in contradiction with the fact that O' separates S from F.

Now suppose that O is a minimal obstacle set. We prove that for each edge $e = (v,w) \in O$ the vertex v is reachable from some vertex $u \in S$ through some O-free path, and that the vertex w is O-unblocked. Assume the opposite, that for some obstacle $e = (v,w)$ there is either no O-free path from vertices from S to v, or there is no O-free path from w to vertices from F. In this case, the set $O' = O \setminus \{e\}$ also separates S from F, because there is no O'-free path from S to F. This is in contradiction with the fact that O is a minimal obstacle set. □

The *minimal obstacle set problem* (MOS) for a given set of obstacles O that separates S from F is defined as follows: find an obstacle set $O_{min} \subseteq O$ such that O_{min} also separates S from F and is a minimal obstacle set. It is clear that such set does not have to be unique.

The procedure `findMinimalObstacleSet` (Algorithm 1) first looks for obstacles $e = (v,w) \in O$ such that v is reachable (through an O-free path) from some vertex in S. It calls the procedure `BFSFindReachableObstacles` (Algorithm 2) that starts BFS in G from the vertices in S, using only O-free paths. The set of obstacles reached in that way is denoted by O_r. It is easy to argue that O_r also separates S from F.

```
Require: $G = (V, E)$ is a directed graph, $S \subseteq V$, $F \subseteq V$.
Require: $O \subseteq E$ set of obstacles that separates $S$ from $F$.
Ensure: $O_{min} \subseteq O$ is minimal obstacle set.
  $O_r$ = BFSFindReachableObstacles$(G, S, O)$
  $V_u$ = DFSTarjanUnblockedVertices$(G, F, O_r)$
  $O_{min} = \varnothing$
  for all $(v, w) \in O_r$ do
    if $w \in V_u$ then $O_{min} = O_{min} \cup \{(v, w)\}$
  return $O_{min}$
```

Algorithm 1: `findMinimalObstacleSet`(G, S, F, O)

In the second stage, for each obstacle $e = (v, w) \in O_r$ it is checked if the vertex w is O_r-unblocked. For this purpose, the procedure `DFSTarjanUnblockedVertices` (Algorithm 3) is called. This procedure returns the set of all O_r-unblocked vertices in G (denoted by V_u). An obstacle $e = (v, w) \in O_r$ is then added to O_{min} if and only if $w \in V_u$. According to Theorem 6, such obstacle set O_{min} is minimal.

Example 4 Consider the graph in Figure 4. The set of start vertices is $S = \{1\}$, and the set of final vertices is $F = \{7, 8\}$. In the first graph, there is the initial set of obstacles $O = \{(2, 4), (3, 6), (3, 5), (4, 8)\}$ (the edges marked with the black dots). The obstacle set O_r without the unreachable obstacle $(4, 8)$ is shown in the second graph. Finally, the minimal obstacle set $O_{min} = \{(2, 4), (3, 6)\}$ is shown in the third graph. The obstacle $(3, 5)$ is removed, because the vertex 5 is blocked by the obstacle $(2, 4)$.

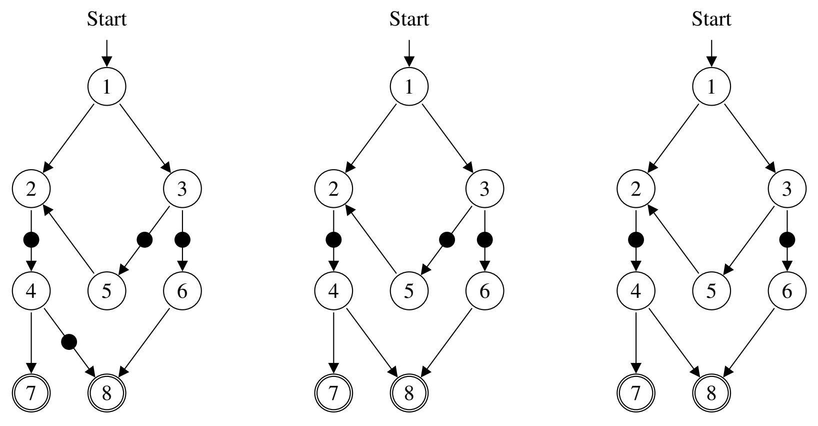

Fig. 4: Minimal obstacle set

The procedure `DFSTarjanUnblockedVertices` (Algorithm 3) deserves more detailed explanation. It is based on Tarjan's algorithm for finding SCCs ([41]). The motivation for using Tarjan's algorithm for detection of unblocked vertices relies on

```
Require: $G = (V, E)$ is a directed graph
Require: $S \subseteq V$ is a set of start vertices
Require: $O \subseteq E$ is a set of obstacles
Ensure: $O_r$ is the set of obstacles reachable from $S$ through $O$-free paths.
  $q.init()$ { $q$ is a vertex queue }
  $O_r = \varnothing$
  for all $u \in S$ do
    $q.enqueue(u)$
    $u.marked = \texttt{true}$
  while not $q.empty()$ do
    $v = q.dequeue()$
    for all $(v, w) \in E$ do
      if $(v, w) \in O$ then
        $O_r = O_r \cup \{(v, w)\}$
      else if not $w.marked$ then
        $w.marked = \texttt{true}$
        $q.enqueue(w)$
  return $O_r$
```

Algorithm 2: `BFSFindReachableObstacles`(G, S, O)

```
Require: $G = (V, E)$ is a directed graph, $F \subseteq V$
Require: $O \subseteq E$ is a set of obstacles
Ensure: $V_u$ is the set of all $O$-unblocked vertices
  $index = 1$
  $S.init()$ {$S$ is a vertex stack}
  $V_u = \varnothing$
  for all $v \in V$ do
    if $v.index = undef$ then
      DFSTarjan_rec($v$)
  return $V_u$

  procedure DFSTarjan_rec($v$)
  $v.index = index$
  $v.lowlink = index$
  $index = index + 1$
  $S.push(v)$
  if $v \in F$ then $V_u = V_u \cup \{v\}$
  for all $(v, w) \in E \setminus O$ do
    if $w.index = undef$ then
      DFSTarjan_rec($w$)
      $v.lowlink = min(v.lowlink, w.lowlink)$
    else if $w \in S$ then
      $v.lowlink = min(v.lowlink, w.index)$
    if $w \in V_u$ then $V_u = V_u \cup \{v\}$
  if $v.lowlink = v.index$ then
    repeat
      $w = S.pop()$
      if $v \in V_u$ then $V_u = V_u \cup \{w\}$
    until $w = v$
```

Algorithm 3: `DFSTarjanUnblockedVertices`(G, F, O)

the following fact: for each SCC W from G, if one of its vertices is unblocked, then all its vertices are unblocked (due to the mutual reachability of vertices from the same SCC). Therefore, it is a property of a SCC, not of a vertex itself. We say that a SCC

W is O-unblocked if its vertices are O-unblocked, and it is O-blocked otherwise. For better understanding, the original Tarjan's algorithm is explained first, and then the modifications included in our algorithm are discussed. Tarjan's algorithm basically performs a standard DFS of the graph G. During the search, nodes in V are indexed in the order in which they are first visited, i.e., in the *preorder* fashion. The index assigned to the vertex v is denoted by $v.index$ (initially $v.index = undef$, meaning that the vertex is not visited yet). For a SCC W, the *root* of W is the vertex from W that is first visited during the search, i.e., the vertex from W with the lowest preorder index. Each vertex v is also assigned a number $v.lowlink$ that is computed during the search and that represents the lowest preorder index of the vertices reachable from v. Thus, a vertex u is the root of its SCC if and only if $u.index = u.lowlink$. The vertices are first pushed on the vertex stack until the root of their SCC is found, when they are popped from the stack and added to their SCC.

The procedure `DFSTarjanUnblockedVertices` differs from described Tarjan's algorithm in the following two points. First, we consider strong connectivity by using only O-free paths (i.e. with the set of edges $E \setminus O$). Second, the algorithm does not return found SCCs as its output. Instead, it uses them to find the set V_u of O-unblocked vertices. The procedure adds the following vertices to V_u (and only them):

– if $v \in F$, then v is added to V_u (final vertices are trivially unblocked)
– if $(v,w) \notin O$ and w is already in V_u, then v is added to V_u.
– if v is the root of a SCC W and v is in V_u, then all vertices in W are added to V_u.

The correctness of the algorithm follows from the next two theorems.

Theorem 7 *The procedure* `DFSTarjanUnblockedVertices` *is* sound*: if v is in V_u at the procedure's end, then v is O-unblocked.*

Proof If a vertex v is added to V_u during the procedure's execution, then it is either because $v \in F$, or $(v,w) \notin O$ and w is already in V_u, or the root of the SCC that v belongs to is already in V_u. In either of these three cases, let k_v be the size of the set V_u right before v is added to V_u. We prove that v is O-unblocked by induction on k_v. For $k_v = 0$ (i.e. the set V_u is empty), the only applicable is the case when $v \in F$, so v is trivially O-unblocked. Assume that $k_v > 0$ and the state holds for all $k' < k_v$. In the first case, if $v \in F$, then v is again trivially unblocked. In the second case, the vertex v is being added to V_u because some vertex w is in V_u and $(v,w) \notin O$. Since w is added to V_u before v, then $k_w < k_v$, so according to the inductive hypothesis it holds that w is O-unblocked. Therefore, v is also unblocked, due to the existence of the edge (v,w). In the third case, the vertex v is being added to V_u because the root w of its SCC is in V_u. Again, w is added to V_u before v, so $k_w < k_v$, meaning that w is O-unblocked, according to the inductive hypothesis. Therefore, v is also O-unblocked, since there is a directed path from v to w. This proves that each vertex v that is added to V_u in any step of the algorithm's execution is O-unblocked. □

Theorem 8 *The procedure* `DFSTarjanUnblockedVertices` *is* complete*: if u is O-unblocked, then u will be in V_u at the procedure's end.* □

Proof For completeness, we must prove that for each unblocked component W, all its vertices will be added to V_u. In this proof we use the following notation:

- $W(v)$ is the strongly connected component that contains v.
- $root(W)$ is the root of the strongly connected component W.
- $root(v)$ is the root of the strongly connected component that contains v (i.e. $root(v) = root(W(v))$).
- a *distance* of an unblocked strongly connected component W (denoted by $d(W)$) is the length of the shortest O-free path $(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$, such that $v_0 \in W$ and $v_k \in F$. Notice that if $W \cap F \neq \varnothing$, then $d(W) = 0$.
- The recursive call `DFSTarjan_rec`(v) (shortly denoted by *v-call*) is *active* if the procedure executes either v-call or some recursive call invoked directly or indirectly from v-call.
- $v \rightarrow w$ means that w-call is invoked (directly) from v-call. The reflexive and transitive closure of the relation $\rightarrow$ is denoted by $v \rightarrow^* w$ (meaning $v = w$ or $v \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow w$). In other words, $v \rightarrow^* w$ means that v-call is still active while the w-call is executing. Notice that $root(v) \rightarrow^* v$ for each vertex v.
- we say that an unblocked vertex v is *simply unblocked* (or *s-unblocked*) if v is added to V_u *before* v-call ends. Otherwise, it is *delayed unblocked* (or *d-unblocked*). A delayed unblocked vertex v is added to V_u *after* v-call ends, but before $root(v)$-call ends (in the final stage of $root(v)$-call, when its strongly connected component is poped from the stack).

Notice that if $v \in F$, then v is s-unblocked, since at the beginning of v-call the procedure checks if v is in F and adds v to V_u, accordingly. Thus, v will certainly be in V_u before v-call ends.

We first prove that if $v \rightarrow^* w$ and if w is s-unblocked, then v is also s-unblocked. To prove this, we first consider the case when $v \rightarrow w$. In this case, w-call is called directly from v-call. When w-call ends and the execution returns to v-call, it will be detected that w is in V_u (since w is s-unblocked) and v will also be added to V_u. The same holds for $\rightarrow^*$, since it is the transitive closure of $\rightarrow$. Notice that, since $root(v) \rightarrow^* v$, if v is s-unblocked, then $root(v)$ is also s-unblocked.

In the rest of the proof, we show that for each unblocked strongly connected component W the vertex $root(W)$ is s-unblocked. We prove this by induction on the distance $d(W)$. For $d(W) = 0$ the state trivially holds, since there is a vertex $v \in W \cap F$ and v is s-unblocked, so $root(v) = root(W)$ is also s-unblocked. Assume that the state holds for $d(W) < k$. We prove that it holds for $d(W) = k$. Indeed, if $d(W) = k$, this means that there is an O-free path $P = (v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$, such that $v_0 \in W$, $v_k \in F$ and that is the shortest such path (thus, $v_1,\ldots,v_k \notin W$). We prove that while v_0-call is executing v_1-call is not active. Indeed, otherwise $v_1 \rightarrow^* v_0$, meaning that v_0 is reachable from v_1. It means that $v_1 \in W$, opposite to assumption. The same can be stated for $root(v_1)$-call. This means that v_1-call and $root(v_1)$-call are either both finished or have not been started yet at the moment when the edge (v_0,v_1) is visited from v_0-call. One of the following can happen:

- v_1 has been already visited. In this case, v_1-call and $root(v_1)$-call have finished. $W(v_1)$ is also unblocked (through the path $P \setminus (v_0,v_1)$ of the length $k-1$) and $d(W(v_1)) = k-1 < d(W)$. Therefore, by induction it follows that $root(v_1)$ is s-unblocked. Because of that, the vertex v_1 is in V_u, since all vertices from $W(v_1)$ are added to V_u at the end of $root(v_1)$-call.

– v_1 has not been visited. In this case, v_1-call and $root(v_1)$-call have not been started yet. This means that v_1 is the first vertex from $W(v_1)$ to be visited, i.e. $root(v_1) = v_1$. The v_1-call is then executed. Again, by induction it follows that v_1 is s-unblocked and it will be in V_u when v_1-call ends.

In both cases, the procedure detects that $v_1 \in V_u$ and adds v_0 to V_u. Since it happens during the execution of v_0-call, the vertex v_0 is s-unblocked. Therefore, $root(v_0) = root(W)$ is also s-unblocked.

If for some component W the vertex $root(W)$ is s-unblocked, then at the final stage of $root(W)$-call all vertices from W are added to V_u. Since we have proven that for all unblocked components W their roots are s-unblocked, it follows that all unblocked vertices will be in V_u at the end of the execution of the procedure. This proves the completeness. □

Complexity. Although the procedure `findMinimalObstacleSet` is based on two graph traversals, it can be optimized to execute in only one traversal. Namely, the vertices visited during the execution of the procedure `BFSFindReachableObstacles` are reachable from S and are certainly O_r-blocked (otherwise some vertex $u \in S$ would be unblocked and that is in contradiction with the fact that O_r separates S from F). This means that there is no need to visit these vertices again in the procedure `DFSTarjanUnblockedVertices` and check if they are unblocked. Therefore, the complexity is $O(|V| + |E|)$.

3.5 Using MOS for explanations

Let B be a bipartite graph initially assigned to the constraint `alldifferent`$(x_1, \dots, x_k)$. After a set R of prunings is asserted, some edges will be removed from the graph. We denote this set of removed edges by E_R, and the new state of the graph by B_R. Assume that B_R is in a consistent state, i.e. there is a perfect matching $\mathbb{M}$ in B_R, and that Regin's algorithm detects a vital or an inconsistent edge in B_R. The corresponding propagation (assignment or pruning, respectively) can be explained at any time later, as stated by the following theorem.

Theorem 9 *Consider the MOS problem for the residual graph $G_{B,\mathbb{M}}$, with the set of obstacles $O = E_R$, the set of start vertices $S = \{v\}$ and the set of final vertices $F = \{u\}$, where (u, v) is the directed edge that corresponds to the propagated pruning or assignment. Let O_{min} be the found minimal obstacle set. The set of prunings that corresponds to the set of edges in O_{min} is a minimal propagation explanation.*

Proof First, notice that all edges from $\mathbb{M}$ are also present in B. The edge (u, v) is alternating in B, but is vital or inconsistent in B_R. It means that all directed cycles that contained (u, v) in B are now broken due to the removal of edges in E_R. Therefore, all directed paths in $G_{B,\mathbb{M}}$ from v to u must contain at least one edge from $O = E_R$ — that means that O is a set of obstacles that separates v from u. Found minimal obstacle set will correspond to a minimal (in sense of inclusion) propagation explanation — because of these prunings the edge (u, v) is not alternating in B_R. □

Assume now that B_R is in an inconsistent state, that is, an optimal matching $\mathbb{M}$ is not perfect. The conflict explanation may be reduced to MOS in a similar fashion, as stated by the following theorem.

Theorem 10 *Consider the MOS problem in the residual graph $G_{B,\mathbb{M}}$ with the set of start vertices $S = \{u\}$, where u is any unmatched vertex from U (that corresponds to an unassigned variable), the set of final vertices $F = \{t\}$ and the set of obstacles $O = E_R$. Let O_{min} be the found minimal obstacle set. The set of prunings that corresponds to the set of edges in O_{min} is a minimal conflict explanation.*

Proof Since $\mathbb{M}$ is optimal in B_R, there is no augmenting path in the residual graph $G_{B_R,\mathbb{M}}$, i.e. all the paths from any unmatched $u \in U$ to t in the graph $G_{B,\mathbb{M}}$ contain at least one edge from $O = E_R$ — that means that the set O separates u from t. Found minimal obstacle set corresponds to a minimal conflict explanation — because of these prunings the vertex u cannot be matched in B_R. □

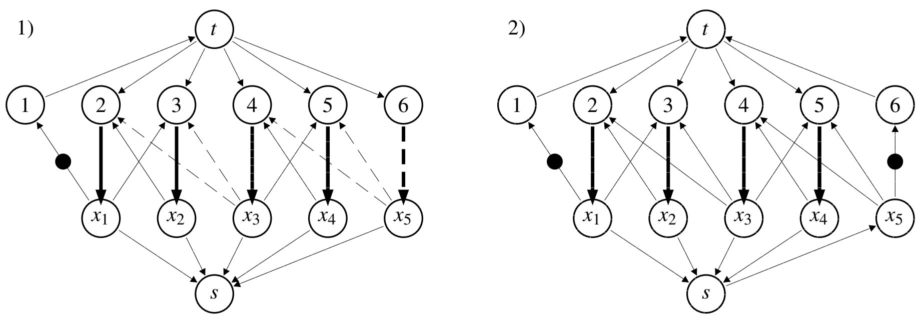

Fig. 5: MOS explanations (black dots represent the obstacles)

Example 5 Let us look again at the same graph as in Example 3. The first graph in Figure 5 is the residual graph, with an obstacle put on the edge $(x_1, 1)$ (the edge removed due to the assertion $x_1 \neq 1$). Dashed edges correspond to prunings and assignments caused by the removal of the edge $(x_1, 1)$. In order to find an explanation of the assignment $x_5 = 6$ (the vital edge $(6, x_5)$) we look for a minimal obstacle set that separates the vertex x_5 from the vertex 6. In case of explaining the pruning $x_5 \neq 4$ (inconsistent edge $(x_5, 4)$), we look for a minimal obstacle set that separates the vertex 4 from the vertex x_5. In both cases, it is the obstacle at the edge $(x_1, 1)$, so the explanation is $x_1 \neq 1$.

We can use the second graph in Figure 5 to describe the conflict explanation algorithm. We put obstacles on edges $(x_1, 1)$ and $(x_5, 6)$ that correspond to the asserted prunings $x_1 \neq 1$ and $x_5 \neq 6$. In order to find the explanation, we look for a minimal obstacle set that separates s and t. In this case, it includes both obstacles, so the explanation is $x_1 \neq 1, x_5 \neq 6$.

3.6 Comparison of MOS and Katsirelos explanations

Comparing examples 3 and 5, we see that Katsirelos' explanations for the assignment $x_5 = 6$ and the pruning $x_5 \neq 4$ are $\{x_5 \neq 4, x_5 \neq 5\}$ and $\{x_3 \neq 2, x_3 \neq 3\}$, respectively. In case of MOS explanations, both are explained by the pruning $x_1 \neq 1$. In general, MOS tends to find explanations that are deeper in the conflict graph than the ones found by Katsirelos' algorithm. Let us look again at the propagations found by Regin's algorithm in Example 2. Described in terms of Hall sets, first the edge $(x_1, 1)$ is removed, causing the set $\{x_1, x_2\}$ to become a Hall set. Because of this, the edges $(x_3, 2)$ and $(x_3, 3)$ are pruned, causing the set $\{x_3, x_4\}$ to become a Hall set. This causes removal of edges $(x_5, 4)$ and $(x_5, 5)$, and because of that the edge $(x_5, 6)$ becomes vital. This means that we have the following chain of implications:

$$x_1 \neq 1 \Rightarrow \{x_3 \neq 2, x_3 \neq 3\} \Rightarrow \{x_5 \neq 4, x_5 \neq 5\} \Rightarrow x_5 = 6$$

In the conflict analysis phase, Katsirelos' algorithm takes only one step backwards in this implication chain, while the MOS-based algorithm goes all the way to the beginning of the implication chain. Notice that all three implications in the chain are actually detected locally within the `alldifferent` constraint during the execution of Regin's algorithm, triggered by the asserted pruning $x_1 \neq 1$. The real cause of all propagations is, thus, the asserted pruning $x_1 \neq 1$, but this fact is not captured by Katsirelos' algorithm. As we can see from this example, a single pruning can break a SCC into three or more smaller SCCs. Katsirelos' algorithm is based on SCCs, so it can find only *minimal* Hall sets — those that correspond to SCCs. It is easy to see that the union of two disjoint Hall sets is also a Hall set, but such Hall sets cannot be found by Katsirelos' algorithm. For instance, finding the Hall set $\{x_1, x_2, x_3, x_4\}$ would produce the explanation $x_1 \neq 1$ immediately, but this set remains undiscovered by the Katsirelos' algorithm. On the other hand, MOS algorithm operates on the initial graph, with obstacles put only on the edges removed due to the assertions. Therefore, MOS algorithm can reach the obstacles that are the real cause for propagations.

Of course, this limitation of Katsirelos' algorithm may be overcome during the conflict analysis by multiple invocations of the explaining algorithm. However, there are the cases when this problem cannot be overcome. Let us look at the example in Figure 6. The example is the same as previous, except we have two asserted prunings $x_1 \neq 1$ and $x_3 \neq 3$ (notice that the second pruning would be entailed by the first in Regin's algorithm). In the first graph in Figure 6, dotted edges correspond to the assertions, dashed edges correspond to the propagations, and framed parts correspond to Hall sets. Katsirelos' algorithm explains the pruning $x_5 \neq 4$ with $\{x_3 \neq 2, x_3 \neq 3\}$ as before. The pruning $x_3 \neq 2$ can be further explained by $x_1 \neq 1$, but the pruning $x_3 \neq 3$ will not be explained by Katsirelos' algorithm, since it is not propagated by this `alldifferent` constraint (it may be explained by some other propagator, probably introducing new literals to the conflict). In the second graph in Figure 6, we see two obstacles that correspond to the asserted prunings. When explaining the pruning $x_5 \neq 4$, the first phase of the MOS algorithm (starting from the vertex 4) finds that both obstacles are reachable, but in the second stage, the obstacle $(x_3, 3)$ is eliminated, since its end vertex 3 is blocked by the obstacle $(x_1, 1)$. Therefore, the explanation is $x_1 \neq 1$. Notice that we avoid explaining $x_3 \neq 3$ by the other propagator, since it does not contribute to the propagation of $x_5 \neq 4$. This example shows that MOS may potentially give smaller propagation explanations, by eliminating some redundant parts of the implication graph.

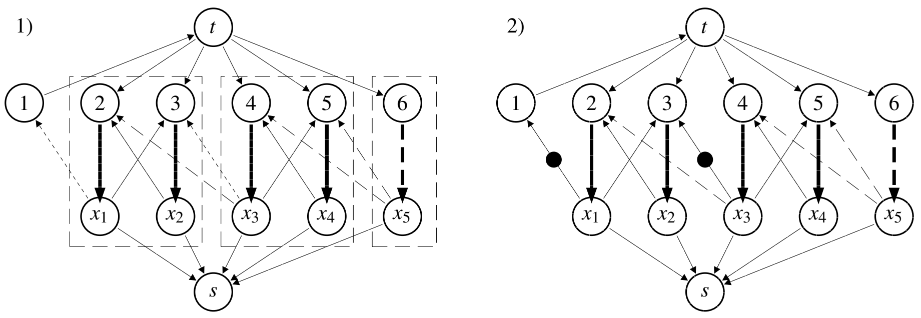

Fig. 6: Comparison of Katsirelos' and MOS explanations

3.7 Comparison of MOS and flow-based explanations

Another interesting approach to `alldifferent` explanations is based on *flow networks* ([11]). In fact, most of the previous discussion concerning the `alldifferent` constraint can be reformulated in terms of flow networks. For instance, the satisfiability checking is equivalent to finding a feasible flow in the corresponding flow network (Figure 7), while the arc-consistency (that is, finding vital and inconsistent edges) is equivalent to finding maximal and minimal flows in the corresponding edges of the flow network. In the similar fashion, the problem of explaining inconsistencies and propagations in the `alldifferent` constraint can be reduced to explaining maximal/minimal flows. For instance, explaining the fact that an edge (x,d) cannot be a part of a perfect matching in B (that is, explaining the pruning $x \neq d$) is reduced to explaining why the maximal flow in the edge (x,d) is equal to 0. Similarly, explaining the fact that an edge (x,d) is vital in B (that is, explaining the assignment $x = d$) is reduced to explaining why the minimal flow in the edge (x,d) is equal to 1. The minimal and maximal flow explaining is based on finding *minimal cuts* in the flow network (i.e. cuts with minimal capacity), due to the famous *min-cut/max-flow* theorem ([11]). When such cut is found, a flow bound for one of its incoming edges can be explained by the flow bounds imposed on its other incoming/outgoing edges ([32], [9]). The important point here is that the minimal cut *is not unique* — there can be more than one minimal cuts and some of them may be preferred for explaining over others. In this section we briefly discuss different approaches based on minimal cuts in flow networks and relate them to our approach based on MOS.

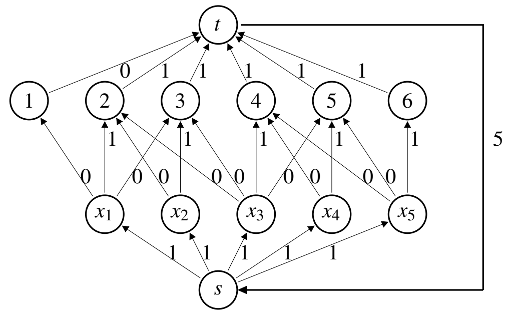

Fig. 7: The flow network for the `alldifferent` constraint from the previous examples. The lower and the upper flow bounds for all edges are 0 and 1, respectively, except for the edge (t,s), for which both the lower and the upper flow bounds are 5 (this guarantees that all the variables are matched). Feasible flows correspond to perfect matchings. An example of a feasible flow is given by the numbers near the edges.

The first approach is based on discovering SCCs in the residual graph[1]. Such approach is considered in [9]. It relies on the fact that the edges (x,d) that cross between different SCCs in the residual graph coincide with the edges whose flows are fixed at their lower or the upper bounds — which, in case of the `alldifferent` correspond to the pruned and vital edges (the proof of this fact can be found in [31]). This means that the SCC of the vertex d can be used as a minimal cut for explaining the flow bounds for (x,d). In the case of the `alldifferent` constraint, Katsirelos' algorithm does exactly that. In [9], this approach is generalized to an arbitrary flow network, but the authors admit that their approach, applied to the `alldifferent` constraint, gives the same explanations as Katsirelos' algorithm ([9], [8]). We already discussed how the explanations obtained this way relate to our MOS-based explanations. In short, the main drawback of this approach is that the explanations must be generated after the SCC-splitting ([16]) is done and, therefore, these explanations may include the bounds (prunings in case of the `alldifferent`) that are also propagated by the same constraint. This increases the length of the implication chain (as discussed in the previous section). The algorithm described in [9] can also be used for explaining assignments, but the same drawback still holds.

The second approach is based on discovering reachable vertices in the residual graph ([32]). Given an edge (x,d) whose maximal/minimal flow should be explained, a traversal of the residual graph if started from the vertex d and all vertices reachable from d are marked. This set of marked vertices is also a minimal cut ([32]) that can be used for explaining the flow bounds of (x,d). In order to relate such explanations to our MOS-based explanations, recall that the MOS algorithm explained in Section 3.4 has two phases. In the first phase it discovers reachable obstacles (i.e. marks vertices reachable from d and finds all pruned edges that used to connect these reachable vertices to other vertices). In the second phase it eliminates redundant obstacles

[1] Notice that our definition of the residual graph from Section 3.1 is rather simplified, compared to the usual general definition of the residual graph in the flow networks ([31]), but the two definitions coincide in case of the graph assigned to the `alldifferent` constraint.

whose end vertices are blocked by other reachable obstacles (which correspond to the pruned edges that cannot be included in augmenting paths, since their end vertices cannot reach the vertex x). Obviously, the approach from [32] is equivalent only to the first phase of our MOS algorithm. This means that it can give explanations that have redundant edges, as admitted by the authors ([32]). In fact, our MOS-based explanations coincide with the explanations that are formally stated in [32] as the preferred explanations — a MOS-based explanation includes exactly the pruned edges that connect the vertices reachable from d with the vertices that can reach the vertex x (notice that in [32] there is no algorithm provided for obtaining such explanations).

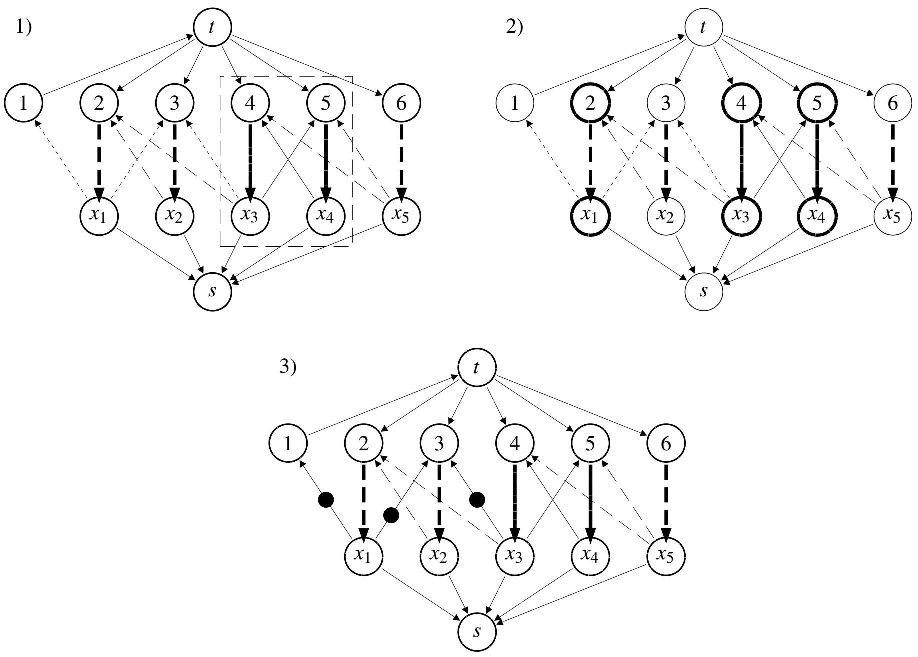

Fig. 8: Comparison of flow-based and MOS-based explanations

Example 6 Consider the same residual graph from the previous examples (Figure 8). Assume that there are three asserted prunings: $x_1 \neq 1$, $x_1 \neq 3$ and $x_3 \neq 3$. These prunings impose upper flow bounds for the corresponding edges to be equal to 0 (dotted edges in the graphs). The propagation algorithm calculates maximal/minimal flows in the edges and propagates prunings for those edges with the maximal flow equal to 0 and assignments for those edges with the minimal flow equal to 1 (dashed edges in the graphs). Suppose that we want to explain the pruning $x_5 \neq 5$. In case of SCC-based algorithm, first we discover the SCC that contains the vertex 5 (the framed area in the first graph in Figure 8) and then use it as a cut. Its incoming flow is bounded from above by 0 because its outgoing flow is bounded from above by 0, due to the flow bounds in the edges $(x_3, 2)$ and $(x_3, 3)$. Thus, the explanation is $x_3 \neq 2$,

$x_3 \neq 3$ (the same as Katsirelos' explanation). Another way to explain the pruning is to find the set of reachable vertices from the vertex 5, avoiding the dotted edges (the bold vertices in the second graph in Figure 8). This set of vertices is used as a cut. Its incoming flow is again bounded from above by 0, but this time because of the bounds of *its* outgoing edges' flows — the explanation is $x_3 \neq 3$, $x_1 \neq 1$, $x_1 \neq 3$. Finally, the third way to explain the pruning is to use MOS (the third graph in Figure 8). The set of reachable obstacles consists of all three obstacles (as in the previous case), but our algorithm recognizes that only one of them $(x_1, 1)$ indeed breaks a path that leads to the vertex x_5, so the explanation is: $x_1 \neq 1$.

4 Solver description

In this section, we describe the structure of our SMT solver and the corresponding CSP theory[2] solver in more details. The solver conforms to DPLL(**T**) ([12]) architecture. The main data structures of the SAT engine are the *set of clauses F* that must be satisfied, the *assertion trail M* and the *conflict set C*. The assertion trail M is a stack-like sequence of literals that are set to *true* in the current partial valuation. To enable non-chronological backtracking (*backjumping*), the literals in M are partitioned into *decision levels*, starting from the level 0. Literals asserted immediately after a new level is established are *decision literals*, while other literals are *inferred literals*. Decision literals are suitable backtracking points. The conflict set C is either an inconsistent subset of literals in M if such subset exists, or a special no_cflct value, otherwise.

The operation of the solver can be described in terms of the *state transition rules* ([12], [22]) given in Figure 9. Each rule has a set of *premises* (written above the horizontal rule) and an *action* that changes the state of the solver (written below the horizontal rule). The `Decide` rule is a case-splitting rule that establishes a new decision level in M. The rules `UnitPropagate`, `Conflict` and `Explain` are used in boolean reasoning, while the rules `TheoryPropagate`, `TheoryConflict` and `TheoryExplain` describe the reasoning in some theory **T**. The rules `UnitPropagate` and `TheoryPropagate` are the inference rules. When a conflict is detected, the `Conflict` (or `TheoryConflict`) rule is applied and the conflict analysis starts. The literals in the conflict set C are explained one by one by using the `Explain` and `TheoryExplain` rules until the first *unique implication point (UIP)* ([12]) is reached. Then the `Backjump` rule is applied, reverting the solver to an appropriate decision level m, and the search continues. The rule `Restart` may be applied periodically to help the solver get out from an unfruitful branch of the search space.

Theory solvers must implement the interface shown in Figure 10. In order to support the rules `Decide` and `Backjump`, each theory solver must implement procedures `newLevel` and `backjump` that enable reverting its state to some previous point. The most important procedure to implement is `checkAndPropagate` procedure. It should

[2] The *CSP theory* refers to the standard first-order theory of integers (as described in SMT-LIB ([3])) extended with the axioms that define the meaning of the global constraints.

$$\texttt{Decide}: \frac{l \in c \quad c \in F \quad l, \bar{l} \notin M}{M := M \mid l} \qquad \texttt{Backjump}: \frac{C = \{l, l_1, \ldots, l_k\} \quad \text{level}(l) > m \geq \text{level}(l_i)}{C := no\_cflct \quad F := F \cup \{\bar{l} \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k\} \quad M := M^{[m]}\bar{l}} \qquad \texttt{Restart}: \frac{C = no\_cflct}{M := M^{[0]}}$$

$$\texttt{UnitPropagate}: \frac{l \vee l_1 \vee \ldots \vee l_k \in F \quad \bar{l}_1, \ldots, \bar{l}_k \in M \quad l, \bar{l} \notin M}{M := Ml} \qquad \texttt{TheoryPropagate}: \frac{M \vDash_{\mathbf{T}} l \quad l, \bar{l} \notin M}{M := Ml}$$

$$\texttt{Conflict}: \frac{C = no\_cflct \quad \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \quad l_1, \ldots, l_k \in M}{C := \{l_1, \ldots, l_k\}} \qquad \texttt{TheoryConflict}: \frac{C = no\_cflct \quad l_1, \ldots, l_k \vDash_{\mathbf{T}} \bot \quad l_1, \ldots, l_k \in M}{C := \{l_1, \ldots, l_k\}}$$

$$\texttt{Explain}: \frac{l \in C \quad l \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \quad l_1, \ldots, l_k \prec l}{C := C \cup \{l_1, \ldots, l_k\} \setminus \{l\}} \qquad \texttt{TheoryExplain}: \frac{l \in C \quad l_1, \ldots, l_k \vDash_{\mathbf{T}} l \quad l_1, \ldots, l_k \prec l}{C := C \cup \{l_1, \ldots, l_k\} \setminus \{l\}}$$

Fig. 9: DPLL(**T**) rules ($\bar{l}$ denotes the literal opposite to l; $M \mid l$ means that l is a decision literal; $level(l)$ denotes the decision level of l; $M^{[m]}$ denotes the prefix of M up to the decision level m; $M \vDash_{\mathbf{T}} l$ denotes that l is a **T**-consequence of M; symbol $\prec$ denotes the order of literals in M)

be able to detect **T**-inconsistencies of M. If a conflict is detected, the procedure generates an explanation of the conflict and applies the `TheoryConflict` rule, starting the conflict analysis. It there is no conflict in **T**, the procedure looks for **T**-inferred literals and propagates them by applying the `TheoryPropagate` rule (the propagated literals are then sent to all other interested theory solvers by the SAT engine). The important parameter of the `checkAndPropagate` procedure is a *deduction layer* — our solver support a layered approach, where cheaper and simpler algorithms may be invoked first, while the invocation of stronger but more expensive algorithms may be postponed (or invoked only periodically). A similar approach is already used in SMT ([6]) and in CP ([34]). Finally, the procedure `explainLiteral` is called by the SAT engine during the conflict analysis for literals that are previously propagated by the theory. The procedure generates explanation of that propagation and applies the `TheoryExplain` rule.

Interface procedure	**Description**
`newLevel`()	called by the SAT engine whenever a new decision level is established in M.
`assertLiteral`(l)	called by the SAT engine whenever a new literal l is pushed on M
`backjump`(m)	called by the SAT engine whenever it backjumps to the level m.
`checkAndPropagate`($layer$)	checks whether the current trail M is **T**-consistent and propagates **T**-inferred literals to M.
`explainLiteral`(l)	generates an explanation of the propagated literal l and applies the resolution.

Fig. 10: **T**-solver interface

4.1 The input language

The solver accepts inputs in SMT-LIB 2.0 format ([3]). The logic that we use is called `QF_CSP` — it is based on the standard `QF_LIA` logic, but it additionally supports function symbols that represent global constraints, and also restricts its semantics such that only finite domains are used. Let us look at the following example:

```
(set-logic QF_CSP)
(declare-fun x_1 () Int) (declare-fun x_2 () Int) (declare-fun x_3 () Int)
(declare-fun x_4 () Int) (declare-fun x_5 () Int) (declare-fun x_6 () Int)
(assert (and
   (<= 1 x_1 9) (<= 1 x_2 9) (<= 1 x_3 9) (<= 1 x_4 9) (<= 1 x_5 9)
   (<= 1 x_6 9) (= (+ x_1 x_2) 13) (= (+ (* 2 x_3) x_4 x_5) 25)
   (or (<= (- x_2 x_4) 7) (>= (+ x_3 x_6) x_1))
   (alldiff x_1 x_2) (alldiff x_1 x_3 x_4 x_5) (alldiff x_2 x_4 x_6)
 ))
(check-sat)
(get-value (x_1)) (get-value (x_2)) (get-value (x_3))
(get-value (x_4)) (get-value (x_5)) (get-value (x_6))
(exit)
```

Variables of the CSP are represented by constants of sort `Int` (`x_1` to `x_6` in the example). Since we are solving finite domain CSPs, all the variables must have finite domains and this must be specified in the formula (in this example, all the variables take values from the domain $\{1,2,\dots,9\}$). After the domains are specified, the list of constraints follows. The `alldifferent` constraint is represented by the `alldiff` function symbol. This symbol may be of any arity $n \geq 2$, it is applied only to constants of sort `Int` and its return sort is `Bool`. At this moment, the implementation limits `alldiffs` only to appear as unit clauses, since the opposite constraint (i.e. `not_alldiff`) is still not supported. Beside the `alldiff` predicates, an input file may contain arbitrary linear arithmetic constraints. They have the same syntax as in `QF_LIA`, and may be combined using the boolean connectives in an arbitrary way.

4.2 CSP theory solver

The CSP theory solver understands two types of literals: *constraints* (such as `alldifferent`(x_1,x_2,x_3), or $2x_1 - 3x_2 + x_3 = 17$) and *domain restrictions* (such as $x \leq 7$, $x = 3$, or $x \neq 2$). Constraint literals correspond to the global constraints that have to be satisfied, while domain restriction literals correspond to the variable domain changes. The SAT engine sends all such literals to the theory solver (by invoking its `assertLiteral` procedure) as soon as they appear on the trail M.

The CSP theory solver consists of *domain handlers* and *constraint handlers*. We assign a domain handler to each CSP variable and a constraint handler to each constraint appearing in the input formula. A constraint handler may be *active* or *inactive*, depending on whether its corresponding constraint literal is *true* in M.

The purpose of a domain handler is to initialize and maintain the (finite) domain of the corresponding variable. It detects trivial consequences between domain restriction literals (like $x < 5 \vDash x \neq 7$ or $x = 3 \vDash x \neq 4$) and propagates such inferred literals.

It also detects conflicts due to *domain wipeouts*, i.e. when all the values are pruned from the domain.

Constraint handlers are of different types, depending on the type of the global constraint they represent. Currently, only two types of constraint handlers are implemented — `alldifferent` constraint handler and `weighted_sum` constraint handler (the latter is responsible for the linear arithmetic constraints). Each constraint handler implements an appropriate filtering algorithm and is able to detect inconsistencies and to propagate literals that correspond to prunings made by the filtering algorithm. Constraint handlers must be backtrackable and *may* support the layered approach. Finally, a constraint handler must be able to explain conflicts and propagations that it has detected.

The CSP theory solver implements the interface in the Figure 10 in the following way. The calls of the procedures `newLevel` and `backjump` are simply delegated to all domain and constraint handlers. When a constraint literal is asserted, the corresponding constraint handler is activated. When a domain restriction literal is asserted, all active handlers that are affected by the literal (i.e. the domain of one of their variables is changed) are pushed to the *handler queues* (there is one such queue for each supported layer). When `checkAndPropagate` is invoked at some layer, the handlers are taken from the corresponding queue one by one and their filtering algorithms are executed. A handler may implement different algorithms at different layers, or may execute the same algorithm at all layers — it is only important that the algorithms at higher layers are deductively at least as strong as those at lower layers. A handler applies the `TheoryConflict` rule if it detects an inconsistency and applies the `TheoryPropagate` rule when the domain of a variable is changed by the filtering algorithm. Since execution of a filtering algorithm may prune additional values from the variable domains and, thus, propagate more domain restriction literals to M, the SAT engine may invoke the `assertLiteral` procedure again. This may cause adding more constraint handlers to the handler queue, if their variables are affected by the newly propagated literals. The whole process finishes once the handler queue becomes empty.

Domain restriction literals are introduced lazily by the constraint handlers, when the corresponding domain change occurs for the first time. This is very important, since the number of possible domain restriction literals tends to grow very big, in case of large CSP problems. The supported domain restriction literals are of the form $x \bowtie d$, where $\bowtie \in \{=, \neq, \leq, \geq, <, >\}$. Propagation of the domain restriction literals (through the assertion trail M) is an essential part of the solver execution, since it is the only communication mechanism between the constraint handlers.

During the conflict analysis, when the interface procedure `explainLiteral` is invoked, the CSP theory solver delegates the call to the handler that caused the propagation (this information is stored in the CSP theory solver when the literal is propagated). The handler now generates the explanation and applies the `TheoryExplain` rule.

4.3 The `alldifferent` constraint handler

The handler for the constraint `alldifferent`$(x_1,\ldots,x_k)$ consists of a bipartite graph whose state can be restored when the `Backjump` rule is applied. When the domain restriction literals are asserted, the appropriate edges are removed from the graph. For each removed edge e, it stores the literal responsible for its removal (denoted by $cause(e)$). This information is used for explaining.

The handler has two layers. At the first layer, its `checkAndPropagate` procedure detects only simple conflicts such as $x_i = d, x_j = d \vDash \bot$, as well as trivial propagations such as $x_i = d \vDash x_j \neq d$ (for $i \neq j$). Therefore, at the first layer the constraint handler establishes the same level of consistency as the set of disequalities $x_i \neq x_j$, where $1 \leq i, j \leq k$ and $i \neq j$.

At the second layer the procedure first runs Ford-Fulkerson's algorithm to check if there is a conflict. In that case, it invokes the procedure for conflict explanation and applies the `TheoryConflict` rule. Otherwise, it runs Regin's algorithm to detect and propagate equalities (that correspond to newly found vital edges in the graph) and disequalities (that correspond to pruned inconsistent edges in the graph).

Both Katsirelos' algorithm and the MOS-based algorithm for explaining conflicts and propagations are implemented, in order to compare the behaviour of the two algorithms within the same solver. Since the obtained explanation is expressed as a set of pruned edges (as described in Section 3), it must be transformed into the set of literals by replacing each edge e by the literal $cause(e)$. Notice that the explanation may also contain inequalities, since an inequality may also be the cause of an edge removal.

4.4 The `weighted_sum` constraint handler

The `weighted_sum` constraints are represented as $a_1 \cdot x_1 + \ldots + a_k \cdot x_k \bowtie c$, where $\bowtie \in \{=, \neq, \leq, <, \geq, >\}$, a_i and c are integers and x_i are finite domain integer variables. Our `weighted_sum` constraint handler implements the standard algorithm for enforcing *bounds consistency* in such constraints ([35]). For each variable, new bounds are calculated based on the bounds of other variables. The values that violate new bounds are pruned from the domain. The handler then propagates the inequalities that correspond to the calculated bounds. The process is repeated until a fixed point is reached.

5 Experimental results

In this section we present an experimental evaluation of our solver described in the previous sections (called `argosmt`). The main goal of the evaluation is to compare the performance of Katsirelos' and MOS-based explaining algorithms within our solver. Another goal is to compare our solver with the state-of-the-art SMT and constraint solvers. In order to achieve these goals, the experiments were performed with the following solvers:

- `argosmt-mos` — our solver[3] using MOS explanations
- `argosmt-kat` — our solver using Katsirelos' explanations
- `sugar-argosmt` — Sugar SAT-based constraint solver ([40]) using `argosmt` as the back-end SAT solver
- `sugar-minisat` — Sugar SAT-based constraint solver using `minisat`[4] as the back-end SAT solver
- `minion` — Minion constraint solver ([13])
- `g12lazy` — G12[5] lazy clause generation solver ([28])
- `opturion` — Opturion[6] lazy clause generation solver
- `yices-lia` — Yices SMT solver ([10]), using `QF_LIA` logic and `distinct` predicate to represent `alldifferent`
- `yices-bv` — Yices SMT solver using `QF_BV` (bitvectors) logic

We tested these solvers on the following six sets of instances:[7]

sudoku25 — the set consists of 200 randomly generated *Sudoku* ([23]) instances of size 25×25. We used 200s cutoff time per instance. The boards are generated with around 45% fields filled in — Sudoku exhibits phase transition and these boards tend to be the hardest ones ([23]). Sudoku instances are encoded using only `alldifferent` constraints — one per each row, column and square.

sudoku36 — similarly to the previous set, it consists of 100 randomly generated Sudoku instances, but of larger size — 36×36 (this time, cutoff is 600s).

kakuro — the set consists of 100 randomly generated *Kakuro* problems ([37]). We used the cutoff time of 600s for these instances. The problems are encoded in a straightforward fashion: for each continuous block of white cells either in a row or in a column we have one `alldifferent` constraint that forces the values in the cells to be pairwise distinct, and one arithmetic constraint that forces the sum of the values in the cells to be equal to the number given in the corresponding black cell.

golfers — the set consists of 65 *Social Golfer* ([14]) instances of different sizes. We used uniform cutoff time of 1200s for all instances, although the hardness of these instances varies from very easy to extremely hard, depending on the size. The similar model as in [14] is used. Let m be the number of groups, n be the number of players per group and p be the number of rounds. The number of players is then nm, and the players are denoted by numbers $0, 1, \dots, mn-1$. The variable x_{ijk} has the domain $\{0, 1, \dots, mn-1\}$ and represents the jth player in ith group in kth round. For each fixed $k \in \{1, \dots, p\}$ we have one `alldifferent` constraint over variables x_{ijk}, for $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n\}$ — it states that in each round none of the players cannot be at two different positions at the same time. In order to state that there are no two players that play more than once in the same team, we establish a "1-1" correspondence between pairs of players and numbers in $\{0, \dots, m^2n^2 - 1\}$: $(u, v) \to nm \cdot u + v$. Now, we introduce a variable $y_{ij_1j_2k}$ for each pair of variables

[3] `http://www.matf.bg.ac.rs/~milan/argosmt/`

[4] `http://minisat.se/`

[5] `http://www.minizinc.org/g12distrib.html`

[6] `http://www.opturion.com/`

[7] All instances can be found at: `http://www.math.rs/~milan/argosmt/instances.zip`

(x_{ij_1k}, x_{ij_2k}), where $j_1 < j_2$. The domain for all these variables is $\{0,\dots,m^2n^2-1\}$. We introduce arithmetic constraints $y_{ij_1j_2k} = nm \cdot x_{ij_1k} + x_{ij_2k}$. Finally, we have one additional `alldifferent` constraint over all "*y*" variables — it states that no pair of players may appear more than once together in the same team. Such encoding is mainly based on `alldifferent`, but also includes a lot of arithmetic constraints over variables with very large domains.

timetabling — this instance set considers a problem that is more suitable for practical applications: the set consists of 100 randomly generated instances of a variant of the *timetabling* problem: we consider a high school with 40 different groups where each group attends 35 classes per week, scheduled in five working days (seven classes per day). Each class occupies one of the available time slots of equal lengths (in each day there are exactly seven time slots). The problem imposes that some of the classes must be scheduled in blocks (of size 2 or 3) — the classes that form a block must be scheduled in the same day and must occupy consecutive time slots. In our instances, there are five 3-blocks and nine 2-blocks, while the remaining two classes may be scheduled arbitrarily. The classes are taught by a number of teachers, where each teacher has from 14 to 20 classes per week. The randomization is achieved by assigning the groups' classes to the teachers in a random way (the classes that form a block must be assigned to the same teacher). The number of classes per week for each teacher is also chosen randomly from the interval $[14,20]$. To encode the problem, we introduce a variable x_{ijk} for each teacher i who teaches the kth class to the group j (according to the class-to-teacher allocation that is known in advance for each instance). Each such variable has domain $\{0,\dots,34\}$ — the value of a variable x_{ijk} determines the time slot in which the corresponding class takes place (the values $0,\dots,6$ are the time slots of the first day, $6,\dots,13$ are the time slots of the second day, etc.). To express that ith teacher cannot teach two different classes at the same time, we impose an `alldifferent` constraint over all variables x_{ijk} for this fixed i. Similarly, to express that jth group cannot attend two different classes at the same time, we impose an `alldifferent` constraint over all variables x_{ijk} for this fixed j. Finally, to express the conditions about blocks, we state that for two variables x_{ijk_1} and x_{ijk_2} representing two consecutive classes in a block it holds that $x_{ijk_1} + 1 = x_{ijk_2}$. Also, in each block, all classes except the last one must not be scheduled in the last time slot of a day (that is, the corresponding variable cannot take neither of the values 6, 13, 20, 27, 34).

wqgc — the set consists of 100 randomly generated instances of the *weighted quasigroup completion* problem[8] ([36]). The problem is similar to the standard quasigroup completion problem, but in addition each cell (i,j) of the corresponding *latin square* has an assigned *weight* p_{ij} – a positive integer from some predefined interval. The goal is to complete the quasigroup starting from the pre-given values, minimizing the value of $M = min_i(\sum_j p_{ij}x_{ij})$, where x_{ij} are the values of the cells. Of course, this is an optimization problem, but we consider its decision variant — is there a correct quasigroup completion such that $M \leq K$, where K is a positive number? The problem is encoded using `alldifferent` constraints over x_{ij} variables (one `alldifferent`

[8] As stated in [36], this problem, as a typical example of a *combinatorial design problem*, has structural properties that are often seen in many practical applications.

for each row and column). Also, for each row, we introduce a variable $y_i = \sum_j p_{ij}x_{ij}$ that represents the weighted sum for that row, and assert the clause $\bigvee_i(y_i \leq K)$ — it ensures that the minimum of the y_i variables is at most K. All instances in the set are of the size 30×30, the weights are between 1 and 100 and the boards are generated with around 42% of cells filled in (the point of the phase transition ([17])). The number K is chosen based on some previously found quasigroup completion, so we know that all instances are satisfiable.

	sudoku25		**sudoku36**		**kakuro**	
	# instances	**cutoff**	**# instances**	**cutoff**	**# instances**	**cutoff**
	200	300s	100	600s	100	600s
	# solved	**avg. time**	**# solved**	**avg. time**	**# solved**	**avg. time**
argosmt-mos	200	1.03s	97	99.7s	100	15.7s
argosmt-kat	200	2.46s	69	307s	99	24.7s
sugar-argosmt	188	72.5s	0	600s	100	10.8s
sugar-minisat	200	13.3s	7	456s	100	2.03s
minion	186	35.9s	23	476s	16	520s
g12-lazy	198	21.8s	3	587s	98	20.7s
opturion	198	39.2s	0	600s	100	4.9s
yices-lia	49	277s	0	600s	12	566s
yices-bv	0	300s	0	600s	100	32.4s

	golfers		**timetabling**		**wqgc**	
	# instances	**cutoff**	**# instances**	**cutoff**	**# instances**	**cutoff**
	65	1200s	100	1200s	100	1200s
	# solved	**avg. time**	**# solved**	**avg. time**	**# solved**	**avg. time**
argosmt-mos	44	463s	99	187s	82	444s
argosmt-kat	45	458s	99	209s	75	516s
sugar-argosmt	24	780s	12	1160s	0	1200s
sugar-minisat	30	660s	100	103s	0	1200s
minion	22	817s	0	1200s	14	1033s
g12-lazy	51	305s	19	1084s	13	1073s
opturion	43	439s	0	1200s	100	43s
yices-lia	1	1193s	0	1200s	0	1200s
yices-bv	46	492s	0	1200s	0	1200s

Table 1: The table shows for each solver and each instance set the number of solved instances within the given cutoff time per instance and the average solving time (for unsolved instances, the cutoff time is used)

The results in Table 1 show that the overall performance of our solver with MOS explaining algorithm is better than with Katsirelos' algorithm. The greatest speedup is achieved for *sudoku* instances, but is also noticeable for *wqgc* and *timetabling* instances. At the other side, there is no significant difference in performance on *kakuro* and *golfers* instances. Since the average time alone does not provide enough information about the runtime distribution, in Figure 11 we show how runtimes relate on particular instances. The points above the diagonal line represent the instances for which MOS explaining algorithm performed better than Katsirelos' algorithm. The chart looks quite convincing for *sudoku* instances (the upper left chart), where most of the points are highly above the diagonal. The lower two charts show the runtimes relation on *timetabling* (left) and *wqgc* (right) instances — while not so drastic this

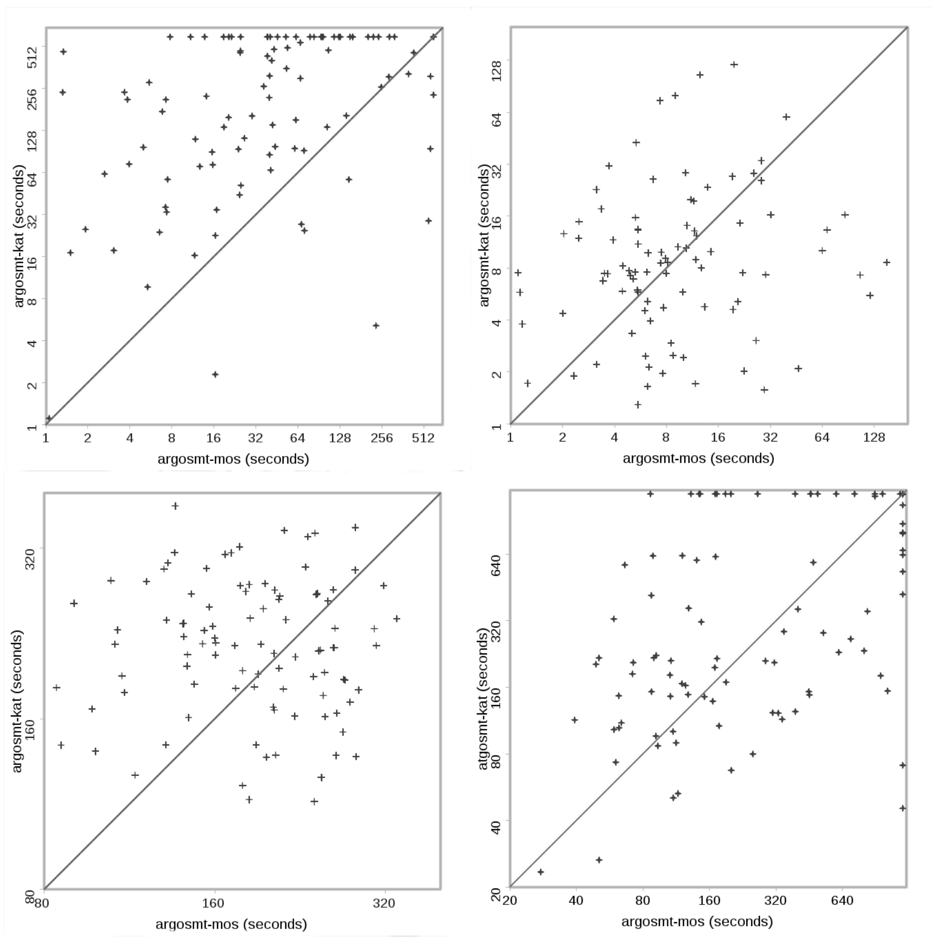

Fig. 11: `argosmt-mos` vs `argosmt-kat` solving times: *sudoku36* (upper left), *kakuro* (upper right), *timetabling* (lower left), *wqgc* (lower right)

time, the charts still judge in favor of MOS-based algorithm. Finally, for *kakuro* instances (the upper right chart), the points are almost evenly scattered on both sides of the diagonal, so we cannot deduce which algorithm is better. The information about the search space reduction (Table 2) also confirm that *sudoku* instances benefit the most from using MOS for explaining.

Different speedup obtained by using MOS algorithm on different instance sets may be roughly explained by the number and the lengths of the `alldifferent` constraints appearing in the instances from these sets. As discussed in Section 3.6, the main benefit of using MOS explaining algorithm may be in making the implication chains shorter, which is more likely to happen in long `alldifferent` constraints (i.e. the constraints with greater arities). As shown in Table 3, *sudoku*, *timetabling* and *wqgc* instances have both great numbers of the `alldifferent` constraints and

	sudoku25		sudoku36		kakuro	
	# decides	**# conflicts**	**# decides**	**# conflicts**	**# decides**	**# conflicts**
argosmt-mos	388	162	13059	6953	44761	31690
argosmt-kat	705	298	22440	9503	44491	31915
opturion	240008	149023	—	—	130489	74390

	golfers		timetabling		wqgc	
	# decides	**# conflicts**	**# decides**	**# conflicts**	**# decides**	**# conflicts**
argosmt-mos	66192	1877	188905	5833	27272	20364
argosmt-kat	58742	1740	220613	5051	28538	21200
opturion	187985	99113	—	—	112373	52255

Table 2: The average numbers of decides and conflicts (for solved instances)

great average lengths of these constraints (making the average cumulative length per instance more than 1500). In case of *kakuro*, the number of `alldifferent` constraint is 124 on average (comparable with *sudoku* and *timetabling* instances), but the average length of the `alldifferent` constraints is only 4.3. On the other hand, the length of the `alldifferent` constraints for the *golfers* instances is 52 on average, but such a great average is due to the one large `alldifferent` per instance (the one with "*y*" variables), while others are much shorter. The average number of `alldifferent` constraints per instance is only 10, so the cumulative length per instance is only 520 on average, which is much smaller than for *sudoku* and more comparable with *kakuro*. Another interesting information shown in Table 3 that can be useful in explaining the effectiveness of MOS algorithm on different instance sets is the portion of propagations that came from Regin's algorithm: in case of *sudoku25*, about 34% of all propagations originate from Regin's algorithm, and about 48% percent of all explanations are the explanations of these Regin's propagations. For *sudoku36*, it is about 39% of Regin's propagations on average, and 47% of Regin's explanations. For this reason, the choice of the algorithm for explaining Regin's propagations has a great impact on the solver's performance on these instances. On the other hand, for *kakuro* instances, less than 1% of all propagations originate from Regin's algorithm, and less than 2% of all explanations concern these propagations. Since the percent of explanations of Regin's propagations is relatively small, the choice of explanation algorithm cannot have substantial impact on performance.

	sudoku25	**sudoku36**	**kakuro**	**golfers**	**timetabling**	**wqgc**
# alldiffs	75	108	124	10	126	60
avg. length	25	36	4.3	52	22.2	30
cum. length	1875	3888	546	520	2800	1800
% Regin's prop.	34%	39%	0.8%	2%	6.4%	0.6%
% Regin's expl.	48%	47%	1.7%	5.7%	9.5%	7.7%

Table 3: The table shows the average number of `alldifferent` constraints per instance, the average length of the `alldifferent` constraints. and the average cumulative length of all `alldifferent` constraints per instance in each of the instance sets. It also shows the percents of propagations that came from Regin's algorithm and the percents of explanations that explain such propagations

Compared to other solvers, it can be seen (Table 1) that our solver is the best choice for *sudoku* instances, while on other problems it is comparable with other solvers (usually the second best choice). Moreover, if we sum the results on all instance sets, argosmt-mos solved 622 instances in total, while argosmt-kat solved 587 instances. In these terms, the next best solver is opturion which solved only 441 instances in total, while sugar-minisat solved 437 instances. All other solvers manage to solve significantly fewer instances. It is also important to notice that while other solvers' performance is very good on some instance sets, but very bad on others, our solver is uniformly good on all instance sets. This makes it quite reliable tool for solving CSP problems.

Yet another interesting conclusion may be drawn from comparison of the performance of the sugar solver when using argosmt and minisat as back-end SAT solvers. In Table 1 it can be seen that minisat performs several times better on some instance sets than our SAT engine. This shows that the implementation of our SAT engine which drives argosmt solver is still not comparable with the state-of-the-art SAT solvers (on which some other solvers used in the experiments are based). Making the SAT engine faster may further enhance the performance of our solver and make it more competitive with other tools. An additional clue that may confirm this hypothesis is the information about the explored search space given in Table 2: the solver opturion has significantly greater number of decides and conflicts on all instance sets, although on some instance sets it runs faster than our solver. This may indicate that its implementation is much faster, while the explored search space is greater than in case of our solver (however, we are not sure whether these numbers are comparable, since the structure of the search space and the nature of the literals used by opturion may be very different).

We also see that yices solver, as a general state-of-the-art SMT solver is not a good choice for solving CSP problems. Using QF_LIA logic, the solver shows poor performance on all instance sets. This might be expected, since this logic is mostly used for quite different problems in SMT, usually involving infinite or very large domains. When QF_BF is used, the solver performs much better on *kakuro* and *golfers* instances, but is still useless on other instances. This confirms our claims that incorporating filtering algorithms for global constraints can dramatically improve SMT technology, when application in CSP solving is concerned.

6 Further work

One direction for future work may be in exploring other possibilities of integration of SMT and CSP technologies. For instance, incorporating filtering algorithms for other common global constraints into our SMT solver might extend its applicability in solving CSP problems. The things may go in the opposite direction too — extending the alldifferent algorithms to support infinite or very large domains may be beneficial for SMT solving. Another possible contribution to SMT solving may be in combination of the proposed CSP theory with other typical SMT theories, such as the theory of uninterpreted functions, the theory of arrays or the linear arithmetics.

Another possible branch of further work is making the implementation more efficient, especially when the SAT engine is concerned, but also optimizing the data structures used in the implementation of CSP algorithms, since the profiling shows that the operations on these structures take a significant portion of the execution time.

7 Conclusions

In this paper, we presented a new approach in explaining inconsistencies and propagations in `alldifferent` constraints based on MOS problem that we also introduced and solved. We compared our algorithm to the standard algorithms that are commonly used in constraint solvers for that purpose (Katsirelos' and flow-based explaining algorithms). We used our algorithms along with Ford Fulkerson's and Regin's algorithms to develop a DPLL(**T**)-compliant SMT theory solver that makes SMT solvers suitable for solving CSPs that include `alldifferent` constraints. This is very important, since it shows the potential of SMT in solving problems outside the software verification area. We presented some experimental results that showed that our algorithm performs better than Katsirelos' algorithm on `alldifferent`-dominant CSP instances. We also compared our prototypical implementation with other state-of-the-art solvers. Results show that using SMT solvers equipped with special-purpose algorithms can be comparable with other approaches when solving CSPs.

Acknowledgements This work was partially supported by the Serbian Ministry of Science grant 174021 and by the SNF grant SCOPES IZ73Z0_127979/1. The author is grateful to Filip Marić and to anonymous reviewers for very careful reading of the text and providing detailed and useful comments and remarks.

References

1. Bankovic, M., Maric, F.: An Alldifferent constraint solver in SMT. In: 8th International Workshop on Satisfiability Modulo Theories (2010)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability, chap. 26, pp. 825–885. IOS Press (2009)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf` (2010)
4. Berge, C.: Graphes et hypergraphes (1970)
5. Bofill, M., Palahí, M., Suy, J., Villaret, M.: Solving constraint satisfaction problems with SAT modulo theories. Constraints **17**(3), 273–303 (2012)
6. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Van Rossum, P., Schulz, S., Sebastiani, R.: An incremental and layered procedure for the satisfiability of linear arithmetic logic. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 317–333. Springer (2005)
7. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**(7), 394–397 (1962). DOI 10.1145/368273.368557. URL `http://doi.acm.org/10.1145/368273.368557`
8. Downing, N., Feydy, T., Stuckey, P.J.: Explaining alldifferent. In: Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122, pp. 115–124. Australian Computer Society, Inc. (2012)
9. Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, pp. 146–162. Springer (2012)
10. Dutertre, B., De Moura, L.: The Yices SMT solver. Tool paper at http://yices. csl. sri. com/tool-paper. pdf **2**, 2 (2006)

11. Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. Canadian journal of Mathematics **8**(3), 399–404 (1956)
12. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: CAV, *Lecture Notes in Computer Science*, vol. 3114, pp. 175–188. Springer (2004)
13. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI, vol. 141, pp. 98–102 (2006)
14. Gent, I.P., Lynce, I.: A SAT encoding for the social golfer problem. Modelling and Solving Problems with Constraints p. 2 (2005)
15. Gent, I.P., Miguel, I., Moore, N.C.: Lazy explanations for constraint propagators. In: Practical Aspects of Declarative Languages, pp. 217–233. Springer (2010)
16. Gent, I.P., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: An empirical survey. Artificial Intelligence **172**(18), 1973–2000 (2008)
17. Gomes, C., Shmoys, D.: Completing quasigroups or latin squares: A structured graph coloring problem. In: proceedings of the Computational Symposium on Graph Coloring and Generalizations, pp. 22–39 (2002)
18. van Hoeve, W.J.: The alldifferent constraint: A survey. arXiv preprint cs/0105015 (2001)
19. Janicic, P.: Uniform Reduction to SAT. Logical Methods in Computer Science **8**(3) (2010)
20. Janicic, P., Maric, F.: Uniform reduction to SMT (2010)
21. Katsirelos, G.: Nogood processing in CSPs. Ph.D. thesis, University of Toronto (2008)
22. Krstic, S., Goel, A.: Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In: FroCoS, *Lecture Notes in Computer Science*, vol. 4720, pp. 1–27. Springer (2007)
23. Lewis, R.: Metaheuristics can solve sudoku puzzles. Journal of heuristics **13**(4), 387–401 (2007)
24. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-Driven Clause Learning SAT Solvers. In: Handbook of Satisfiability, chap. 4, pp. 131–155. IOS Press (2009)
25. Moore, N.: Improving the efficiency of learning CSP solvers. Ph.D. thesis, University of St Andrews (2011)
26. Nieuwenhuis, R.: Sat modulo theories: Enhancing SAT with special-purpose algorithms. In: Theory and Applications of Satisfiability Testing-SAT 2009, pp. 1–1. Springer (2009)
27. Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Challenges in satisfiability modulo theories. In: Term Rewriting and Applications, pp. 2–18. Springer (2007)
28. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3), 357–391 (2009)
29. Petke, J., Jeavons, P.: The order encoding: from tractable CSP to tractable SAT. In: Theory and Applications of Satisfiability Testing-SAT 2011, pp. 371–372. Springer (2011)
30. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: AAAI, vol. 94, pp. 362–367 (1994)
31. Régin, J.C.: Generalized arc consistency for global cardinality constraint. In: Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1, pp. 209–215. AAAI Press (1996)
32. Rochart, G., Jussien, N., Laburthe, F.: Challenging explanations for global constraints. In: CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS03), pp. 31–43 (2003)
33. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier (2006)
34. Schulte, C., Stuckey, P.J.: Speeding up constraint propagation. In: Principles and Practice of Constraint Programming–CP 2004, pp. 619–633. Springer (2004)
35. Schulte, C., Stuckey, P.J.: When do bounds and domain propagation lead to the same search space? ACM Transactions on Programming Languages and Systems (TOPLAS) **27**(3), 388–425 (2005)
36. Sellmann, M., Kadioglu, S.: Dichotomic search protocols for constrained optimization. In: Principles and Practice of Constraint Programming, pp. 251–265. Springer (2008)
37. Simonis, H.: Kakuro as a constraint problem. Proc. seventh Int. Works. on Constraint Modelling and Reformulation (2008)
38. Stojadinović, M., Marić, F.: meSAT: multiple encodings of CSP to SAT. Constraints pp. 1–24 (2014)
39. Stuckey, P.J.: Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 5–9. Springer (2010)
40. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints **14**(2), 254–272 (2009)
41. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM journal on computing **1**(2), 146–160 (1972)