Symmetry Breaking in the Subgraph Isomorphism Problem

Joseph Loughney
University of St Andrews, Scotland

Ruth Hoffmann
University of St Andrews, Scotland

Mun See Chang
University of St Andrews, Scotland

Ciaran McCreesh
University of Glasgow, Scotland

Abstract

The Subgraph Isomorphism Problem (SIP) asks whether a given graph occurs within another. Despite being theoretically difficult, SIP can be solved quickly in practice. *Symmetry breaking* identifies symmetric states during the search process, and avoids searching all of them. Symmetry breaking has been shown to be useful in reducing runtimes for constraint programming, but there are many varied approaches with nuanced differences. It is unclear which symmetry breaking technique is "best" for SIP. We present various symmetry breaking techniques with a focus on influencing constraints to reflect the search order, and study the efficacy of each. Initial experimental data indicates significant speed-up over the solver with no symmetry breaking, and proposed improvements to basic techniques yield strong practical gains.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Graph algorithms, constraints programming, symmetry breaking

Digital Object Identifier 10.4230/LIPIcs.CP.2025.0

Funding *Mun See Chang*: This work was supported by the International Science Partnerships Fund (ISPF) and the UK Research and Innovation [EP/Y000609/1].

1 Introduction

The *Subgraph Isomorphism Problem* (SIP) involves finding a “copy” of a smaller graph (called a *pattern*) inside a larger graph (*target*). The problem is a generalisation of several famous problems in graph theory (such as finding a Hamiltonian cycle [10]) and as such has various applications [15, 3, 2]. SIP is NP-complete [4], but in practice can be solved very quickly. State-of-the-art subgraph isomorphism solvers, such as the *Glasgow Subgraph Solver* (GSS) [9] achieve competitive runtimes by using constraint-programming style approaches.

A technique which has seen improvements to efficiency of constraint-programming style algorithms for similar combinatorial problems is *symmetry breaking* [16, 5]. A common way of symmetry breaking is to fix an order on sets of symmetrical solutions and only searching for the ‘smallest’ in each set. This may reduce the search tree a significant amount, avoiding traversing branches that are equivalent to ones covered previously.

Our aim is to investigate how to most effectively formulate and propagate symmetry breaking constraints in SIP with the objective of increasing efficiency, including how to add constraints during search instead of before it, and whether we should do so.

Symmetry breaking in graph search problems has been studied before ([17, 16, 8, 6]), but previous attempts have been held back by overhead costs from identifying symmetries. Even so, they have found that symmetry breaking can lead to gains in a solver. Our contributions are methods which optimise the breaking of variable and value symmetries, and balance the trade-off between faster inferences and lower search node counts. Additionally, with the very efficient Dejavu library [1], we overcome the barrier of the identification overhead.

We will begin in Section 2 by framing SIP as a Constraint Satisfaction Problem, define what it means for solutions to be 'symmetric', and how we might go about breaking these symmetries by adding constraints to the solver. In Section 3 we examine *when* best to create these constraints and add them to the solver. Then in Section 4 we will see two implementations of the general techniques discussed, followed by results in Section 5.

2 SIP as a Constraint Satisfaction Problem

A *graph* $G = (V, E)$ is a set of vertices V together with a set of edges E consisting of 2-sets of V. A *subgraph isomorphism from graph* $P = (V_P, E_P)$ *to graph* $T = (V_T, E_T)$ is an injective function $\phi : V_P \to V_T$ such that edges are preserved. That is, $(u, v) \in E_P \implies (\phi(u), \phi(v)) \in E_T$. Given two graphs $P = (V_P, E_P)$ and $T = (V_T, E_T)$ (called the *pattern* and *target* graphs respectively), the *decision (enumeration, counting*, resp.) versions of SIP asks for the existence (all occurrences, the number of, resp.) of subgraph isomorphisms $\phi : V_P \to V_T$ from P to T.

We can treat subgraph isomorphism as a CSP in the following manner: pattern vertices are variables, with domains of target vertices; target vertices are values; constraints are defined by edges (such that $(u, v) \in E_P \implies (\phi(u), \phi(v)) \in E_T$) and an 'all different' constraint on the image $\phi(V_P)$ (where $\phi(u) \neq \phi(v)$ for any $u, v \in V_P$). The decision, enumeration and counting problems aim to find one, all, and the number of solutions respectively.

Gent and Smith [7] present a method they call "Symmetry Breaking During Search" and its applications to various constraint-programming-style graph search problems, which requires user input to specify problem symmetries. Puget [11] gives a method to add symmetry breaking constraints during search, which proves to be competitive with other techniques – this method uses an auxiliary CSP to calculate graph isomorphisms at each node. Zampelli et al. [17] discuss detection techniques for various symmetries in SIP along with methods of exploitation, finding that the overhead produced by calculating automorphisms on variables and values was enough to significantly diminish their gains. Yang et al. [16] present several symmetries in SIP all based on what they refer to as 'structural symmetry', which is a subset of the whole automorphism group of a graph that can be calculated easily.

Thanks to Dejavu [1], we can take a step further: we are able to automatically identify symmetries using the Schreier-Sims algorithm [13], and do so with minimal overhead. Furthermore, the set we identify describes *all* variable and value symmetries.

Variable and Value Ordering Symmetry breaking techniques partition the set of solutions into equivalence classes, and only searching for the 'smallest' solution in each class. To properly determine which solution is the smallest, we must first determine an order on the solutions. In this paper, we use $\leqslant$ and $\preccurlyeq$ to denote total orderings over V_P and V_T, respectively. We can naturally represent solutions as tuples in the following way: for a subgraph isomorphism $\phi : V_P \to V_T$, let $[[\phi]]_{\leqslant}$ be the $|V_P|$-tuple $[\phi(p_1), \phi(p_2), \cdots, \phi(p_{|V_P|})]$, where $p_i \leqslant p_{i+1}$ for all i. Representing subgraph isomorphisms as tuples gives us a means of ordering subgraph isomorphisms using lexicographical ordering: for a total ordering $\leqslant$ over the set S, the *lexicographical ordering* $\leqslant_{lex}$ *over* $\leqslant$ is a total ordering on all n-tuples over S such that $A \leqslant_{lex} B \iff a_i < b_i$ for some $1 \leqslant i \leqslant n$ and $a_j = b_j$ for all $1 \leqslant j < i$, or they are all the same. We sometimes omit the brackets for readability, for example in Example 7.

▸ **Example 1.** A solution $\phi : (a \mapsto 1, b \mapsto 4, c \mapsto 2)$ using the variable ordering a, c, b would have a tuple representation $[[\phi]]_{\leqslant} = [1, 2, 4]$. If the variable ordering was c, b, a we would have $[[\phi]]_{\leqslant} = [2, 4, 1]$. Note that the value ordering does not matter (yet).

Now if we had solutions $[1, 3, 4]$ and $[1, 2, 4]$ with a value ordering $1, 2, 3, 4$, we have that $[1, 2, 4] \preccurlyeq_{lex} [1, 3, 4]$ (since 1=1 and $2 \preccurlyeq 3$). If instead the value order was $4, 1, 3, 2$, we would have $[1, 3, 4] \preccurlyeq_{lex} [1, 2, 4]$.

Symmetries in SIP An *automorphism* of a graph $G = (V_G, E_G)$ is a permutation $\alpha : V_G \to V_G$ of the vertex set that preserves the edge set E_G. The set of all automorphisms of a graph G forms an algebraic structure called the *automorphism group of* G and denoted $\mathrm{Aut}(G)$. We write permutations using the cycle notation.

Let P, T be graphs, $g \in \mathrm{Aut}(P)$ and $h \in \mathrm{Aut}(T)$ and $\phi : V_P \to V_T$ be a subgraph isomorphism from P to T. Denote by ϕ^g and ϕ^h the compositions $g \circ \phi$ and $\phi \circ h$ respectively. Then ϕ^g and ϕ^h are subgraph isomorphisms from P to T as well. This now gives a notion of symmetry. More specifically, we say that two subgraph isomorphisms ϕ and ϕ' from P to T are *symmetrical* if there exists $g \in \mathrm{Aut}(P)$ and $h \in \mathrm{Aut}(T)$ such that $\phi' = (\phi^g)^h$.

A common way of completely breaking symmetries in CSP is by using the lex-leader constraint, which takes the form $X \leqslant X^s$, $\forall s \in \mathcal{S}$, where X is the decision variable, $\mathcal{S}$ is the set of all symmetries and X^s denotes the image of a value of X under the symmetry s. We may replace $\mathcal{S}$ with a subset to obtain a sound but incomplete symmetry breaking constraint (see [12] for more details). For SIP, we take $\mathcal{S}$ as the set $\{g \circ h : g \in \mathrm{Aut}(P), h \in \mathrm{Aut}(T)\}$ consisting of all combinations of pattern and target graph automorphisms. Therefore, in general, our symmetry breaking constraint takes the following form, where $S \subseteq \mathcal{S}$:

$$[[\phi]]_{\leqslant} \preccurlyeq_{lex} [[\phi^g]]_{\leqslant}, \ \forall \ g \in S. \tag{1}$$

▸ **Example 2.** Let ϕ be the assignment $(a \mapsto 0, b \mapsto 1, c \mapsto 2)$ found at some point during search, and let us use the orderings $[a, c, b]$ and $[0, 1, 2]$ for $\leqslant$ and $\preccurlyeq$ respectively; then $[[\phi]]_{\leqslant} = [0, 2, 1]$. Let $g = (b \ c)$ be an automorphism; then $[[\phi^g]]_{\leqslant} = [0, 1, 2]$. Since $[[\phi^g]]_{\leqslant} \prec_{lex} [[\phi]]_{\leqslant}$, an algorithm with symmetry breaking should reject ϕ.

In Equation (1), there are 3 things we may vary: (a) $\leqslant$, the variable ordering; (b) $\preccurlyeq$, the value ordering; and (c) the subset of the symmetry group. We now have a choice of seemingly independent things to vary when breaking symmetries in the problem. It is unlikely that one variation will be the optimal choice in all cases. Can we say why some choices are better for a given problem? Can we decide when is best to use each variation?

In the next section, we will see how we may theoretically vary the first two points, and whether they interact with each other; the following section will look at practical implementations of this theory; and Section 5 will examine some empirical evidence supporting the use of the various techniques. We will not examine the third point in this paper; this is left as future work.

3 Varying the Variable and Value Ordering

The types of variable and value ordering we may use are the *fixed* ordering, the *flexible* ordering and the *dynamic* ordering. We shall illustrate these ideas using examples of finding pattern graph P (Figure 1) in target graph T (Figure 2). Note that the examples use only variable ordering, but the concepts are identical for value ordering. We shall take the natural ordering of integers as the our value ordering.

Assume also that, for the purposes of illustration, we perform no inferences on symmetry breaking, and only naïve accept-or-reject conditions when two symmetric vertices are both assigned. In practice, we can make inferences much earlier on to significantly reduce the size of the search tree.

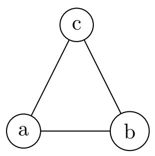

Figure 1 A pattern graph P.

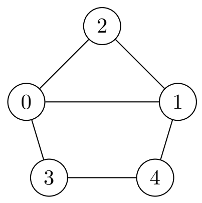

Figure 2 A target graph T.

▸ **Example 3** (Fixed – Predefined Static). We pick a variable and value ordering before search. Suppose our predefined static ordering for the case Figure 1 $\mapsto$ Figure 2 found a and c to be symmetric, and gave a variable order of $c < b < a$. Then the search tree looks as the one in Figure 3.

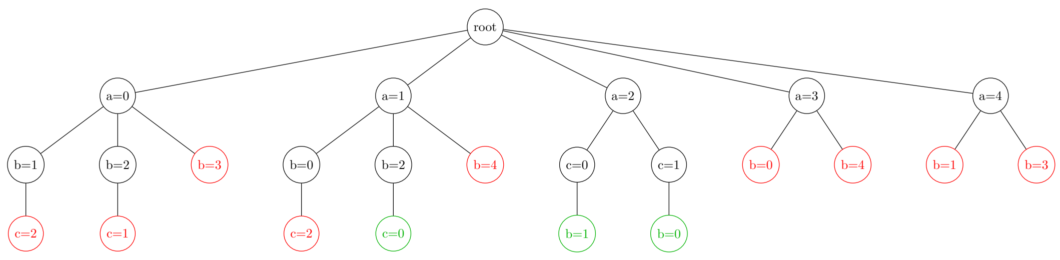

Figure 3 The search tree with symmetry breaking with respect to a predefined static variable ordering of $c < b < a$. Green nodes are accepted solutions, red nodes are rejected states.

As we see, using the given variable order with a and c symmetric rejects the otherwise accepted solutions $[2, 1, 0], [2, 0, 1], [1, 2, 0]$ but still accepts $[0, 1, 2], [0, 2, 1], [1, 0, 2]$. Note that even though b is in the variable order, it is not found to be symmetric with anything and so does not have an effect on which solutions are kept or rejected.

▸ **Example 4** (Flexible – Static During Search). In this version, we decide on an ordering only as needed, corresponding to the variable/value order first picked by the solver and remaining fixed once fully ordered. Note in particular that unlike the previous method, we do not choose a variable order before search, though we still identify a and c as symmetrical. The search tree now looks as in Figure 4.
Figure 4 shows that, with the variable ordering $a < b < c$, the solutions $[2, 1, 0], [2, 0, 1], [1, 2, 0]$ are rejected, and $[0, 1, 2], [0, 2, 1], [1, 0, 2]$ accepted. This strategy favours the leftmost branch of the search tree, i.e. the one traversed first, and makes it preferable for the decision problem.

We may note that both Example 3 and Example 4 use a fixed ordering once fully constructed in either case, the key distinction being that predefined ordering does not take into account the variable order that the solver might use.

▸ **Example 5** (Dynamic). Finally, consider the case with dynamic variable ordering, where the ordering at any search node corresponds directly to the variables assigned at that node.

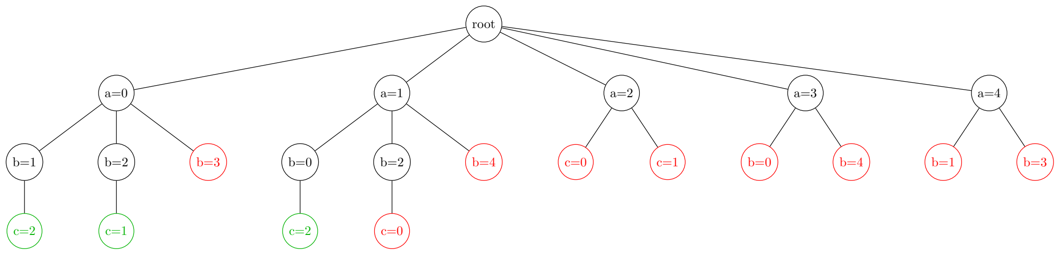

Figure 4 The search tree with symmetry breaking with respect to a static variable ordering constructed during search. In this case we visit a then b then c, so the ordering is $a < b < c$.

If the problem in Example 4 had used dynamic symmetry breaking, the only point at which the order would be different is at the subtree $a \mapsto 2$, where the order is $a < c$ rather than $a < b$. Not only that, but the search trees would look identical! Dynamic symmetry breaking is strong and breaks many symmetries, but may be slower to compute overall.

4 Symmetry Breaking Techniques

We have so far decided that we may define solutions (and non-solutions) to be symmetrical if there exists an automorphism, or composition of automorphisms, mapping one onto the other. It remains to decide how best to describe the automorphisms with constraints. Presented in this section are two methods of doing so.

4.1 Using Orbits for Symmetry Breaking

For a vertex v in a graph G, its *orbit* $\mathrm{orb}(v)$ is the set $\{v^\alpha \mid \alpha \in \mathrm{Aut}(G)\}$ consisting of all images of v under automorphisms of G. For example, for the graph in figure 2, $\mathrm{orb}(1) = \{0, 1\}, \mathrm{orb}(2) = \{2\}$ and $\mathrm{orb}(3) = \{3, 4\}$.

Using vertex orbits of the pattern and target graphs, we can generate constraints to break symmetries. The constraints are added to the solver in the following forms:

- For **variable** (or *pattern*) symmetry, $a < b$ means that a appears before b in our variable order, and therefore appears before b in our tuples in (1).
- For **value** (or *target*) symmetry, $a \prec b$ means that a appears before b in the value order, and therefore $[a, \cdots] \preccurlyeq_{lex} [b, \cdots]$.

We can make inferences about these constraints which allow us to refine domains with forward checking and arc consistency. For pattern constraints, for example, we may reason that the constraint $p < q$ (where p and q are symmetric) implies $\phi(p) \preccurlyeq \phi(q)$. Hence using the constraint $p < q$ we can infer two things about the domains of p and q, even before they are assigned:

- the smallest value (with respect to $\prec$) of the domain of q cannot be less than or equal to the smallest value of the domain of p.
- the largest value (with respect to $\prec$) of the domain of p cannot be greater than or equal to the largest value in the domain of q.

We have implemented an algorithm into the solver which removes values which do not satisfy the above requirements from variable domains.

For target constraints we may reach similar conclusions, with one notable exception: SIP is non-surjective, so not all values need be assigned. We therefore reach the following inferences from a constraint $t \prec u$:

- if $\phi(p) = t$, u cannot be used for any variable $q < p$.
- if $\phi(p) = u$, we must have $\phi(q) = t$ for some variable $q < p$.
- if t is not assigned, u cannot be assigned either.

Note here that since we only use a single cycle from each permutation, we cannot derive the complete ordering in either case. As such, we cannot use variable and value ordering simultaneously in case they conflict.

4.2 Using Permutations for Symmetry Breaking

When using orbits, we lose some information about symmetries of other vertices – our running example, we could only consider $\{0, 1\}$ or $\{3, 4\}$ since fixing any vertex also fixes the rest. Because of this, we cannot use variable and value symmetries at the same time.

We may avoid this, then, with some technique which uses all cycles from a permutation at once. For a permutation $(p\ q)(r\ s)$, rather than adding binary constraints to the solver like the previous $p < q$, we instead add the permutation itself, stored in a map object.

▸ **Example 6** (A simple case). Let's return to our running example and examine the search tree with fixed-order permutation-based value symmetry breaking.

This first step is to determine the value order. We find the group $\mathrm{Aut}(T)$ contains only $()$ and $(0\ 1)(3\ 4)$, which allows an implicit value ordering of either $[0, 1, 2, 3, 4]$ or $[1, 0, 2, 4, 3]$. Without loss of generality, we select the latter. Then the search tree looks like this:

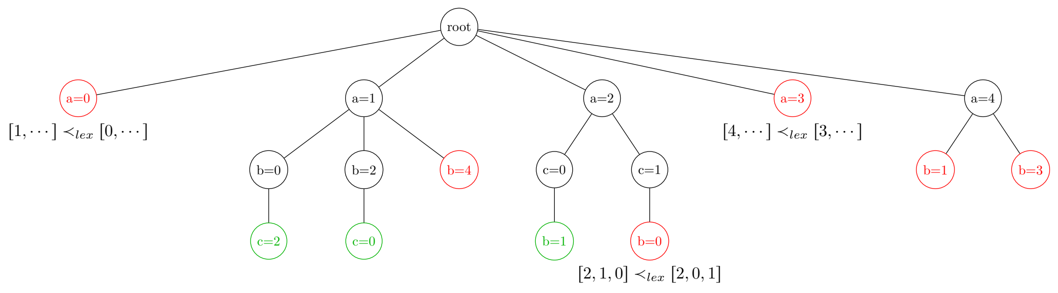
root
a=0
a=1
a=2
a=3
a=4
$[1, \cdots] \prec_{lex} [0, \cdots]$
$[4, \cdots] \prec_{lex} [3, \cdots]$
b=0
b=2
b=4
c=0
c=1
b=1
b=3
c=2
c=0
b=1
b=0
$[2, 1, 0] \prec_{lex} [2, 0, 1]$

Figure 5 The search tree with symmetry breaking with respect to a predefined static value ordering of $1 \prec 0 \prec 2 \prec 4 \prec 3$, and the permutation $(0\ 1)(3\ 4)$. Green nodes are accepted solutions, red nodes are rejected states.

As we can see, this technique allows both the $a \mapsto 0$ and $a \mapsto 3$ subtrees to be filtered, whereas if we were to use orbits we could only filter one.

Now, unlike with the previous technique, we know the complete total orderings, and hence can use both variable and value ordering simultaneously. This approach has also been implemented into the Glasgow Subgraph Solver, but has been omitted from this paper for space reasons.

Here we compute a permutation the current assignment at each search node. We then examine the variables in lexicographical order; if either is unassigned, we cannot say whether the permutation or the current assignment is lexicographically smallest, and so we break and continue the search. On the other hand, if both are assigned, we know either to check the next vertex or either accept or reject.

An important step in this algorithm is domain refinement of unassigned variables. Working backwards, we start with "if the (unassigned) variable v^p were assigned to a value x such that $x^t \prec \phi(v)$, we would have a smaller solution and would thus reject that branch". Using this logic, we can reject that branch ahead of time by removing x from the domain of v^p. This technique is illustrated in Example 7.

▸ **Example 7.** Let $\phi = [1, 2, 5, \_, \_]$, $p = (c\ d)$ and $t = (3\ 4\ 5\ 6)$, where $\_$ denotes an unassigned value. Then $\phi^{pt} = [1, 2, \_, 6, \_]$ (we exchange the variables c and d, then permute $5 \mapsto 6$). Then since $c^{p^{-1}} = d$ and $6^t \prec 5, 3^t \prec 5$, we remove 3 and 6 from the domain of d.

We can verify: $[1^t, 2^t, 5^t, 3^t, \_]^p = [1, 2, 4, 6, \_] \prec_{lex} [1, 2, 5, 3, \_]$ and $[1^t, 2^t, 5^t, 6^t, \_]^p = [1, 2, 3, 6, \_] \prec_{lex} [1, 2, 5, 6, \_]$ – both would be permuted to a 'smaller' solution.

5 Experimental Results

Shown below are the runtimes of the Glasgow Subgraph Solver with various symmetry breaking techniques applied against the runtimes with no symmetry breaking. The dataset used is the graph benchmark set found at [14], which contains a large number of problem instances from a combination of real-world and theoretical sources. The experiments shown were run with a timeout of 30000ms, or 5 minutes, including the time taken to detect symmetries.

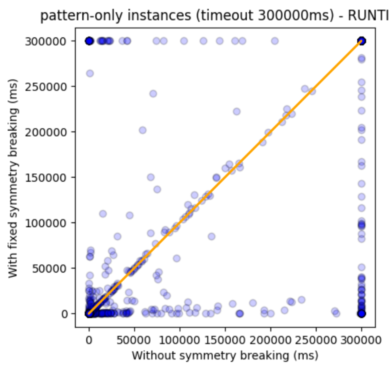

Figure 6 Fixed order orbit variable symmetry breaking vs. no symmetry breaking, runtime.

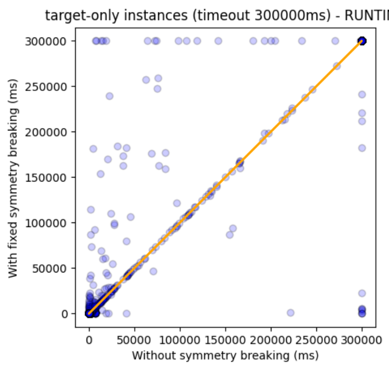

Figure 7 Fixed order orbit value symmetry breaking vs. no symmetry breaking, runtime.

The fixed-order orbit symmetry breaking shown in Figure 6 and Figure 7 gives us several insights – the orange line runs through $x = y$, so points below the line represent cases where symmetry breaking has sped up the solver, and cases above the line have been slowed down by symmetry breaking. The fact that some cases are slowed down may be unexpected! Unfortunately, while the number of search nodes is reduced, this comes at the cost of more expensive propagations at each remaining node, which leads to a slower time overall. As a rough estimate, we should only expect a speed-up when $\frac{\text{\% reduction in search nodes}}{\text{\% increase in time per node}} > 1$.

Moreover, it seems that variable symmetry breaking is significantly more effective than value symmetry breaking; while only a small number of value cases show a significant speed-up, large numbers of variable symmetry breaking cases are clustered at the lower right corner – where the solver had previously timed out, it now solves the problem very quickly.

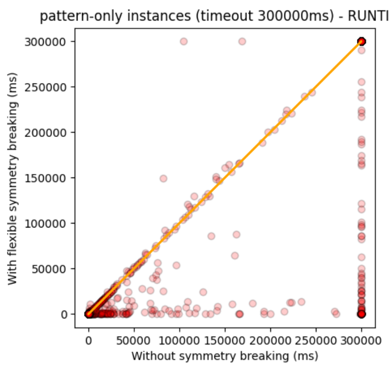

Figure 8 Flexible order orbit variable symmetry breaking vs. no symmetry breaking, runtime.

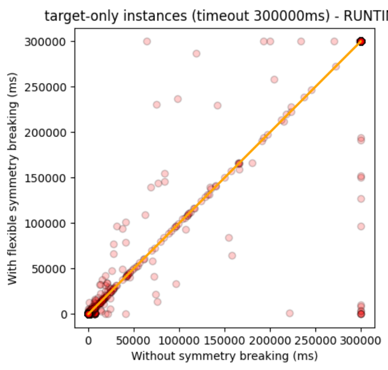

Figure 9 Flexible order orbit value symmetry breaking vs. no symmetry breaking, runtime.

The data in Figure 8 and Figure 9 shows that allowing for a flexible variable or value ordering yield significant improvements over the arbitrary static ordering. While variable still vastly outperforms value, both outperform their fixed-order counterparts.

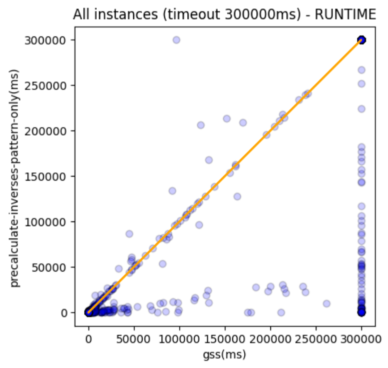

Figure 10 Fixed order permutation-based variable symmetry breaking vs. no symmetry breaking, runtime.

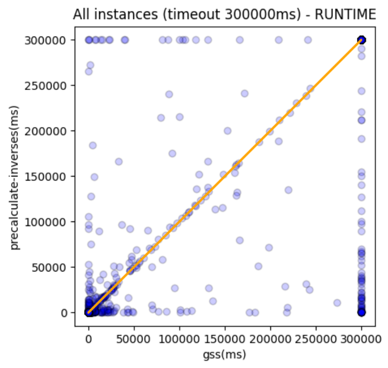

Figure 11 Fixed order permutation-based variable and value symmetry breaking vs. no symmetry breaking, runtime.

Figure 10, fixed order permutation-based symmetries, reflects a similar distribution to the flexible-order variable orbit symmetry breaking. This is somewhat unexpected – it seems that the additional information retained about symmetries by using permutations with a fixed variable ordering is enough to match the advantage of using a flexible variable ordering with orbits. The flexible order technique is not yet implemented for permutation-based symmetry breaking; once it is, another dataset will be generated which will make drawing more conclusions easier.

Figure 11 introduces a great variance into the result set. This indicates that using variable and value symmetries in this method, though still more effective than the original solver on average, is less efficient than only using variable symmetries. Similar conclusions are drawn from value-only, omitted for brevity.

6 Conclusion

We have introduced a universal equation for symmetry breaking in SIP, and identified three key elements that can be varied independently of each other within this formula. We have discussed what it means to change a variable or value order, and what this looks like in terms of both solutions and search trees.

We have presented two techniques for describing symmetries within a constraints-style solver, and shown that both can provide significant improvements in average runtime even without variable/value-order influence. Further, we have shown that altering these orderings to be closer to the search order can yield even greater improvements to the solver.

Future work involves: investigating the effect of varying S, the set of permutations used for symmetry breaking; examining the effect of fully dynamic ordering on orbit-based symmetry breaking; investigating flexible and dynamic ordering in permutation-based symmetry breaking; and applying these generalised techniques to other graph search problems.

References

1 Markus Anders and Pascal Schweitzer. Dejavu Github repository. `https://github.com/markusa4/dejavu`, 2023.

2 Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14:1–13, 2013.

3 Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

4 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. `doi:10.1145/800157.805047`.

5 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for sat. In *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19*, pages 104–122. Springer, 2016.

6 Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In *Principles and Practice of Constraint Programming—CP 2001: 7th International Conference, CP 2001 Paphos, Cyprus, November 26–December 1, 2001 Proceedings 7*, pages 93–107. Springer, 2001.

7 Ian P Gent and Barbara Smith. *Symmetry breaking during search in constraint programming.* Citeseer, 1999.

8 Marijn JH Heule. Optimal symmetry breaking for graph problems. *Mathematics in Computer Science*, 13:533–548, 2019.

9 Ciaran McCreesh et al. GSS Github repository. `https://github.com/ciaranm/glasgow-subgraph-solver`, 2024.

10 Ivan Olmos, Jesus A Gonzalez, and Mauricio Osorio. Reductions between the subgraph isomorphism problem and hamiltonian and sat problems. In *17th International Conference on Electronics, Communications and Computers (CONIELECOMP'07)*, pages 20–20. IEEE, 2007.

11 Jean-Francois Puget. Symmetry breaking using stabilizers. In *International Conference on Principles and Practice of Constraint Programming*, pages 585–599. Springer, 2003.

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

13 Charles C. Sims. Computational methods in the study of permutation groups††this research was supported in part by the national science foundation. In JOHN LEECH, editor, *Computational Problems in Abstract Algebra*, pages 169–183. Pergamon, 1970. URL:

https://www.sciencedirect.com/science/article/pii/B9780080129754500205, doi:10.1016/B978-0-08-012975-4.50020-5.

14 Christine Solnon. Benchmarks for the subgraph isomorphism problem. https://perso.liris.cnrs.fr/christine.solnon/SIP.html.

15 Christine Solnon, Guillaume Damiand, Colin De La Higuera, and Jean-Christophe Janodet. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognition*, 48(2):302–316, 2015.

16 Dominic Yang, Yurun Ge, Thien Nguyen, Denali Molitor, Jacob D Moorman, and Andrea L Bertozzi. Structural equivalence in subgraph matching. *IEEE Transactions on Network Science and Engineering*, 2023.

17 Stéphane Zampelli, Yves Deville, Mohamed Réda Saıdi, and Belaıd Benhamou. Symmetry breaking in subgraph isomorphism. In *Proc. SymCon*, volume 7. Citeseer, 2007.