Efficiently Solving Problems Where the Solutions Form a Group

Karen E. Petrie and Christopher Jefferson*

Computing Laboratory, University of Oxford, UK
karen.petrie@comlab.ox.ac.uk, chris.jefferson@comlab.ox.ac.uk

Abstract. Group theory is the mathematical study of symmetry. This paper presents a CP method of efficiently solving group-theoretic problems, where each of the solutions is an element of a group. This method allows us to answer questions in group theory which are computationally unfeasible with traditional CP techniques.

1 Introduction

Many problems arising in group theory can be naturally expressed as constraint problems but current solvers are often unable to solve instances of an interesting size. Our aim is to create a constraint programming based tool for mathematicians, that allows group theorists to search for groups with a specific property. It will allow counter example generation by answering for example: "Does a group exist with a given subgroup, and a given element of a certain order".

This paper provides the fundamental basis for such a system; by providing a constraint programming method for solving group-theoretic problems, where each of the solutions is an element of a group. This algorithm works by finding only a small subset of solutions which are sufficient to generate every other solution. As we will see our method allows group-theoretic problems to be solved which can not be solved using traditional constraint techniques.

2 Overview of Method

In this section we will briefly define a number of common group-theoretic concepts, for a more complete introduction see [1]. Stabiliser chains provide an algorithmic method of constructing a small generating set [2] for any group and provide the inspiration for our algorithm. The stabiliser chain relies on the concept of the point wise stabiliser. We start by giving the definition of a stabiliser.

Definition 1. *Let G be a permutation group acting on the set of points Ω. Let $\beta \in \Omega$ be any point. The* stabiliser *of β is the subgroup of G defined by: $Stab_G(\beta) = \{g \in G \mid \beta^g = \beta\}$, which is the set of elements in G which fixes or*

* Chris Jefferson was supported by EPSRC Grant EP/D032636/1 and Karen Petrie by a Royal Society Fellowship. We would like to thank the referees for their comments.

P.J. Stuckey (Ed.): CP 2008, LNCS 5202, pp. 529–533, 2008.

stabilises *the point β. The stabiliser of any point in a group G is a subgroup of G. The stabiliser of a set of points, denoted $Stab_G(i, j, \dots)$, is the elements of G which move none of the points.*

The definition of the stabiliser chain follows.

Definition 2. *Stabiliser chains are built in an recursive fashion. Given a permutation group G and a point p, the first level of the stabiliser chain is built from an element of G which represents each of the places p can be mapped to. The next level of the stabiliser chain is built from applying this same algorithm to $Stab_G(p)$, again choosing representative elements for all the places some point $q \neq p$ can be mapped to. The stabiliser chain is finished when the stabiliser generated contains only the identity element.*

Stabiliser chains, in general, collapse quickly to the subgroup containing only the identity since the order of each new stabiliser must divide the order of the stabilisers above it. The following example is given to crystallise the stabiliser chain concept.

Example 1. Consider the symmetric group consisting of the 24 permutations of $\{1, 2, 3, 4\}$. We compute a chain of stabilisers of each point, starting arbitrarily with 1 (denoted $Stab_{S_4}(1)$). 1 can be mapped to 2 by $[2, 1, 3, 4]$, 3 by $[3, 1, 2, 4]$ and 4 by $[4, 1, 2, 3]$. These group elements form the first level of the stabiliser chain.

The second level is generated by looking at the orbit and stabiliser of 2 in $Stab_{S_4}(1)$. In the stabiliser of 1, 2 can be mapped to both 3 and 4 by the group elements $[1, 3, 2, 4]$ and $[1, 4, 2, 3]$. We now stabilise both 1 and 2, leaving only the group elements $[1, 2, 3, 4]$ and $[1, 2, 4, 3]$. Here 3 can be mapped to 4 by the second group element, and once 1, 2 and 3 are all stabilised the only element left is the identity and the algorithm finishes.

The stabiliser chain shows how a generating set of elements can be generated from a limited number of simple calculations. Our method is based around splitting search into a number of pieces, and finding only the first solution in each of these pieces. Each of these pieces is equivalent to a step in the stabiliser chain. We state without proof due to space restrictions that an arbitrary solution, should one exist, to each member of the split we define, form a stabiliser chain, and therefore a set of generators for the group of solutions. The precise split we use is given in Definition 3.

Definition 3. *Given a CSP P where the projection of the solutions onto some list of variables V of length n forms a permutation group, then the* ***generator split*** *of P is the following set of CSPs, each equal to P with a list of additional constraints: $\forall 1 \leq i < j \leq n$. $P_{i,j} = P \wedge (\forall 1 \leq k \leq i - 1 . V_k = k) \wedge V_i = j$.*

Our algorithm is very simple. It requires creating a generator split of a CSP and then finding the first solution, if one exists, to each of the CSPs in the generator split by whichever means we wish. The major strength of this algorithm is that it can be implemented with no changes to the constraint solver. This does however

create a small overhead due to having to start the solver many times. In our Minion implementation, we instead create the CSP once in the solver, and then solve it multiple times. This is possible as the extra conditions imposed by the generator split are only variable assignments. No other changes were necessary to implement the algorithm.

3 Experimental Results

We consider a number of experiments, each of which involves solving a CSP whose solutions form a group. These will show that the gains made from identifying that the solutions to a problem form a group often provides massive advantages, and almost no loss in even the worst case.

3.1 Graph Automorphism

Probably the most famous problem whose solutions form a group is graph automorphism. Our model does not use the propagators given in [3] due to lack of an implementation in the Minion constraint solver.

We will consider finding the symmetries of two families of graphs, randomly generated graphs and the grid graph, given in Definition 4.

Definition 4. *The $l \times w$ **grid** graph is a graph on $l \times w$ vertices arranged in a grid of height l and width w, where two vertices are connected by an edge if they are either in the same row or same column.*

The symmetry group of the grid graph arises frequently in constraint programming, as this is the symmetry group of problems with "row and column" matrix symmetry [4], a commonly occurring group in constraint programming. Therefore being able to quickly identify this group would be an important and useful property for any system which would be used to identify the graph automorphisms which occur in CP.

Table 1 shows a comparison of our algorithm against a traditional complete search for identifying the automorphism group of grid graphs of various sizes. It is clear from these results that using a traditional search quickly becomes unfeasible. Using the generators found by our algorithm, a computational group theory package such as GAP can almost instantly produce the total size of the group, which we fill in for the two largest instances. Clearly no constraint solver could enumerate this many solutions.

Note that while our algorithm takes a non-trivial period of time for large graphs, the size of the search is still very small. Given a more efficient propagator for graph automorphism, we expect these times would drop dramatically.

We also conducted experiments to compares finding the symmetries of a small selection of random graphs. In general we expect such graphs to have no symmetries except for the identity symmetry, and indeed all the graphs we considered did only have this symmetry. As these graphs have no symmetries, we do not expect our algorithm to perform any better than a complete search. The aim therefore of these experiments is to investigate the overhead which is introduced.

Table 1. Comparing algorithms for finding the automorphisms of a grid graph

Size	Traditional			New		
	Solutions	Nodes	Time	Generators	Nodes	Time
3×3	72	143	0.007	12	31	0.07
4×4	1,152	2303	0.26	24	103	0.08
5×5	28,800	57599	9.64	40	238	0.12
6×6	1,036,800	2073599	711.8	60	455	0.31
10×10	2.6×10^{13}	-	-	180	2523	11.61
15×15	3.4×10^{24}	-	-	420	9233	297.6

The results show that there is almost no measurable overhead introduced by our algorithm.

For a static variable ordering along the permutation, we expect the searches produced with and without our algorithm to be almost identical, except for a tiny variance in the number of search nodes introduced from splitting the search into pieces before beginning and this is what we see in practice. We also conducted experiments using a dynamic variable ordering, smallest domain first. While this does introduce some measurable differences into the resulting searches, it is not clear if our algorithm is better or worse, and once again any variance is small. While this by no means proves our algorithm would not interfere with any dynamic variable ordering, it produces promising evidence that it does not effect search even when dynamic heuristics are used.

One important step we have not taken here is comparing our algorithm against specialised graph isomorphism systems, such as those provided in specialise graph isomorphism tools such as NAUTY [5] and SAUCY [6]. We feel for a fair comparison our algorithm must first be combined with a specialised propagator. We note that the experiments in [3] show a specialised propagator can find single automorphisms very competitively, giving hope that combined with our new algorithm the result should be comparable to these systems, while allowing a much greater degree of flexibility.

3.2 Group Intersection

One of the major advantages of designing our algorithm as a modification to search in a traditional CP framework is that allows us to use the flexibility of CP when modelling our problems. As an example of this flexibility, we consider finding the intersection of the grid graph, given in Definition 4, with the alternating group, given in Definition 5. Expressing this as a CSP requires simply imposing the constraints for both in the same problem. Definition 5 does not provide an obvious method of expressing that a permutation is even. A well known alternative method of checking if a permutation V is even is to check if the value of the expression $\sum_{1 \leq i < j \leq n} (V[i] > V[j])$ is even. This is the formulation which we use to express that a permutation is alternating.

Definition 5. *A permutation is even if it can be expressed as an even number of swaps of pairs of values. The alternating group contains even permutations.*

Table 2. Comparing algorithms for finding the intersection of the grid graph and alternating group

Size	Traditional			New		
	Solutions	Nodes	Time	Generators	Nodes	Time
3×3	36	107	0.02	11	30	0.04
4×4	1,152	2,303	0.81	24	103	0.06
5×5	14,400	43,199	7.5	39	237	0.13
6×6	518,400	1,555,199	509.4	55	274,010	109
7×7	25,401,600	76,204,799	40304	83	772	1.46

Table 2 gives results for this experiment. The results show how the power of constraint programming can be used to solve complex problems. It is unclear from where the large number of nodes for the 6×6 grid arise and this shows that it is non-obvious how hard it will be to find the intersection of two groups. Our algorithm still noticeably outperforms complete search in this instance and performs magnitudes better on the largest instance.

4 Conclusion

We have extended the abilities of constraint programming to allow problems in Computational Group Theory to be efficiently solved. We have, moreover, demonstrated experimentally that constraint programming can be a useful tool to solve group theoretic problems. Our method, allows us to solve problems which would not be possible without this constraint based decomposition technique. This result is important, since it shows the scope for constraint programming to be applied to group theoretic research.

References

1. Armstrong, M.A.: Groups and Symmetry. Springer, Heidelberg (2000)
2. Jerrum, M.: A compact presentation for permutation groups. J. Algorithms 7, 71–90 (2002)
3. Sorlin, S., Solnon, C.: A parametric filtering algorithm for the graph isomorphism problem. Journal of constraints (December 2008)
4. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 462–476. Springer, Heidelberg (2002)
5. McKay, B.: Practical graph isomorphism. In: Proc. 10th Manitoba Conf. on Numerical mathematics and computing, Winnipeg/Manitoba 1980, Congr. Numerantium, vol. 30, pp. 45–87 (1981), `http://cs.anu.edu.au/people/bdm/nauty`
6. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for cnf. In: DAC 2004: Proceedings of the 41st annual conference on Design automation, pp. 530–534. ACM, New York (2004)