# The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants

Ciaran McCreesh(✉), Patrick Prosser, and James Trimble

University of Glasgow, Glasgow, Scotland
ciaran.mccreesh@glasgow.ac.uk

**Abstract.** The Glasgow Subgraph Solver provides an implementation of state of the art algorithms for subgraph isomorphism problems. It combines constraint programming concepts with a variety of strong but fast domain-specific search and inference techniques, and is suitable for use on a wide range of graphs, including many that are found to be computationally hard by other solvers. It can also be equipped with side constraints, and can easily be adapted to solve other subgraph matching problem variants. We outline its key features from the view of both users and algorithm developers, and discuss future directions.

## 1 Introduction

The subgraph isomorphism family of problems involves finding a small "pattern" graph inside a larger "target" graph, or establishing that the pattern does not occur. When the pattern graph is part of the input, these problems are NP-complete; despite this, subgraph isomorphism algorithms are widely used in practice, including for model checking [23], for law enforcement [9], in biological applications [1,6,20], for compiler implementation [5], in designing mechanical locks [27], and inside graph databases [19]. This has encouraged the development of practical subgraph isomorphism algorithms, which fall into two categories: those based upon backtracking and connectivity [6–8], and those based upon constraint programming [3,4,15,18,25]. Presently, the constraint programming approaches give spectacularly better performance on hard instances [19,26], although simple backtrackers will often (but inconsistently) run faster on some very easy instances due to lower overheads and faster startup costs.

This paper gives an overview of the Glasgow Subgraph Solver, which is the current state of the art in subgraph solving for hard instances [26]. First, we will discuss the range of subgraph isomorphism problems that people sometimes wish to solve, and then describe the main techniques the solver uses to solve these problems. We finish with a list of potential future directions.


F. Gadducci and T. Kehrer (Eds.): ICGT 2020, LNCS 12150, pp. 316–324, 2020.
https://doi.org/10.1007/978-3-030-51372-6_19

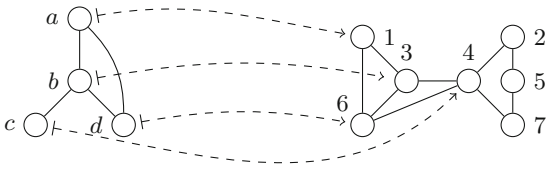


**Fig. 1.** The arrows show a non-induced subgraph isomorphism from the pattern graph on the left to the target graph on the right. This subgraph isomorphism is not induced, due to the extra edge between vertices 4 and 6 when $c$ and $d$ are not adjacent.

## 2 Subgraph Isomorphism Problems and Variants

Figure 1 illustrates a basic subgraph isomorphism problem: we have a small *pattern* graph and a large *target* graph (both of which are inputs to the problem), and we wish to decide whether the pattern graph occurs inside the target graph. Usually this is expressed in terms of finding a mapping from the vertices of the pattern graph to the vertices of the target graph, as shown using the dotted arrows. Beyond this, different applications have different views of what exactly the problem to be solved is—we therefore give a brief overview of the common problem variants.

*Adjacency, Loops, and Directed Edges.* It is generally agreed that for a mapping to be a valid subgraph isomorphism, adjacent vertices must be mapped to adjacent vertices. However, authors (particularly in application-oriented papers) disagree over whether non-adjacent vertices must be mapped to non-adjacent vertices. We use the term *induced* if non-adjacency must be preserved, and *non-induced* otherwise; when we do not qualify our terms, we are talking about both variants. A further question is on how to handle loops (that is, vertices which are adjacent to themselves). We take the view that loops may only be mapped to loops, and for induced problems, additionally that non-loops may only be mapped to non-loops; some other solver authors disagree or have not considered this question, and may handle this differently. Finally, in the case of graphs with directed edges (which could potentially go in both directions), we treat non-induced as meaning "the edges mapped to must be equal to or be a superset of the pattern edges", and induced as meaning "exactly equal to".

*Vertices and Injectivity.* In the classical subgraph isomorphism problem, the mapping is required to be injective—that is, each pattern vertex must be mapped to a different target vertex. In some applications this restriction can be relaxed: for example, we may prefer local injectivity (no two vertices that share a neighbour are mapped to the same vertex) [12], or even to find a homomorphism, where there are no injectivity requirements at all.

*Labels.* In some applications, either vertices, edges or both have labels, and may only be mapped to vertices or edges with matching labels—for example, in

chemistry problems, labels may represent different atoms in a molecule, and we may not map a carbon atom to a hydrogen atom. Richer labelling rules may be necessary in some applications, such as in temporal graphs when we care about "before/after" labels rather than exact matches [21].

*Deciding, Enumerating, and Counting.* Instead of simply asking whether a subgraph isomorphism exists, some applications want to find all such mappings. They may require that these be explicitly enumerated, but sometimes a count is sufficient—and counting can be exponentially faster than enumerating in some situations. A further complication is that the number of mappings and the number of *images* of mappings are not the same, and different applications assume different definitions—sometimes, the number of mappings are called "labelled" countings, whilst the number of images of mappings are called "unlabelled".

*Performance.* Finally, we briefly discuss the common misconception that subgraph isomorphism being NP-complete somehow means that it is not viable to solve the problem in practice, or that every instance will exhibit exponential complexity. In fact, with good algorithms, instances that are actually hard to solve in practice are rare. We caution that benchmarking algorithms for NP-complete problems is challenging, that the size of the inputs is not an indicator of difficulty, and that only comparing performance on a few easy instances can lead to design flaws in applications built on top of these algorithms [19,26].

## 3 The Glasgow Subgraph Solver

The Glasgow Subgraph Solver provides a high quality implementation of algorithms for many subgraph isomorphism problem variants. It is open source software, released under the MIT licence (which allows for commercial and closed source reuse). It may be downloaded from https://github.com/ciaranm/glasgow-subgraph-solver. It is implemented in C++, using the Boost libraries. It supports a variety of input file formats, but given the subtle and often undocumented differences in meanings of inputs in supposedly common file formats (e.g. whether edges are explicitly listed in both directions for undirected graphs), the solver has been designed to make it easy to add new parsers. The solver is primarily intended to be run from the command line or as a separate process, and its output is easy to parse for use with other tools.

### 3.1 Algorithmic Details

The Glasgow Subgraph Solver is based upon ideas from constraint programming. In a general constraint programming problem, we have a set of *variables*, each of which has a *domain* of possible *values*. We also have a set of *constraints*, which restrict valid combinations of values for subsets of the variables. The goal is to give each variable a value from its domain, respecting all constraints; usually this is done using a combination of inference and intelligent backtracking search.

To model subgraph isomorphism using constraint programming, we have a variable for each pattern vertex, and the domains are all of the target vertices. The constraints depend upon the exact variant being modelled, but we will usually have one constraint to deal with injectivity, and then a set of constraints to deal with edge and adjacency rules. A key strength of constraint programming is in the ability to add additional *implied* constraints, which we will now discuss—these can vastly speed up the solving process.

*Degree Filtering.* In an injective mapping, it is easy to see that a pattern vertex of degree $d$ can never be mapped to a target vertex of degree less than $d$. This often allows many values to be eliminated from domains before any search starts. The solver uses even stronger filtering, based upon a result by Zampelli et al. [28], which looks at the neighbourhood degree sequence of vertices.

*Distances and Paths.* Another source of additional constraints comes from reasoning about distances or paths, rather than just adjacency. Audemard et al. [4] observed that the fact that subgraph isomorphisms preserve or reduce distances can be used to provide additional filtering during search. An early precursor to the Glasgow Subgraph Solver [18] strengthened this result, using instead the fact that subgraph isomorphisms preserve paths: if there are exactly $k$ paths of length exactly $\ell$ between two vertices in a pattern graph, then there must be at least $k$ paths of length exactly $\ell$ between wherever these two vertices are mapped in the target graph. This is exploited through the use of supplemental graphs, as follows.

We define a *supplemental graph* to be a graph with two distinguished vertices, that is subgraph isomorphic to itself under the interchange of these vertices. Letting $G$ be a graph, and $S$ a supplemental graph, we define a new graph $G^S$ as follows: the vertex set of $G^S$ is the same as the vertex set of $G$. Meanwhile, there is an edge between vertices $v$ and $w$ in $G^S$ if there exists a non-induced subgraph isomorphism $i$ from $S$ to $G$ which maps the two distinguished vertices of $S$ to $v$ and $w$ respectively. It is reasonably straightforward to prove that any subgraph isomorphism $i : P \to T$ also defines a subgraph isomorphism $i^S : P^S \to T^S$, where $i^S(v) = i(v)$. The Glasgow Subgraph Solver uses this result to generate additional degree and adjacency-like constraints. This is sometimes extremely powerful, as illustrated by the example in Fig. 2. Currently the choice of supplemental graphs is hard-coded, based upon performance on a range of standard benchmark instances, but we believe it may be possible to automatically make different choices for different families of problem instance.

*All-Different Filtering.* Suppose a pattern graph and a target graph both have exactly five vertices of degree five or higher, then those five vertices in the target graph cannot be mapped to by any other pattern vertex in an injective mapping. This is an example of all-different reasoning: more generally, if any $n$ undecided pattern vertices have less than $n$ available target vertices between them, we have found a contradiction, and if they have exactly $n$ available target vertices between them then those target vertices must all be used only for those

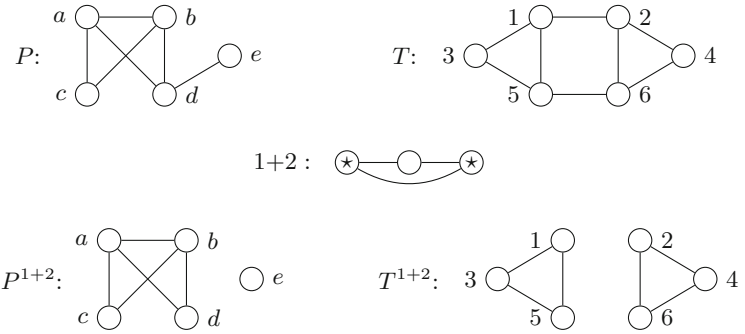


**Fig. 2.** On top, a pair of graphs $P$ and $T$. In the middle, the supplemental graph $1+2$. On the bottom, the modified graphs $P^{1+2}$ and $T^{1+2}$. These modified graphs make it immediately clear that no subgraph isomorphism exists between $P$ and $T$.

pattern vertices. Deciding exactly how to filter all-different constraints is one of the big differences between constraint programming approaches for subgraph isomorphism [4,22,25]. Currently, the Glasgow Subgraph Solver uses a special bit-parallel propagator, which gives a good tradeoff between performance and filtering power [18].

The other major contributing aspect to a constraint programming solver's performance is how it carries out backtracking search.

*Search Order.* When performing a backtracking search, the choices of which variable to branch on, and which value to try first, can make a staggering difference to performance in practice. The Glasgow Subgraph Solver uses carefully chosen strategies to decide how to direct its search [19], including always branching on whichever vertex has fewest possibilities available to it (tie-breaking on highest degree). This has interesting implications, which are not yet fully understood. For example, in the absence of other filtering, this will cause the solver to always grow connected components, which is the optimal behaviour for certain kinds of pattern graph—but it is not clear whether exploiting additional filtering could theoretically lose us performance guarantees in some cases.

*Restarts and Nogood Recording.* Rather than using simple backtracking, the solver employs restarts and nogood recording [16,17]: the solver runs for a small amount of time, and then restarts from the beginning, remembering not to revisit any part of the search space which has already been explored. Combined with a small amount of heavily biased randomness in how branching is carried out, this avoids a strong commitment to early branching choices, which are most difficult for a heuristic to get right [3].

*Parallelism.* Modern hardware provides a range of opportunities for parallelism. The Glasgow Subgraph Solver exploits this in two ways: by using bit-parallel

data structures and algorithms to carry out inference as quickly as possible [18], and by using threads to explore multiple parts of the search space in parallel [3]. These parallel search capabilities scale at least as far as thirty-six cores.

### 3.2 Future Directions

We finish with a discussion of possible future directions for the solver, and with ideas for research and engineering challenges which may be of broader interest.

*Problem Variants.* There are other problems involving finding mappings between subgraphs, such as a surjective variant [13]. Some problems also involve wildcards, not just on labels, but on pattern vertices; work on $k$-less subgraph isomorphism [14] may prove useful for continuing to allow powerful inference when wildcards are present. More generally, we have experimental support for connecting the solver to an external constraint programming solver, to handle arbitrary side constraints (a bit like Satisfiability Modulo Theories). This could be useful, for example, for temporal graphs [21]; we would be interested in exploring this direction further to tackle suitable real-world applications. Another potential application area is inside graph rewriting systems [10]. Here, the pattern graphs are considered "fixed", rather than being part of the input, which has implications for the theoretical complexity of the problem. However, when patterns are numerous or large and complex, or when side constraints are involved [2], it may be more practical to use a general purpose solver than a dedicated algorithm for each special case.

*Symmetries.* Some applications involve heavily symmetric pattern and target graphs [27]. Handling such symmetries in constraint programming is, in principle, a well-understood problem. However, a practical difficulty is that because the symmetries vary on an instance by instance basis, symmetry-breaking constraints must be computed for each individual input rather than for a model as a whole. An implementation of the Schreier-Sims algorithm [24] which has no costly external dependencies would make this approach more practical.

*Faster Counting.* Currently, the solver handles the counting problem by explicit enumeration, except that for non-induced isomorphisms, any isolated vertices in the pattern graph are treated specially. Although counting and enumeration are equally difficult in general, we believe there are further opportunities for speeding up counting, for example by decomposing the pattern graph into nearly-unconnected components, or by handling pattern vertices of degree one and two specially. We would also be interested in implementing approximate counting as an option, as well as seeing whether uniform sampling of solutions can be carried out more efficiently in practice than by explicit enumeration.

*Special Classes of Pattern.* Certain special classes of pattern may be counted efficiently—for example, if the pattern is a star graph. Some applications involve counting occurrences of many different kinds of small graph [1,9,20], and so it

would be useful if solvers could detect when they were in an "easy" case and switch algorithms, rather than relying upon end users to do this. There are also classes of pattern graph where decision and counting are still NP-hard, but where more efficient solving techniques are available—the solver currently switches to a different dedicated algorithm if the input graph is a clique, for example.

*Proof Logging.* Given the increasing complexity of both the theory and implementations of subgraph isomorphism algorithms, we should be concerned as to whether the outputs produced are correct. In the Boolean satisfiability community, proof logging is the standard solution to this problem: solvers that claim unsatisfiability are expected to be able to output a machine-verifiable proof of this fact. Recently, Elffers et al. [11] introduced a more flexible form of proof logging, that we believe is better suited for algorithms that perform strong inference. The Glasgow Subgraph Solver includes experimental support for producing proofs in this format, and we hope to see further research in this direction.

*Automatic Configuration.* The solver supports a wide range of filtering options. Its default configuration is designed to reduce the chances of poor performance on hard instances, rather than to do well on very easy instances—for example, it will create supplemental graphs before attempting any search, which is a relatively expensive one-time cost that is not necessary for solving many instances. We have previously shown that it can be beneficial to employ a simple connectivity-based algorithm as a presolver [15]. However, it may be possible to take automatic algorithm configuration further, for example by selecting the set of supplemental graphs to use on an instance by instance basis.

*Benchmarking.* Finally, given the importance of having good instances for benchmarking and for informing algorithm design, we would be very interested in collecting sets of instances from other applications. The instances by Solnon[1] originally used for algorithm portfolios [15] give a good starting point, but having more instances from a diverse range of applications would be very beneficial—even if those instances are all either very easy for all solvers, or are too hard for any current solver to solve at all. We would very much welcome contributions from the community.

**Acknowledgements.** We would like to thank Blair Archibald, Fraser Dunlop, Jan Elffers, Stephan Gocht, Ruth Hoffmann, Jakob Nordström, and Christine Solnon for their contributions to the design and implementation of the solver. This work was supported by the Engineering and Physical Sciences Research Council (grant numbers EP/P026842/1, EP/M508056/1, and EP/N007565).

## References

1. Alon, N., Dao, P., Hajirasouliha, I., Hormozdiari, F., Sahinalp, S.C.: Biomolecular network motif counting and discovery by color coding. Bioinformatics (Oxford, England) **24**(13), i241–i249 (2008)

[1] https://perso.liris.cnrs.fr/christine.solnon/SIP.html.

2. Archibald, B., Calder, M., Sevegnani, M.: Conditional bigraphs. In: 13th International Conference on Graph Transformation (ICGT 2020), Bergen, Norway, 25–26 June 2020 (2020)
3. Archibald, B., Dunlop, F., Hoffmann, R., McCreesh, C., Prosser, P., Trimble, J.: Sequential and parallel solution-biased search for subgraph algorithms. In: Rousseau, L.-M., Stergiou, K. (eds.) CPAIOR 2019. LNCS, vol. 11494, pp. 20–38. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19212-9_2
4. Audemard, G., Lecoutre, C., Samy-Modeliar, M., Goncalves, G., Porumbel, D.: Scoring-based neighborhood dominance for the subgraph isomorphism problem. In: O'Sullivan, B. (ed.) Principles and Practice of Constraint Programming, CP 2014. Lecture Notes in Computer Science, vol. 8656, pp. 125–141. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_12
5. Blindell, G.H., Lozano, R.C., Carlsson, M., Schulte, C.: Modeling universal instruction selection. In: Proceedings of Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, 31 August–4 September 2015, pp. 609–626 (2015)
6. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D.E., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinf. **14**(S–7), S13 (2013)
7. Carletti, V., Foggia, P., Saggese, A., Vento, M.: Introducing VF3: a new algorithm for subgraph isomorphism. In: Foggia, P., Liu, C.-L., Vento, M. (eds.) GbRPR 2017. LNCS, vol. 10310, pp. 128–139. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58961-9_12
8. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004)
9. Davies, T., Marchione, E.: Event networks and the identification of crime pattern motifs. PLOS ONE **10**(11), 1–19 (2015)
10. Dörr, H. (ed.): Efficient Graph Rewriting and Its Implementation. LNCS, vol. 922. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0031909
11. Elffers, J., Gocht, S., McCreesh, C., Nordström, J.: Justifying all differences using Pseudo-Boolean reasoning. In: Proceedings of AAAI (2020). in press
12. Fiala, J., Kratochvíl, J.: Locally constrained graph homomorphisms-structure, complexity, and applications. Comput. Sci. Rev. **2**(2), 97–111 (2008)
13. Gay, S., Fages, F., Martinez, T., Soliman, S., Solnon, C.: On the subgraph epimorphism problem. Discret. Appl. Math. **162**, 214–228 (2014)
14. Hoffmann, R., McCreesh, C., Reilly, C.: Between subgraph isomorphism and maximum common subgraph. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, California, USA, 4–9 February 2017, pp. 3907–3914 (2017)
15. Kotthoff, L., McCreesh, C., Solnon, C.: Portfolios of subgraph isomorphism algorithms. In: Festa, P., Sellmann, M., Vanschoren, J. (eds.) LION 2016. LNCS, vol. 10079, pp. 107–122. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50349-3_8
16. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Nogood recording from restarts. In: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, 6–12 January 2007, pp. 131–136 (2007)
17. Lee, J.H.M., Schulte, C., Zhu, Z.: Increasing nogoods in restart-based search. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, Arizona, USA, 12–17 February 2016, pp. 3426–3433 (2016)

18. McCreesh, C., Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In: Proceedings of Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, 31 August–4 September 2015, pp. 295–312 (2015)
19. McCreesh, C., Prosser, P., Solnon, C., Trimble, J.: When subgraph isomorphism is really hard, and why this matters for graph databases. J. Artif. Intell. Res. **61**, 723–759 (2018)
20. Mukherjee, K., Hasan, M.M., Boucher, C., Kahveci, T.: Counting motifs in dynamic networks. BMC Syst. Biol. **12**(1), 6 (2018)
21. Redmond, U., Cunningham, P.: Temporal subgraph isomorphism. In: Advances in Social Networks Analysis and Mining 2013, ASONAM 2013, Niagara, ON, Canada, 25–29 August 2013, pp. 1451–1452 (2013)
22. Régin, J.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, 31 July–4 August 1994, vol. 1, pp. 362–367 (1994)
23. Sevegnani, M., Calder, M.: Bigraphs with sharing. Theor. Comput. Sci. **577**, 43–73 (2015)
24. Sims, C.C.: Computational methods in the study of permutation groups. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 169–183. Pergamon (1970)
25. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. Artif. Intell. **174**(12–13), 850–864 (2010)
26. Solnon, C.: Experimental evaluation of subgraph isomorphism solvers. In: Conte, D., Ramel, J.-Y., Foggia, P. (eds.) GbRPR 2019. LNCS, vol. 11510, pp. 1–13. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20081-7_1
27. Vömel, C., de Lorenzi, F., Beer, S., Fuchs, E.: The secret life of keys: on the calculation of mechanical lock systems. SIAM Rev. **59**(2), 393–422 (2017)
28. Zampelli, S., Deville, Y., Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints **15**(3), 327–353 (2010)