Solving Graph Homomorphism and Subgraph Isomorphism Problems Faster Through Clique Neighbourhood Constraints

Sonja Kraiczy[1] , **Ciaran McCreesh**[2]
[1]University of Oxford
[2]University of Glasgow
ciaran.mccreesh@glasgow.ac.uk

Abstract

Graph homomorphism problems involve finding adjacency-preserving mappings between two given graphs. Although theoretically hard, these problems can often be solved in practice using constraint programming algorithms. We show how techniques from the state-of-the-art in subgraph isomorphism solving can be applied to broader graph homomorphism problems, and introduce a new form of filtering based upon clique-finding. We demonstrate empirically that this filtering is effective for the locally injective graph homomorphism and subgraph isomorphism problems, and gives the first practical constraint programming approach to finding general graph homomorphisms.

1 Introduction

The *subgraph isomorphism problem*, as defined by Garey and Johnson [1979], is to determine whether an injective mapping exists from one given graph to another, such that adjacent pairs of vertices are mapped to adjacent pairs of vertices, and non-adjacent vertices are mapped to non-adjacent vertices. However, in application-oriented papers, particularly from bioinformatics and chemistry, the same name is often implicitly used to mean the *non-induced* version of the problem (which does not require that non-adjacency be preserved) since this more accurately models the real-world problem being solved [Willett, 1999; Ehrlich and Rarey, 2011]. Despite being theoretically hard problems, these applications, along with others in areas including compilers [Blindell *et al.*, 2015], graph databases [McCreesh *et al.*, 2018] and pattern recognition [Foggia *et al.*, 2014], have given rise to a large amount of research into designing practical algorithms for solving these problems. Most approaches are based either upon very fast but simple backtracking algorithms [Cordella *et al.*, 2004; Bonnici *et al.*, 2013; Carletti *et al.*, 2017] which often but not always perform well on very easy instances, or upon constraint programming algorithms [Zampelli *et al.*, 2010; Solnon, 2010; Audemard *et al.*, 2014; McCreesh and Prosser, 2015; Archibald *et al.*, 2019], which have higher startup costs but that perform vastly better on harder instances and much more consistently on easy instances [McCreesh *et al.*, 2018; Solnon, 2019].

The current state of the art is the Glasgow Subgraph Solver [McCreesh *et al.*, 2020], which is a dedicated constraint programming solver for subgraph-finding problems. Much of its performance comes from inference strategies based upon degrees and neighbourhood degree sequences [Zampelli *et al.*, 2010], counting paths between vertices [Audemard *et al.*, 2014; McCreesh and Prosser, 2015], and cardinality reasoning [Solnon, 2010], which can be used to eliminate many infeasible candidate assignments without search. Like most other solvers, it can handle both the non-induced version of the problem and the traditional induced variant, and users can select whichever variant better fits their application. It also supports directed edges, and can use an external constraint solver to handle other extensions to the basic notion of a graph, such as temporal networks and multi-graphs.

As well as questions over exactly how edges are represented and mapped, some applications would prefer not to specify injectivity. A *graph homomorphism* is a function between two graphs that maps adjacent vertices to adjacent vertices, with no injectivity requirement, whilst a *locally injective homomorphism* is one which is injective when restricted to any individual vertex and its neighbourhood. Both are better models for some applications [Fiala *et al.*, 2001; Baget, 2005; Corby and Faron-Zucker, 2007; Fiala and Kratochvíl, 2008; Fan *et al.*, 2010; Chaplick *et al.*, 2015].

These injectivity relaxations have received much less attention from a practical solving perspective. One might hope that the inference techniques developed for subgraph isomorphism would also be helpful for other graph homomorphism problem variants. Indeed, this paper proves that many, but not all, of these strategies are also valid in the locally injective case, and that simple distance filtering is valid for all homomorphisms. However, we also show that *none* of the other strategies are valid for finding homomorphisms where there is no injectivity requirement. Finally, we introduce a new filtering technique that is based upon finding a maximum clique in the neighbourhood of each domain vertex, which is valid even in the general case. Although this new filtering technique involves solving many additional NP-complete problems as a preprocessing step, we demonstrate that it is effective in practice, particularly for the non-injective problem where filtering allows a constraint programming algorithm to solve instances over eight hundred times faster in aggregate, over a collection of over fourteen thousand stan-

dard benchmark instances. This shows, for the first time, that modern constraint programming techniques can be practical for less constrained graph homomorphism finding problems; previous algorithmic approaches have focused instead upon worst-case computational bounds (e.g. Fomin *et al.*; Fiala and Kratochvíl; Rzazewski; Chaplick *et al.* [2007; 2008; 2014; 2015]), whose practical utility has yet to be demonstrated. The empirical effectiveness of our results is especially important because many applications use subgraph isomorphism solvers only because they perform well off the shelf, rather than because they exactly match domain requirements.

2 Background and Theory

We begin by introducing notation and terminology, and by giving the theory which supports our implementation.

Graphs. Let G and H be graphs. Let $v \in \mathrm{V}(G)$ be a vertex of G. The *(open) neighbourhood* of v, written $\mathrm{N}_G(v)$, is the set of vertices adjacent to v not including v itself, whilst the closed neighbourhood of v, written $\mathrm{N}_G[v]$, is the neighbourhood of v plus v. The *degree* of a vertex, $\deg_G(v)$, is the cardinality of its open neighbourhood. The *neighbourhood degree sequence* of a vertex is the sequence consisting of the degrees of its neighbours, in descending order. Given a vertex set $S \subseteq \mathrm{V}(G)$, the subgraph *induced by* S, written $G[S]$, is the subgraph of G with only the vertices in S together with all the edges between them. A *clique* is a subgraph where every vertex is adjacent to every other in the subgraph.

Homomorphisms. A *homomorphism* from G to H is a function mapping vertices of G to vertices of H, such that adjacent vertices in G are mapped to adjacent vertices in H. A homomorphism h is *locally injective* if for every vertex w, the restriction of h to $G[N_G[w]]$ is injective; if h is injective globally we call it a *(non-induced) subgraph isomorphism*.

Loops. A vertex which is adjacent to itself is called a *loop*. By a careful reading of the definition, any homomorphism must map loops onto loops—and indeed, the Glasgow Subgraph Solver and this paper take this approach (although some other algorithm implementations do not). It therefore follows that for the homomorphism *decision* problem where there are no injectivity constraints, any problem instance which has a loop in the codomain graph is trivially satisfiable (although the *counting* problem remains #P-hard [Dyer and Greenhill, 2000]).

Constraint programming. A *constraint satisfaction problem* is defined in terms of a set of *variables*, each of which has a *domain* of possible *values*, together with a set of constraints; the goal is to give each variable a value from its domain, whilst satisfying all of the constraints. Homomorphism problems have a natural representation as a constraint satisfaction problem: we have a variable for each vertex in the domain graph, whose values range over the codomain graph, and a set of constraints saying that adjacent vertices must be mapped to adjacent vertices. For the injective variants, one or more *all-different* constraints are also present. A typical constraint programming approach to solving such a problem is to combine inference through constraint propagation with an intelligent backtracking search.

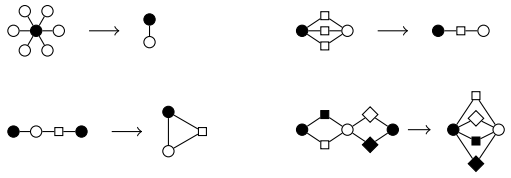

Figure 1: Counter-examples for various properties which are not invariants. The different shapes for vertices show a mapping, with vertices being mapped to vertices of the same shape. The top left example shows that degree is not preserved in a homomorphism; the top right that path counts are not preserved in a homomorphism; the bottom left that path counts of length three are not preserved in a locally injective homomorphism; and the bottom right that a locally injective homomorphism i can give rise to a homomorphism $i^{2,2}$ which is not locally injective.

Degree-based invariants. Constraint programming techniques for finding homomorphisms can be made much more effective by exploiting certain invariants. It is straightforward to verify the following.

Proposition 1 (neighbourhood degree sequences are preserved)**.** *Suppose i is a locally injective homomorphism from G to H. Let v be any vertex. Then i cannot map v to a vertex of lower degree, $\deg_G(v) \leq \deg_H(i(v))$. Furthermore, v cannot be mapped to any vertex whose neighbourhood degree sequence is not pointwise greater than or equal to its own.*

Since subgraph isomorphisms are in particular locally injective, these invariants also hold for subgraph isomorphisms [Zampelli *et al.*, 2010]; indeed, every recent constraint programming approach for subgraph isomorphism finding makes use of degrees and neighbourhood degree sequences. The simplest way for a constraint programming algorithm to use these results is as unary constraints, which are propagated when domains are initialised, before search begins. For each domain representing a domain vertex, any value representing a codomain vertex whose degree or neighbourhood degree sequence is too low may immediately be rejected.

However, these invariants do *not* hold for homomorphisms in general. As a counter-example, a homomorphism may map a star graph onto a single edge, as in figure 1 (top left).

Path-based invariants. Constraint programming solvers can also exploit invariants that are based upon paths. This is done by automatically adding *implied* constraints to the problem that are implied by the original model, but that will give stronger propagation power.

Proposition 2 (paths are preserved by subgraph isomorphisms)**.** *For the problem of finding a subgraph isomorphism i from a graph G to a graph H, the following constraints are implied for any pair of vertices $v, w \in \mathrm{V}(G)$:*

1. *The distance between v and w is at most the distance between $i(v)$ and $i(w)$ [Audemard* et al.*, 2014].*
2. *If there are at least k simple paths of length exactly ℓ between v and w, then there must be at least k simple paths of length exactly ℓ between $i(v)$ and $i(w)$ [McCreesh and Prosser, 2015].*

It is easy to verify that the first of these two properties is valid for *any* homomorphism. The second property does

not hold for homomorphisms in general: for example, a pair of vertices connected by three paths of length two may be mapped onto a pair of vertices connected by a single path of length two (see figure 1, top right). The second property also does not hold in full generality for local injectivity, because, for example, a path of length three may be mapped onto a triangle (figure 1, bottom left): while standard injectivity prevents any kind of merging, local injectivity only prevents two vertices from being merged if they are both in the neighbourhood of the same vertex. However, a weaker result does hold:

Proposition 3 (paths of length two are preserved by locally injective homomorphisms)**.** *For the problem of finding a locally injective graph homomorphism i from a graph G to a graph H, for any pair of vertices $v, w \in \mathrm{V}(G)$, if there are at least k simple paths of length exactly two between v and w, then there must be at least k simple paths of length exactly two between $i(v)$ and $i(w)$.*

Proof. Let $\{x_1, \ldots, x_n\}$ be the intermediate vertices on the paths of length two between v and w. Observe that each x_j is in the neighbourhood of v, and so must be mapped to different vertices. Thus each sequence $(i(v), i(x_j), i(w))$ gives a distinct simple path of length two between $i(v)$ and $i(w)$. □

Rather than using distance properties directly as Audemard *et al.* [2014] did, McCreesh and Prosser [2015] introduced the notion of *supplemental* graphs, with the idea that a constraint programming algorithm can search for a mapping which is simultaneously a subgraph isomorphism between several different pairs of graphs. Let G^d be the graph with the same set of vertices as G, but with an edge between vertices v and w if the distance between v and w in G is at most d. Similarly, let $G^{n,\ell}$ be the graph with the same set of vertices as G, but an edge between vertices v and w if there are at least n simple paths of length exactly ℓ between v and w in G. Following on from propositions 2 and 3, we generalise the result of McCreesh and Prosser as follows.

Corollary 1. *Any homomorphism i from G to H gives a homomorphism i^d from G^d to H^d defined by $i^d(v) = i(v)$, for all d. Furthermore, if i is locally injective, then for all n, there is a homomorphism $i^{n,2}$ from $G^{n,2}$ to $H^{n,2}$ defined by $i^{n,2}(v) = i(v)$.*

Note carefully that $i^{n,2}$ is *not* necessarily locally injective, even if i is; we illustrate a counter-example in figure 1 (bottom right). This is in contrast to subgraph isomorphism, where $i^{n,\ell}$ is a subgraph isomorphism for all ℓ [McCreesh and Prosser, 2015].

Cliques and homomorphisms. We now discuss the main new technique of this paper. Observe that so far, we have not seen any unary constraints which are valid for the general homomorphism problem. All existing constraint programming approaches for the subgraph isomorphism problem begin by branching on the variable which has the smallest domain—typically this will be a domain for a vertex of high degree, or which has many high degree neighbours. Furthermore, once one such guessed assignment has been made, adjacency propagation means many other domains will be substantially reduced in size, making subsequent branching choices simpler. However, for homomorphism problems, if we cannot find any implied unary constraints then every domain will initially be of the same size, which will make it much harder for a solver to know where to start. This motivates the following.

Proposition 4 (Cliques are preserved)**.** *Let i be a homomorphism from G to H where H does not contain any loops. Let $S \subseteq V$ be such that $G[S]$ is a k-vertex clique. Then $H[i(S)]$ is also a k-vertex clique.*

Proof. Let j and k be two distinct vertices of the clique in G. By definition of a homomorphism, $i(j)$ and $i(k)$ must be adjacent. And, because H has no loops, $i(j) \neq i(k)$. □

Corollary 2. *Proposition 4 holds for locally injective graph homomorphisms and for subgraph isomorphisms even if loops are present in the second graph.*

Next we will discuss how this is useful in practice.

3 Design and Implementation

Having determined which commonly-used subgraph isomorphism invariants do and do not hold for other forms of homomorphism, and having discovered a new clique-based invariant, we will now look at how these invariants may be used in practice. Our starting point is the Glasgow Subgraph Solver [McCreesh *et al.*, 2020], due to it being the current strongest solver for subgraph isomorphism. Adapting the solver to handle locally injective and general homomorphism problems required the following straightforward changes.

Injectivity constraints. For subgraph isomorphism, injectivity is handled by a combination of binary constraints and a specialised bit-parallel all-different propagator [McCreesh and Prosser, 2015]. This was disabled entirely for homomorphism problems, and for local injectivity only binary constraints were used.

Path-based filtering. For subgraph isomorphism, the Glasgow Subgraph Solver searches for a mapping which is simultaneously a subgraph isomorphisms between (G, H) and each $(G^{n,2}, G^{n,2})$ for each n from 1 to 4, and optionally also between (G^3, H^3). For homomorphisms, we use (G^2, H^2), whilst for locally injective homomorphisms we use $(G^{n,2}, G^{n,2})$ for each n from 1 to 4.

Degree-based filtering. This was disabled for homomorphisms and left enabled for locally injective homomorphisms, as per proposition 1. For subgraph isomorphism, the Glasgow Subgraph Solver uses degree and neighbourhood degree sequences not just on the original graphs, but also on each $(G^{n,\ell}, H^{n,\ell})$ graph pair. This is *not* possible for locally injective homomorphisms, due to the counter-example following corollary 1.

Search order heuristics. The solver's default search order heuristics are based upon three principles: that it is good to branch on variables with few remaining values in a domain, that it is good to branch on variables corresponding to high-degree vertices, and that it is good to try mapping to vertices of high degree [McCreesh *et al.*, 2018; Archibald *et al.*, 2019]. These principles do not appear to be specific to subgraph isomorphism, and so we do not alter the search order heuristics.

Loops. For the homomorphism decision problem, we modified the solver so that if the codomain graph contains a loop, we immediately return a satisfying assignment mapping every vertex to one of these loops.

3.1 Clique Filtering

Implementing clique constraints required substantially more work. The Glasgow Subgraph Solver contains a maximum clique implementation, which is based upon a variation of Tomita *et al.* [2010]'s MCS algorithm which Prosser [2012] calls "MCSa1", with bit-parallelism [San Segundo *et al.*, 2013] and an altered branching scheme for faster optimality proofs [McCreesh and Prosser, 2014]. The implementation also makes use of a shortcut due to Batsyn *et al.* [2014], which allows certain cliques to be detected without branching—this turns out to be particularly useful in this setting, making many of the clique instances generated solvable without branching. Initially we used this solver to perform domain filtering as a preprocessing step. Preliminary experiments showed that a naïve approach, which calculates the maximum clique size for the neighbourhood of each domain vertex and each codomain vertex, would add as much as three minutes of preprocessing time to some problem instances which could otherwise be solved in a few seconds. We therefore invested more engineering effort, as follows.

Assuming a problem instance is not detected as obviously unsatisfiable, we calculate the maximum clique size for the neighbourhood of every domain vertex in turn. However, having found a maximum clique with k vertices in the neighbourhood of a vertex p, we remember for every other vertex in this maximum clique that its neighbourhood maximum clique size is at least k. This can be used to accelerate subsequent clique solver calls, by starting the branch and bound algorithm with an initial incumbent size of k rather than zero.

We then move on to the codomain vertices, using a similar caching routine. Rather than calculating a clique size for every single codomain vertex, we only calculate a value for codomain vertices which are present in at least one variable's domain, after other unary constraints have been applied. Additionally, we do not require the maximum clique solver to run to completion and guarantee that it has found a maximum clique. Instead, we allow it to stop as soon as it has found a clique with as many vertices as the largest domain clique. This is useful because it may be very hard to decide whether a particular codomain vertex has neighbourhood clique size of, say, 15 or 16, but if the largest domain vertex has a neighbourhood clique size of only 5 then this is irrelevant.

3.2 Proof Logging

Given the complexity of modern solvers, a critical question is how we can be sure that they are producing correct answers. The Glasgow Subgraph Solver's subgraph isomorphism and clique algorithms are both *certifying* [McConnell *et al.*, 2011]: that is, they can output a mathematical proof that they have reached a correct answer by sound reasoning, and this proof log can be verified by the VeriPB proof checking tool [Gocht *et al.*, 2020a; Gocht *et al.*, 2020b]. We therefore extended the solver's existing proof logging support to cover our additions. For the changes not involving clique-finding, this was entirely routine—although proof logging did catch an implementation bug where we were incorrectly applying degree-based filtering on the $G^{2,n}$ graphs for locally injective homomorphisms, in violation of the counter-example to corollary 1. For clique-based filtering, the process involved substantially more technical work. Briefly, for a given domain vertex v that cannot be mapped to a codomain vertex w, it is possible to instruct the proof verifier to derive a set of conditional clique constraints saying that "either v cannot be mapped to w, or the following clique-like problem must be satisfiable" by following steps similar to the maximum common subgraph to clique reduction presented by Gocht *et al.* [2020a], and then to reuse clique proof-logging techniques to show that this clique-like problem is unsatisfiable. We were thus able to produce and verifying proofs of correctness for medium-sized instances, giving us confidence our implementation was correct.

This situation is not entirely ideal. The proofs produced this way are inherently dependent upon a *particular* choice of domain and codomain vertices. Thus, when proof logging, we must solve restricted clique problems for each assignment that is to be filtered using a clique rule, rather than for each clique found; this is potentially a linear factor blowup. This seems wasteful, since for any given codomain vertex these proofs are effectively the same up to substitution of variable names. It would therefore seem desirable to extend the proof system with a substitution rule which would allow such proofs to be recycled to avoid duplication.

4 Experiments

We now evaluate this approach empirically. Our experiments are performed on a cluster of machines with dual Intel Xeon E5-2697A v4 CPUs with 512GBytes of RAM, running Ubuntu 18.04, and using a timeout of one hour; we do not enable proof logging for these benchmarks since proof logging introduces large slowdowns due to I/O costs [Gocht *et al.*, 2020a]. Because injectivity relaxations have received less attention until now, for all three problem variants we will use a suite of 14,621 benchmark instances designed for the subgraph isomorphism problem [Kotthoff *et al.*, 2016]. Note that 8,904 of these instances contain at least one loop in the codomain graph, and so are trivial for the homomorphism problem—we still include these to avoid using different y-axes on different plots.

In the top left plot of figure 2 we show the cumulative number of instances solved over time, for each of the three problem variants, using the Glasgow Subgraph Solver with and without our clique-filtering added. For the homomorphism problem, we also compare with distance filtering enabled or disabled. For subgraph isomorphism, we further compare against the PathLAD [Kotthoff *et al.*, 2016], VF2 [Cordella *et al.*, 2004] and RI [Bonnici *et al.*, 2013] solvers. To make the details of this plot readable, the second row of figure 2 zooms in on the results for each problem variant in turn. We see that for the homomorphism problem, using either clique or distance filtering gives a clear improvement to performance, and using both together is better still. For both injective problems, clique filtering gives a more moderate improvement.

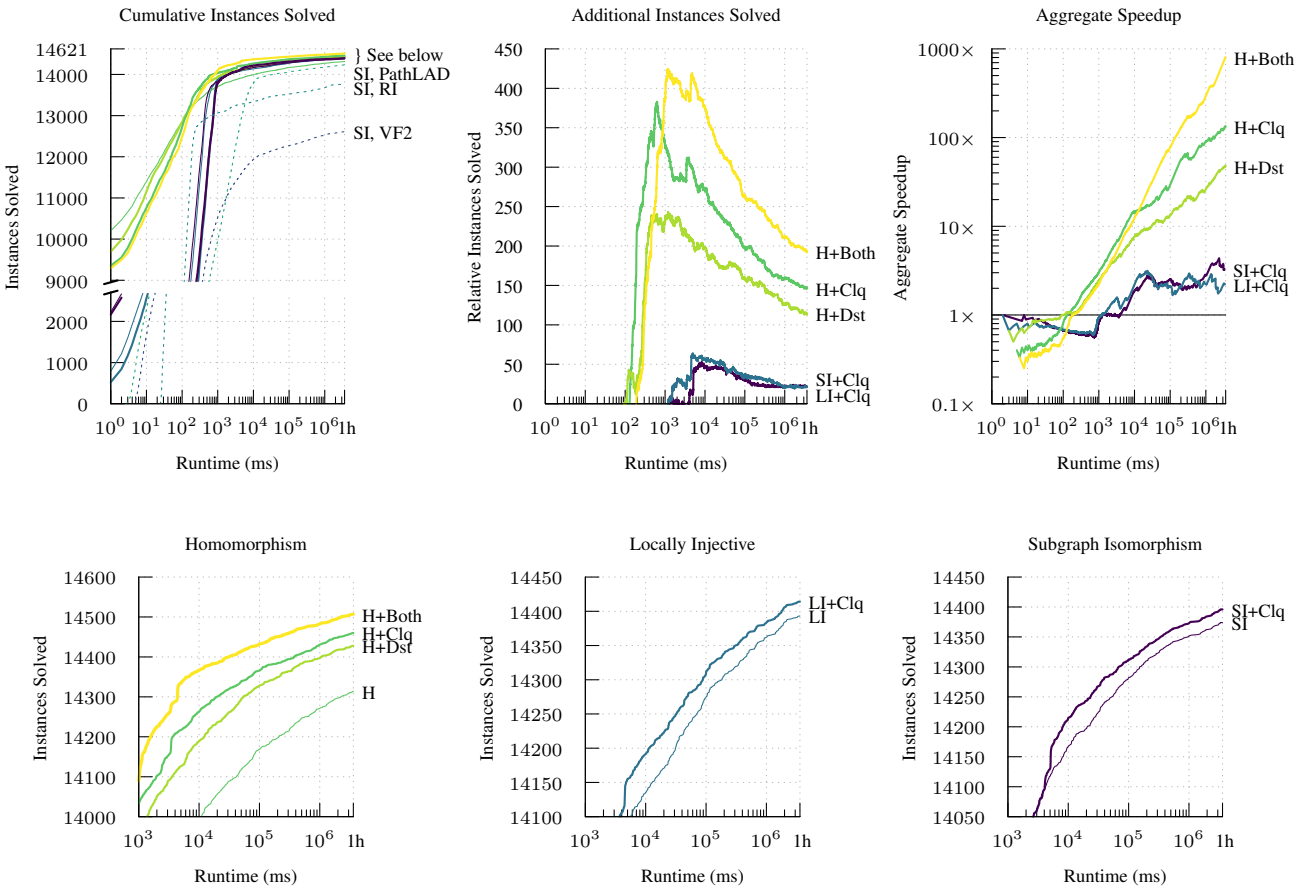

Figure 2: On the top left, the cumulative number of instances solved over time for the three problem variants, with and without clique filtering, and with and without distance filtering for the homomorphism problem; for comparison, results for subgraph isomoprhism using the PathLAD, RI, and VF2 solvers are also shown. The remaining plots re-display this data, as follows. The three plots on the bottom row zoom in on the cumulative number of instances solved, for the homomorphism problem on the left, the locally injective homomorphism problem in the centre, and the subgraph isomorphism problem on the right. The top centre plot shows the additional number of instances solved at any given time when using the new forms of filtering for all problem variants, and the top right plot shows the aggregate speedups from each form of filtering.

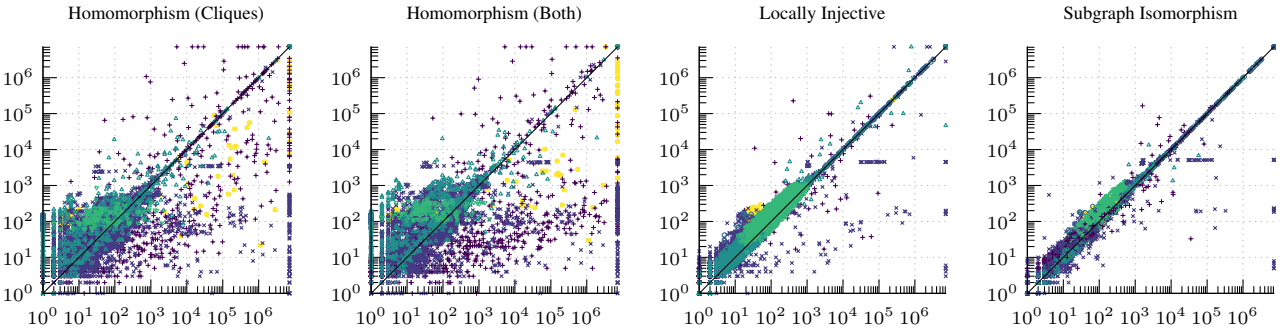

Figure 3: Looking at the effects of additional filtering on an instance by instance basis, for homomorphism with just clique filtering and with both clique and distance filtering, and for the other two variants with clique filtering. Each point represents one instance, the vertical axis is the runtime with filtering in ms, and the horizontal axis is the runtime without filtering in ms (and so points below the diagonal are speedups). Points on the outer axes are timeouts. The different point styles show the different families of instance from the benchmark set, and illustrate that in each the filtering is broadly useful rather than being specific to a single kind of application.

To quantify these gains, the top centre plot of figure 2 shows the additional number of instances solved at any given time (that is, the vertical distance between two cumulative plots) when enabling additional filtering. We see that for subgraph isomorphism, at the one hour timeout we solve 22 additional instances using clique filtering; for locally injective homomorphisms, 21 additional instances; and for homomorphisms, 114, 146 or 193 additional instances using just distance filtering, just clique filtering, or both. Or, at the best choices of timeouts, we solve 53 additional subgraph isomorphism instances; 64 additional locally injective instances; and 243, 383, and 424 additional homomorphism instances.

Alternatively, we can measure the horizontal distance between plots for a given choice of timeout t as follows. Let t' be the last time no greater than t where the slower algorithm solved an instance, and let u be the lowest amount of time needed for the faster algorithm to solve the same number of instances. The *aggregate speedup*, which we plot in the top right of figure 2, is the ratio of t' to u or u to t' as appropriate. At the one hour timeout, we see aggregate speedups of 2.2 for locally injective homomorphisms, 3.2 for subgraph isomorphism, and 47.5, 131.7, and 803.5 for homomorphisms.

The crossover point, where clique filtering begins to pay off, is at around the 100ms mark without injectivity, and at around one second for the injective problems. This does not show how much time was spent in clique-finding, but rather how long it takes for an algorithm with increased filtering to catch up from a slower start. In fact, for domain graphs, no maximum clique call took even one millisecond to complete (and none required more than two hundred recursive calls from the clique solver), and more time was spent setting up the data structures to represent the neighbourhood graphs than on the actual solving. For codomain graphs, 20,838,008 of the clique problems we solved took under a millisecond, and the remaining 279 calls took between one and four milliseconds, with no instance requiring more than a thousand recursive calls. Although it may seem counter-intuitive to try to speed up the solving of a hard problem by first solving many other hard problems, these values demonstrate that clique-solving on these smaller graphs is *much* easier in practice than solving the larger problem, and so is worthwhile.

Finally, figure 3 plots instance-by-instance comparisons of the runtimes. The point styles in these plots show different families of benchmark instance, and demonstrate that the benefits of filtering are not restricted to one application. They also show that clique filtering very rarely makes performance a lot worse, but often makes it a lot better.

5 Conclusion

Our results have shown that constraint programming approaches are viable for a wider range of subgraph mapping finding problems than had previously been considered, and will increase the versatility of subgraph solvers for application users going forwards. We saw that using clique-finding as a filtering step could give moderate to very large speedups for different problem variants—although only through careful engineering choices, to mitigate the potential cost of solving NP-hard problems. We would be interested to see if filtering based upon other hard problems could give similar benefits—for example, homomorphisms also interact in potentially helpful ways with various colouring properties.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/P026842/1].

References

[Archibald *et al.*, 2019] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Proceedings*, 2019.

[Audemard *et al.*, 2014] Gilles Audemard, Christophe Lecoutre, Mouny Samy Modeliar, Gilles Goncalves, and Daniel Cosmin Porumbel. Scoring-based neighborhood dominance for the subgraph isomorphism problem. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014*, 2014.

[Baget, 2005] Jean-François Baget. RDF entailment as a graph homomorphism. In *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005*, volume 3729 of *LNCS*, pages 82–96. Springer, 2005.

[Batsyn *et al.*, 2014] Mikhail Batsyn, Boris Goldengorin, Evgeny Maslov, and Panos M. Pardalos. Improvements to MCS algorithm for the maximum clique problem. *J. Comb. Optim.*, 27(2):397–416, 2014.

[Blindell *et al.*, 2015] Gabriel Hjort Blindell, Roberto Castañeda Lozano, Mats Carlsson, and Christian Schulte. Modeling universal instruction selection. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015*, 2015.

[Bonnici *et al.*, 2013] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(S-7):S13, 2013.

[Carletti *et al.*, 2017] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Introducing VF3: A new algorithm for subgraph isomorphism. In *Graph-Based Representations in Pattern Recognition - 11th IAPR-TC-15 International Workshop, GbRPR 2017*, volume 10310 of *LNCS*, 2017.

[Chaplick *et al.*, 2015] Steven Chaplick, Jirí Fiala, Pim van 't Hof, Daniël Paulusma, and Marek Tesar. Locally constrained homomorphisms on graphs of bounded treewidth and bounded degree. *Theor. Comput. Sci.*, 590:86–95, 2015.

[Corby and Faron-Zucker, 2007] Olivier Corby and Catherine Faron-Zucker. Implementation of SPARQL query language based on graph homomorphism. In *Conceptual*

Structures: Knowledge Architectures for Smart Applications, 15th International Conference on Conceptual Structures, ICCS 2007, volume 4604 of *LNCS*. Springer, 2007.

[Cordella *et al.*, 2004] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[Dyer and Greenhill, 2000] Martin E. Dyer and Catherine S. Greenhill. The complexity of counting graph homomorphisms. *Random Struct. Algorithms*, 17(3-4):260–289, 2000.

[Ehrlich and Rarey, 2011] Hans-Christian Ehrlich and Matthias Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(1):68–79, 2011.

[Fan *et al.*, 2010] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proc. VLDB Endow.*, 3(1), 2010.

[Fiala and Kratochvíl, 2008] Jirí Fiala and Jan Kratochvíl. Locally constrained graph homomorphisms - structure, complexity, and applications. *Comput. Sci. Rev.*, 2(2):97–111, 2008.

[Fiala *et al.*, 2001] Jirí Fiala, Ton Kloks, and Jan Kratochvíl. Fixed-parameter complexity of λ-labelings. *Discret. Appl. Math.*, 113(1):59–72, 2001.

[Foggia *et al.*, 2014] Pasquale Foggia, Gennaro Percannella, and Mario Vento. Graph matching and learning in pattern recognition in the last 10 years. *IJPRAI*, 28(1), 2014.

[Fomin *et al.*, 2007] Fedor V. Fomin, Pinar Heggernes, and Dieter Kratsch. Exact algorithms for graph homomorphisms. *Theory Comput. Syst.*, 41(2):381–393, 2007.

[Garey and Johnson, 1979] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Gocht *et al.*, 2020a] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020*, 2020.

[Gocht *et al.*, 2020b] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 2020.

[Kotthoff *et al.*, 2016] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. Portfolios of subgraph isomorphism algorithms. In *Learning and Intelligent Optimization - 10th International Conference, LION 10*, 2016.

[McConnell *et al.*, 2011] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011.

[McCreesh and Prosser, 2014] Ciaran McCreesh and Patrick Prosser. Reducing the branching in a branch and bound algorithm for the maximum clique problem. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014*, volume 8656 of *LNCS*. Springer, 2014.

[McCreesh and Prosser, 2015] Ciaran McCreesh and Patrick Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Proceedings*, 2015.

[McCreesh *et al.*, 2018] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *J. Artif. Intell. Res.*, 61:723–759, 2018.

[McCreesh *et al.*, 2020] Ciaran McCreesh, Patrick Prosser, and James Trimble. The Glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In *Graph Transformation - 13th International Conference, ICGT 2020*, 2020.

[Prosser, 2012] Patrick Prosser. Exact algorithms for maximum clique: A computational study. *Algorithms*, 5(4):545–587, 2012.

[Rzazewski, 2014] Pawel Rzazewski. Exact algorithm for graph homomorphism and locally injective graph homomorphism. *Inf. Process. Lett.*, 114(7):387–391, 2014.

[San Segundo *et al.*, 2013] Pablo San Segundo, Fernando Matía, Diego Rodríguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optim. Lett.*, 7(3):467–479, 2013.

[Solnon, 2010] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, 2010.

[Solnon, 2019] Christine Solnon. Experimental evaluation of subgraph isomorphism solvers. In *Graph-Based Representations in Pattern Recognition - 12th IAPR-TC-15 International Workshop, GbRPR 2019*, 2019.

[Tomita *et al.*, 2010] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and Computation, 4th International Workshop, WALCOM 2010*, volume 5942 of *LNCS*. Springer, 2010.

[Willett, 1999] Peter Willett. Matching of chemical and biological structures using subgraph and maximal common subgraph isomorphism algorithms. In Donald G. Truhlar, W. Jeffrey Howe, Anthony J. Hopfinger, Jeff Blaney, and Richard A. Dammkoehler, editors, *Rational Drug Design*, pages 11–38. Springer New York, New York, NY, 1999.

[Zampelli *et al.*, 2010] Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010.