Finding Subgraphs with Side Constraints

Özgür Akgün[1], Jessica Enright[2], Christopher Jefferson[1], Ciaran McCreesh[2](✉), Patrick Prosser[2], and Steffen Zschaler[3]

[1] University of St Andrews, St Andrews, Scotland
[2] University of Glasgow, Glasgow, Scotland
ciaran.mccreesh@glasgow.ac.uk
[3] King's College London, London, UK

Abstract. The subgraph isomorphism problem is to find a small "pattern" graph inside a larger "target" graph. There are excellent dedicated solvers for this problem, but they require substantial programming effort to handle the complex side constraints that often occur in practical applications of the problem; however, general purpose constraint solvers struggle on more difficult graph instances. We show how to combine the state of the art Glasgow Subgraph Solver with the Minion constraint programming solver to get a "subgraphs modulo theories" solver that is both performant and flexible. We also show how such an approach can be driven by the Essence high level modelling language, giving ease of modelling and prototyping to non-expert users. We give practical examples involving temporal graphs, typed graphs from software engineering, and costed subgraph isomorphism problems.

1 Introduction

Finding small "pattern" graphs inside larger "target" graphs is a widely applicable hard problem, with applications including compilers [5], bioinformatics [6,16], chemistry [29], malware detection [8], pattern recognition [17], and the design of mechanical locks [35]. This has led to the development of numerous dedicated algorithms, with the Glasgow Subgraph Solver [24] being the current state of the art [33]. However, practitioners are often interested in versions of the problem with additional restrictions, or side constraints. Some of these, such as exact vertex labelling schemes, are trivial to include in a dedicated solver, but others currently require either extensive programming or inefficient post-processing. This paper explores a different approach: by allowing the Glasgow Subgraph Solver to use the Minion constraint programming (CP) solver [19] for side constraints, we achieve both the performance only a dedicated solver can offer, with the flexibility of a full CP toolkit. This hybrid modelling system can be driven by the Essence high level modelling language [18] and the Conjure toolchain, making it accessible to non-specialists.

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

© Springer Nature Switzerland AG 2021
P. J. Stuckey (Ed.): CPAIOR 2021, LNCS 12735, pp. 348–364, 2021.
https://doi.org/10.1007/978-3-030-78230-6_22

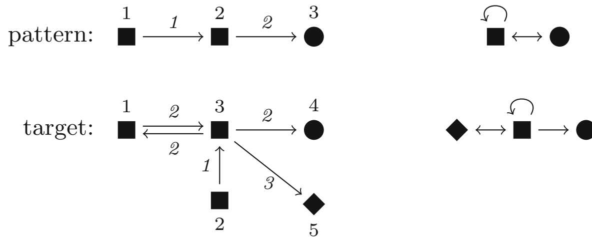

Fig. 1. A small pattern and a larger target graph used in examples throughout this paper. The plain text numbers are vertex names, and the shapes on vertices represent vertex labels. The graphs to the right are *type graphs*, which are used in Sect. 3.1. The italic labels on edges are used for *temporal graphs*, which are discussed in Sect. 3.2, and should otherwise be ignored.

1.1 Preliminaries

We begin with a look at the subgraph isomorphism problem, from a high level constraint modelling perspective. The basic non-induced subgraph isomorphism problem is to find an injective mapping from a pattern graph to a target graph, such that adjacent vertices in the pattern are mapped to adjacent vertices in the target. Variations on the problem are common, and are often combined. For example, in the induced version of the problem, non-edges must be mapped to non-edges; in the directed version, the input graphs have directed edges whose orientations must be preserved by the mapping; in the vertex labelled version, each vertex has a label, and the mapping must map vertices to like-labelled vertices; and in the edge-labelled version, edges have labels which must be preserved. It is also common to want to count or enumerate all solutions, rather than deciding whether at least one solution exists. Subsets of these variations are supported by many dedicated subgraph isomorphism algorithms, including the Glasgow Subgraph Solver.

We can express these problems in the Essence modelling language, as follows. We assume vertices take their labels from the set $L = \{1 \ldots \ell\}$ for some given ℓ, and edges from $E = \{1 \ldots e\}$ (and so ℓ and / or e may be 1, for applications that do not use labels on vertices and/or edges):

```
given l, e : int
letting L be domain int(1..l)
letting E be domain int(1..e)
```

We take as input a directed pattern graph which has p vertices (which we number from 1 to p, in the set P), and a directed target graph which has t vertices (numbered from 1 to t, the set T). Each graph is represented as total function from vertices to vertex labels, and a *partial* function from pairs of (not necessarily distinct) vertices to edge labels:

```
given p, t : int
letting P be domain int(1..p)
letting T be domain int(1..t)

given pat : function (P, P) --> E
given tgt : function (T, T) --> E
given plab : function (total) P --> L
given tlab : function (total) T --> L
```

Now we wish to find an injective mapping f:

```
find f : function (total, injective) P --> T
```

that preserves vertex labels,

```
such that forAll a : P .  plab(a) = tlab(f(a))
```

and directed edges, including their labels:

```
such that forAll ((a, b), lbl) in pat .
    ((f(a), f(b)), lbl) in toSet(tgt)
```

As a simple example, the following inputs show the problem instance represented in Fig. 1. We have three different vertex labels (circle, square, and diamond), and only a single edge type (which is directed; the numerical labels on edges are not used in this section):

```
letting l be 3
letting e be 1
```

We may now describe the pattern:

```
letting p be 3
letting pat be function ((1, 2) --> 1, (2, 3) --> 1)
letting plab be function (1 --> 1, 2 --> 1, 3 --> 2)
```

and the target:

```
letting t be 5
letting tgt be function ((1, 3) --> 1, (3, 1) --> 1,
  (2, 3) --> 1, (3, 4) --> 1, (3, 5) --> 1)
letting tlab be function (1 --> 1, 2 --> 1, 3 --> 1,
  4 --> 2, 5 --> 3)
```

Using the Conjure tool to compile Essence to a constraint programming model which is then solved by Minion, we find there are exactly two solutions to the problem, as we would expect:

```
(1 --> 1, 2 --> 3, 3 --> 4)
(1 --> 2, 2 --> 3, 3 --> 4)
```

But what if our application requires induced isomorphisms? Then we can easily add the constraint

```
such that forAll (a, b) : (P, P) .
    (f(a), f(b)) in defined(tgt) -> (a, b) in defined(pat)
```

And Conjure will now find us a single solution,

```
(1 --> 2, 2 --> 3, 3 --> 4)
```

As we will see in Sect. 3, supporting other problem variants and constraints is similarly straightforward, even if auxiliary variables are required. For example, if instead we want to allow relabelling on vertex labels (which is typically not supported by dedicated solvers), we could do the following:

```
find r : function (total, injective) L --> L
such that forAll a : P .  r(plab(a)) = tlab(f(a))
```

and we would find two additional solutions,

```
(1 --> 1, 2 --> 3, 3 --> 5)
(1 --> 2, 2 --> 3, 3 --> 5)
```

and if we removed the injective keyword for the relabelling, we would find a fifth mapping

```
(1 --> 2, 2 --> 3, 3 --> 1)
```

We return to relabelling in Sect. 3.1.

1.2 Initial Experiments and Motivation

Unfortunately, whilst elegant and flexible, the performance of this approach leaves a lot to be desired on basic subgraph isomorphism instances. The computational experiments in this paper are performed on a cluster of machines with dual Intel Xeon E5-2697A v4 processors and 512GBytes RAM running Ubuntu 18.04. The source code used for these experiments is released as part of the Glasgow Subgraph Solver[1], Minion[2], and Conjure[3] distributions, and we provide a separate archive for experimental scripts[4].

For graphs, we will be using the 14,621 unlabelled, undirected instances from Solnon's benchmark suite[5]. This benchmark suite was originally designed for algorithm portfolios work [21], and brings together several collections of application and randomly-generated instances with varying difficulties and solution counts (including many unsatisfiable instances). Some of the instances have up to 900 vertices and 14,420 edges in patterns and up to 6,671 vertices and 209,000 edges in targets. These lead to rather large models, by constraint programming standards: the largest generated table constraint has nearly half a million entries. However, these sizes are realistic from an applications perspective, and it would be desirable if solvers could handle even larger target graphs.

In Fig. 2 we plot the cumulative number of instances solved over time for the non-induced decision problem, comparing the high level approach to the

[1] https://github.com/ciaranm/glasgow-subgraph-solver/releases/tag/cpaior2021-finding-subgraphs-with-side-constraints.

[2] https://github.com/minion/minion/releases/tag/1.9.

[3] https://github.com/conjure-cp/conjure.

[4] https://github.com/ciaranm/cpaior2021-finding-subgraphs-with-side-constraints.

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

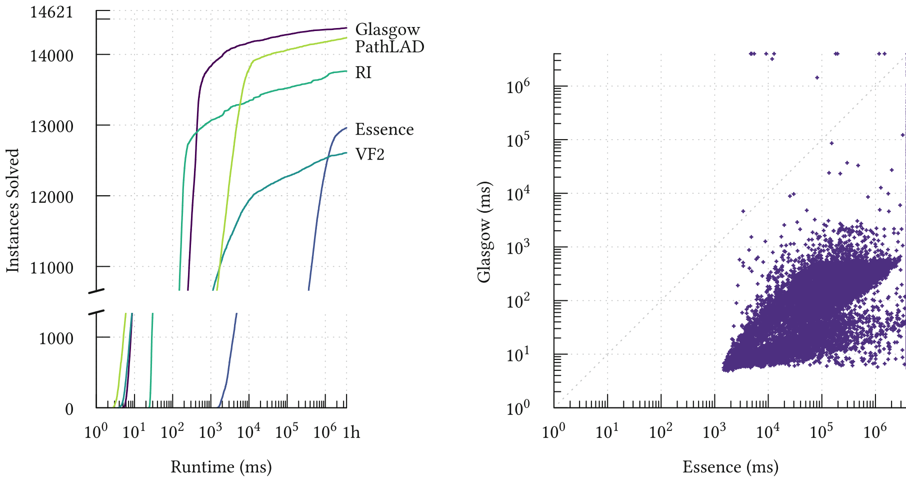

Fig. 2. Left, the cumulative number of instances solved over time, for the non-induced decision problem with no side constraints. Right, comparing the high level approach with the Glasgow Subgraph Solver on an instance by instance basis; points on the outer axes represent timeouts.

Glasgow Subgraph Solver [24] and PathLAD [21] (the two strongest CP-inspired approaches), and to VF2 [10] and RI [6] (simpler algorithms which perform well on easy instances). The high level approach has very slow startup times (which is to be expected as it involves launching a Java virtual machine and reading in a very large table constraint), but much more worryingly, only catches up with the worst other solver in number of instances solved as the timeout approaches. Worse, as the scatter plot in Fig. 2 shows, there are almost no instances where the high level approach does better than the Glasgow Subgraph Solver. (For induced problems, the results are even less favourable).

These first results motivate the remainder of this paper. We want to retain the convenience of the high level modelling approach, and to be able to add arbitrary side constraints to suit different applications, but we do not want to have to abandon the performance that dedicated solvers can get on hard instances. In Sect. 2 we evaluate several ways of using a CP solver in conjunction with the Glasgow Subgraph Solver, with a focus on low level implementation details. In Sect. 3 we then return to high level modelling, and look at the convenience it provides for retyping typed graphs, for temporal problems, and for optimisation problems.

2 Hybrid Solving with High-Level Modelling

Designing an effective hybrid solving system involved three major decisions: how the high-level modelling language would identify suitable problems to hybridise, how the solvers would communicate, and how often this communication would

occur. The first two decisions were relatively straightforward to make, but the third required using computational experiments to evaluate different options. This section discusses all three of these decisions.

2.1 High-Level Modelling

We chose to use the Essence modelling language [18] because of its support for convenient high-level types like functions and relations, which can easily describe graphs and related abstractions. In a conventional modelling pipeline, problems are specified in Essence and then are converted to concrete models that can be solved by a CP solver (in our experiments Minion) via the Conjure and SavileRow tools. We augmented Conjure with a command line option that instructs it to generate an extra file which describes how the variables representing the graph are represented in the SavileRow input (known as Essence'). This is used by the graph solver so it can map between its internal state and the variables in Essence'. SavileRow converts the graph model to Minion input, and this conversion contains information which allows Minion to map its internal state back to the Essence' given to SavileRow. The graph solver and Minion then communicate mappings between the nodes and edges in the graph using the identifiers in the Essence' representation of the problem.

This design is based upon the notion that a CP solver and a subgraph solver can have enough of a shared understanding of a problem to solve it co-operatively. Indeed, the Glasgow Subgraph Solver [24] employs a CP approach to solve subgraph-finding problems, but using special data structures and algorithms—for example, rather than representing the adjacency constraint using an explicit table, it uses bitset adjacency matrices [22]. The solver also exploits various graph invariants involving degrees [36] and paths [2] to further reduce the search space, and employs special search order heuristics [1]. From this paper's perspective, the most important design aspect is that internally, the solver has a CP style variable for each vertex in the pattern graph, whose domains range over the vertices of the target graph. The solver performs a backtracking search with restarts and nogood recording, attempting to assign each variable a value from its domain, whilst respecting adjacency and injectivity constraints. At each recursive call of search, the solver performs *propagation* to eliminate infeasible values from domains. If any domain becomes empty, the solver backtracks; otherwise, it selects a variable, and tries assigning it each value from its domain in turn.

The high-level approach, then, gives us a way of setting up the subgraph solver and a CP solver such that they both have an equivalent set of variables and values for the graph part of the model, and tells us how to form a correspondence between their internal representations. Importantly, this allows the CP solver to have additional variables that the subgraph solver does not know about, and we do not specifically require the CP solver to be aware of all of the graph constraints. (Additionally, due to preprocessing, the CP solver may also sometimes have only a subset of values for some graph variables visible to it).

2.2 When to Communicate?

Having found a way to set up the two solvers, we must next ask when they should communicate. The simplest approach would be to use the CP solver as a *solution checker* for the subgraph solver. Whenever the subgraph solver finds a solution, it will pass it to the CP solver, which will treat the solution as a set of equality constraints. The CP solver will then attempt to find a satisfying assignment. If the CP solver does not have any additional variables, this is equivalent to simply checking that the remaining constraints hold, but in general this will require search. For a decision problem, the CP solver then communicates back to the subgraph solver either "yes, this is a valid solution", or "no, reject this solution and keep going". If we are solving a counting or enumeration problem, the CP solver must find *all* solutions and communicate this back to the subgraph solver.

For more power, but possibly also greater cost, we could additionally ask a CP solver at every stage of search to test whether the subgraph solver is in an obviously infeasible state. Whenever the subgraph solver has finished performing propagation, it can communicate the *trail* (that is, its current sequence of guessed assignments) to the CP solver, which again treats these as additional equality constraints. The CP solver then performs its own propagation (but not search), and communicates back either a "yes, keep going" or a "no, backtrack immediately". Finally, after this testing, we could also ask the CP solver to communicate any deletions it infers back to the subgraph solver. In other words, the subgraph solver would use the CP solver as an additional propagator.

Unfortunately, each of these approaches has drawbacks. The solution we will settle upon is based upon *rollbacks*; we will describe this below, after presenting experiments that demonstrate the difference between these approaches.

2.3 How to Communicate

To enable communication between the two solvers, we use FIFOs (named pipes), and a simple text-based protocol. Both solvers are run and initialised, and then the subgraph solver proceeds as normal, whilst the CP solver waits to be given commands. The subgraph solver then communicates its trail or a candidate solution as a set of assignment constraints to the CP solver. The CP solver then sets these assignments as its state and either performs a single propagation, or complete search, as directed. When finished the solver communicates its success or failure state, and any deletions (if requested), back to the subgraph solver and reverts any changes made by setting the assignments. This approach is designed to be solver-agnostic, and adding support for different CP solvers (or non-CP solvers) is simple, as long as they support performing a search or propagation from a given set of assignments.

2.4 Design Experiments

We now present the results of some computational experiments. The experiments in this section are designed to be hard, and to emphasise the difference

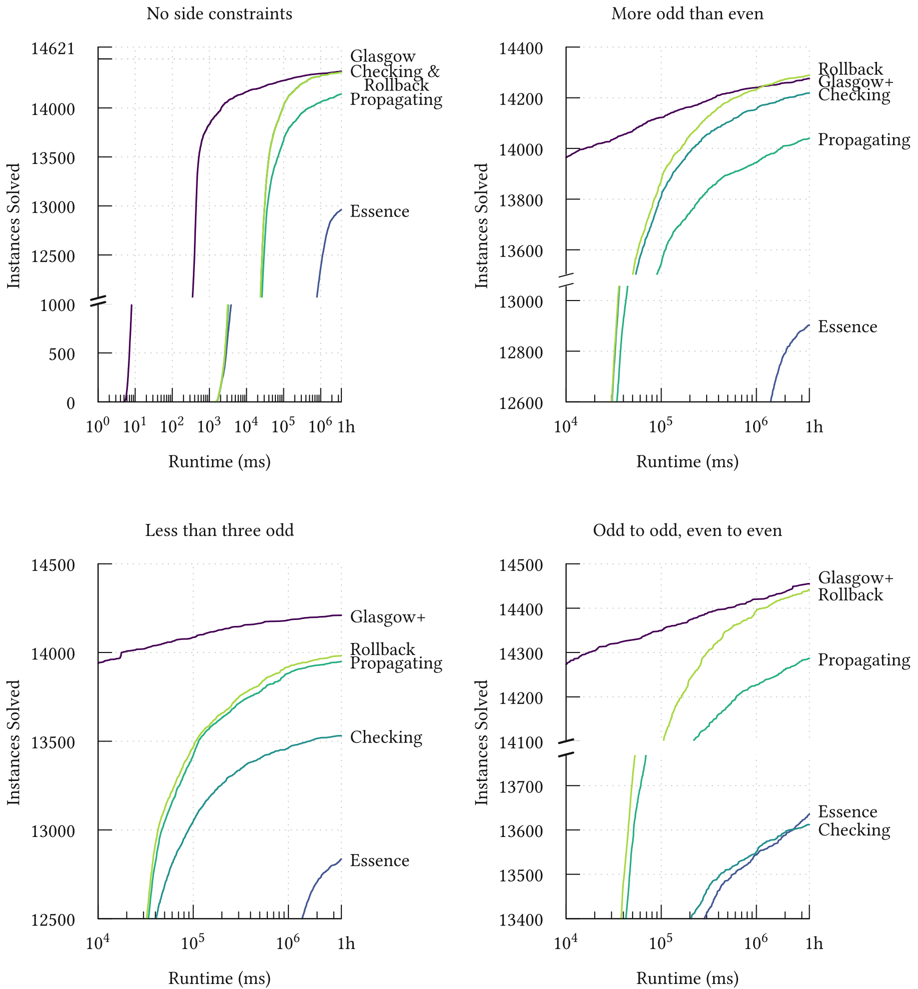

Fig. 3. Comparing different approaches to hybrid solving, showing the cumulative number of instances solved over time. On the top row, "no side constraints" then with the "more odd target vertices than even target vertices" side constraint; on the bottom row, the "mostly odd target vertices" side constraint on the left, and the "odd to odd, even to even" side constraint on the right. In the top left plot, the "Checking" and "Rollback" lines are indistinguishable. The Glasgow+ lines show the Glasgow Subgraph Solver with manually-implemented side constraints.

between the approaches, rather than to be realistic. We will continue to work with Solnon's non-induced subgraph isomorphism benchmark instances, but will consider four variations. Firstly, we will consider the problem with no side constraints. This is, in some sense, the worst case scenario, where we must pay the full price of hybrid solving, but cannot get any benefit from it. Secondly, let us

say that the number of odd-indexed target vertices used must be greater than the number of even target vertices used:

```
such that (sum i : P . f(i) % 2) >
    (sum i : P . (f(i) + 1) % 2)
```

Thirdly, let us instead say that fewer than three odd target vertices may be used:

```
such that (sum i : P . f(i) % 2) < 3
```

And fourthly, let us say that even pattern vertices must be mapped to even target vertices, and odd pattern vertices to odd target vertices:

```
such that forAll a : P . (a % 2) = (f(a) % 2)
```

We chose these problems because the second is likely not to be able to perform inference until deep in a search tree (when most pattern vertices are mapped to specific target vertices), the third is likely to be able to perform inference early in the search tree (after a few assignments have been made, but not at the root node), and the fourth should propagate only at the root node.

The results of these experiments are presented in Fig. 3. Let us first look at the top left plot, where we do not actually have any side constraints. When calling the CP solver as a solution checker, we ultimately achieve the same performance as the Glasgow Subgraph Solver[6], although we can pay a substantial startup overhead. This should not be surprising: with no side constraints, the subgraph solver runs as normal, and will perform just one call to the CP solver on satisfiable instances. The propagating approach is over an order of magnitude slower: calling the CP solver at every search node is clearly very expensive. (We also tried calling the CP solver to test feasibility, without communicating deletions; this made no noticeable difference to performance, and so is not pictured.) Finally, all approaches substantially outperform using a CP solver on its own without help from the subgraph solver.

What about the remaining three plots in Fig. 3, where we do have side constraints? As we hoped, we see differences between the three plots. On the top right, where we expect the side constraints to fire late, solution checking clearly beats propagation during search. However, on the bottom left, where we expect side constraints to fire early, the propagating approach is much better than solution checking. Meanwhile, on the bottom right, where our constraints fire only at the root node, checking performs extremely poorly compared to propagating. In each case, any hybrid approach using the subgraph solver remains much better than a pure CP approach, except that in bottom right plot checking is slightly worse than just using the CP solver on its own.

As a point of comparison, we also implemented these three sets of side constraints natively inside the Glasgow solver: we label these as Glasgow+. For the "more odd than even" case, we did this through solution checking; for "less than three odd" we implemented checking during search; and for "odd to odd, even to even" we implemented initial domain filtering. Our implementation choices here

[6] Actually, because high level modelling can use different names for variables and values, we get slight differences due to changes to tiebreaking in search order heuristics.

are intended to reflect what a reasonable programmer would do if the high level approach were not available (and we intentionally selected side constraints that would not be too difficult to implement). In two of the three cases, the hand-crafted code does somewhat outperform the hybrid solving, but in the "more odd than even" case, hybrid solving actually beats the hand-crafted dedicated solver implementation when using the rollback approach, which we now describe.

2.5 A Rollback Approach to Communication

From what we have seen so far, it is obviously important to call the CP solver *some* of the time during search, but too expensive to call it *all* of the time. We will therefore introduce a new approach, which we call *rollback*. This approach is inspired by backjumping [27], as well as by the conflict analysis methods used in SAT and SMT solvers [32] and in lazy clause generating CP solvers [25,34]. The idea is as follows. Firstly, we call the CP solver with full propagation at the root node, in case we are dealing with a particularly rich labelling scheme. Secondly, we use the CP solver for solution checking, since this is required for correctness. Now, suppose the CP solver rejects a candidate solution: this will cause the subgraph solver to backtrack. At this point, we call the CP solver again, with full propagation. Either the CP solver indicates feasibility, in which case we proceed with search (potentially with a reduced set of domains), or the CP solver indicates failure, in which case we backtrack again, and do another attempt at full propagation, and so on until feasibility is reached.

The idea behind this approach is to avoid calling the CP solver when it is unlikely to do anything useful, but that once a failure has been encountered, we want to extract as much information as we can from the CP solver. If the failure encountered was due to a "local" property of the solution, such as in the "more odd than even" example, then we will quickly return to just using the subgraph solver for search. However, if the failure is due to only a few early assignments, as in the "fewer than three odd vertices" example, then we will jump back to nearly the root of the search tree.

The results in Fig. 3 demonstrate the success of this approach. When there are no side constraints, this approach has no overheads compared to solution checking. When constraints fire late, this approach is better than solution checking, and when constraints fire early, this approach is better than always propagating during search. In other words, rolling back from failures gives us all of the strengths and none of the weaknesses of the simpler approaches. We will therefore use this method for the remainder of the paper.

3 Subgraph Problems with Side Constraints

We now look at three classes of real-world subgraph-finding problems that, until now, have been solved using dedicated approaches. We show how easy it is to model these problems in Essence, demonstrating the usefulness of the high-level modelling approach for prototyping and development.

3.1 Retyping Problems

The basic notion of a graph conveys only adjacency information, and a subgraph isomorphism simply finds a certain structural pattern. In practice, this is often augmented with additional information—for example, we have seen how labels can be associated with vertices and edges, which can be used in chemical applications to represent different kinds of atom or bond. In this case, subgraph isomorphisms are also expected to preserve labels, so carbon atoms can only be mapped to carbon atoms, and double bonds must be mapped to double bonds. A richer labelling abstraction comes in the form of *typed graphs*, where the labels themselves also carry a graph structure [15]; we show an example in Fig. 1. In practice this labelling structure is specified by providing two graphs, together with a morphism from the main graph to its type graph.

For typed graphs, morphisms between the graphs are typically defined between graphs typed over the same type graph, but there are situations where we are interested in mapping between graphs typed over different type graphs. One such scenario from a software engineering context is described by Durán et al. [12,13], where graph transformation systems are composed by defining morphisms between the rules constituting the respective transformation systems. In this case, the source and target transformation systems will normally have different type graphs; a morphism must also be established between the two type graphs. Mappings between the various graphs making up the rules then need to preserve structure and typing subject to the morphism between type graphs. This approach to specification composition is implemented by the GTSMorpher tool.[7] A key objective is to minimise the amount of specification that needs to be written. For example, the tool allows morphisms between graph transformation systems to be only partially specified and then automatically completes the full morphism, if it can do so unambiguously—this requires solving a subisomorphism problem.

We may describe typed graph subisomorphism problems in Essence as follows. As before, we are given a pattern graph and a target graph, both of which carry labels; we will draw the vertex labels from different sets, to emphasise the relabelling.

```
given pl, tl, e : int
letting PL be domain int(1..pl)
letting TL be domain int(1..tl)
letting E be domain int(1..e)
```

We are also given labelled graphs,

```
given p, t : int
letting P be domain int(1..p)
letting T be domain int(1..t)

given pat : function (P, P) --> E
given tgt : function (T, T) --> E
```

[7] https://github.com/gts-morpher/gts_morpher.

```
given plab : function (total) P --> PL
given tlab : function (total) T --> TL
```

but now the labels also carry a graph structure,

```
given pattype : function (PL, PL) --> E
given tgttype : function (TL, TL) --> E
```

We are looking for an injective mapping from the pattern graph to the target graph,

```
find f : function
    (total, injective) P --> T
```

as well as an injective mapping between the label graphs,

```
find r : function
    (total, injective) PL --> TL
```

in such a way that graph structure and labels are preserved,

```
such that forAll ((a, b), lbl) in pat .
    ((f(a), f(b)), lbl) in toSet(tgt)
such that forAll a : P .
    r(plab(a)) = tlab(f(a))
```

and also requiring that the structure on the labels is preserved,

```
such that forAll (a,b) in defined(pattype) .
    pattype((a,b)) = tgttype((r(a),r(b)))
```

Consider again the example in Fig. 1, and now suppose they are equipped with the type structures shown to the right of each graph,

```
letting pattype be function (
  (1, 1) --> 1, (1, 2) --> 1, (2, 1) --> 1 )
letting tgttype be function (
  (1, 1) --> 1, (1, 2) --> 1,
  (1, 3) --> 1, (3, 1) --> 1 )
```

We now find two solutions:

```
(1 --> 1, 2 --> 3, 3 --> 5)
(1 --> 2, 2 --> 3, 3 --> 5)
```

because we can map pattern vertex 3 to target vertex 5 through retyping, but mapping pattern vertex 3 to target vertex 4 would not respect the type graph structure.

More generally, the field of model-driven software engineering includes numerous examples of using search and optimisation techniques to generate or transform graphs [7]. Existing approaches largely make use of ad-hoc [31] and metaheuristic methods [4,9,14], but we believe that with the help of suitably accessible high-level modelling tools, this could become a fruitful area for constraint programming research in the future.

3.2 Temporal Subgraph Problems

Another labelling scheme is used in *temporal graphs*, where edges are labelled with timestamps that denote times when edges are active—here we use integers as timestamps. Including information on the timing of edges substantially increases the modelling power of these graphs, allowing them to more accurately reflect the structure and dynamics of a wide variety of real-world systems (e.g. trade networks, changing contact networks, transport networks), and address optimisation questions in which the timing of edges is fundamental.

As algorithms and formalisms have become available for temporal graphs, examples of their application have become widespread [20], notably including applications within epidemiology [3] and computational social science [30]. Because the use of temporal graphs has spread beyond theoretical researchers, the ability to rapidly define and experiment with new problem definitions and constraints is valuable—practitioners are unlikely to define bespoke algorithms for novel problems as they arise.

There are at least three common kinds of temporal subgraph isomorphism. In an *exact* subisomorphism, times are simply labels that must match exactly. If we look at Fig. 1, now ignoring vertex labels but using the edge labels to carry the timestamps,

```
letting l be 1
letting e be 3

letting p be 3
letting pat be function ((1, 2) --> 1, (2, 3) --> 2)
letting plab be function (1 --> 1, 2 --> 1, 3 --> 1)

letting t be 5
letting tgt be function ((1, 3) --> 2, (3, 1) --> 2,
  (2, 3) --> 1, (3, 4) --> 2, (3, 5) --> 3)
letting tlab be function (1 --> 1, 2 --> 1, 3 --> 1,
  4 --> 1, 5 --> 1 )
```

then there are two solutions,

```
(1 --> 2, 2 --> 3, 3 --> 1)
(1 --> 2, 2 --> 3, 3 --> 4)
```

A less strict kind of subisomorphism is an *offset*, where edge labels must match exactly, but offset by an integer constant. In our example, this means "find a mapping where the event from 2 to 3 occurs one time unit after the event from 1 and 2". We can model this as follows:

```
find o : int(-e..e)
such that forAll (a,b) in defined(pat) .
    pat((a,b)) = o + tgt((f(a), f(b)))
```

and we find one additional solution,

```
(1 --> 1, 2 --> 3, 3 --> 5)
```

Finally, in an *order* embedding, the pattern edge labels simply define an order on events. We can model this as follows:

```
find o : function (total) E --> E
such that forAll x : int(1..e - 1) .  o(x) <= o(x + 1)
such that forAll (a,b) in defined(pat) .
    pat((a,b)) = o(tgt((f(a), f(b))))
```

which gets us yet another solution,

```
(1 --> 2, 2 --> 3, 3 --> 5)
```

Of course, when using a high level modelling approach, we are not restricted to these three problem variants, and could easily try out new models in an interactive setting. For example, it would take only a few minutes to write a model for a temporal problem where all edges must occur within a short but unspecified time period [28], whereas adapting a dedicated solver to check this constraint would be a substantial programming effort (and making the solver propagate rather than check this constraint would be even harder).

3.3 Subgraph Isomorphism with Costs

The system we created also support optimisation problems (and does not require that the subgraph isomorphism solver be aware that this is what is going on). If, for example, each target vertex has a cost associated with it,

```
given tcost : function (total) T --> int
```

then we can ask to find the cheapest solution,

```
minimising sum([ tcost(f(a)) | a : P])
```

We could also just as easily ask for the solution whose most expensive edge is cheapest, or that uses fewest vertices with a particular label. These kinds of problem occur widely in practice, including in skyline graph queries [26], labelled subgraph finding [11], and weighted clique problems [23].

4 Conclusion

The system we have presented shows that it is possible to combine the power of modern subgraph solvers with the flexibility of a general purpose constraint programming toolkit, although doing so efficiently requires careful consideration of how frequently the solvers communicate. We believe further research in this direction may be useful—for example, would it be possible to make use of some kind of conflict analysis rather than a backjumping approach?

When driven by a high level modelling approach, this system is particularly suitable for rapid prototyping and for dynamic queries where side constraints can be specified in response to user need. However, the high level modelling approach does come with a large startup cost, which makes it unsuitable for deployment in application contexts that involve solving many thousands of problem instances

in real-time. Fortunately though, connecting the low level solvers manually is also an option once a design has been decided upon. We also expect that new approaches may be necessary to deal with the huge but sparse graphs that arise in some applications, since table constraints and conventional CP domain stores both struggle when moving beyond ten thousand of vertices in target graphs.

References

1. 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
2. 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.) CP 2014. LNCS, vol. 8656, pp. 125–141. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_12
3. Bansal, S., Read, J., Pourbohloul, B., Meyers, L.A.: The dynamic nature of contact networks in infectious disease epidemiology. J. Biol. Dyn. **4**(5), 478–489 (2010)
4. Bill, R., Fleck, M., Troya, J., Mayerhofer, T., Wimmer, M.: A local and global tour on MOMoT. Softw. Syst. Model. **18**(2), 1017–1046 (2017). https://doi.org/10.1007/s10270-017-0644-3
5. Hjort Blindell, G., Castañeda Lozano, R., Carlsson, M., Schulte, C.: Modeling universal instruction selection. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 609–626. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_42
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. Boussaïd, I., Siarry, P., Ahmed-Nacer, M.: A survey on search-based model-driven engineering. Autom. Softw. Eng. **24**(2), 233–294 (2017). https://doi.org/10.1007/s10515-017-0215-4
8. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 129–143. Springer, Heidelberg (2006). https://doi.org/10.1007/11790754_8
9. Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic consistency preserving search operators for search-based model engineering. In: Kessentini, M., Yue, T., Pretschner, A., Voss, S., Burgueño, L. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, 15–20 September 2019, pp. 106–116. IEEE (2019). https://doi.org/10.1109/MODELS.2019.00-10
10. 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)
11. Dell'Olmo, P., Cerulli, R., Carrabs, F.: The maximum labeled clique problem. In: Adacher, L., Flamini, M., Leo, G., Nicosia, G., Pacifici, A., Piccialli, V. (eds.) Proceedings of the 10th Cologne-Twente Workshop on Graphs and Combinatorial Optimization. Extended Abstracts, Villa Mondragone, Frascati, Italy, 14–16 June 2011, pp. 146–149 (2011)

12. Durán, F., Moreno-Delgado, A., Orejas, F., Zschaler, S.: Amalgamation of domain specific languages with behaviour. J. Log. Algebraic Methods Program. **86**, 208–235 (2017). https://doi.org/10.1016/j.jlamp.2015.09.005
13. Durán, F., Zschaler, S., Troya, J.: On the reusable specification of non-functional properties in DSLs. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 332–351. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36089-3_19
14. Efstathiou, D., Williams, J.R., Zschaler, S.: Crepe complete: multi-objective optimization for your models. In: Paige, R.F., Kessentini, M., Langer, P., Wimmer, M. (eds.) Proceedings of the First International Workshop on Combining Modelling with Search- and Example-Based Approaches co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, 28 September 2014. CEUR Workshop Proceedings, vol. 1340, pp. 25–34. CEUR-WS.org (2014)
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCSSeries. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
16. Elhesha, R., Sarkar, A., Kahveci, T.: Motifs in biological networks. In: Yoon, B.-J., Qian, X. (eds.) Recent Advances in Biological Network Analysis, pp. 101–123. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-57173-3_5
17. Foggia, P., Percannella, G., Vento, M.: Graph matching and learning in pattern recognition in the last 10 years. IJPRAI **28**(1), 1450001 (2014). https://doi.org/10.1142/S0218001414500013
18. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence: a constraint language for specifying combinatorial problems. Constraints Int. J. **13**(3), 268–306 (2008). https://doi.org/10.1007/s10601-008-9047-y
19. Gent, I.P., Jefferson, C., Miguel, I.: Minion: a fast scalable constraint solver. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) ECAI 2006, 17th European Conference on Artificial Intelligence, 29 August–1 September 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings. Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 98–102. IOS Press (2006)
20. Holme, P., Saramäki, J.: Temporal networks. Phys. Rep. **519**(3), 97–125 (2012). https://doi.org/10.1016/j.physrep.2012.03.001
21. 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
22. McCreesh, C., Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 295–312. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_21
23. McCreesh, C., Prosser, P., Simpson, K., Trimble, J.: On maximum weight clique algorithms, and how they are evaluated. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 206–225. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_14
24. McCreesh, C., Prosser, P., Trimble, J.: The glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In: Gadducci, F., Kehrer, T. (eds.) ICGT 2020. LNCS, vol. 12150, pp. 316–324. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51372-6_19

25. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = lazy clause generation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 544–558. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_39
26. Pande, S., Ranu, S., Bhattacharya, A.: SkyGraph: retrieving regions of interest using skyline subgraph queries. Proc. VLDB Endow. **10**(11), 1382–1393 (2017). https://doi.org/10.14778/3137628.3137647
27. Prosser, P.: Hybrid algorithms for the constraint satisfaction problem. Comput. Intell. **9**, 268–299 (1993). https://doi.org/10.1111/j.1467-8640.1993.tb00310.x
28. Redmond, U., Cunningham, P.: Temporal subgraph isomorphism. In: Rokne, J.G., Faloutsos, C. (eds.) Advances in Social Networks Analysis and Mining 2013, ASONAM 2013, Niagara, ON, Canada, 25–29 August 2013, pp. 1451–1452. ACM (2013). https://doi.org/10.1145/2492517.2492586
29. Régin, J.: Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique. Ph.D. thesis, Université Montpellier 2 (1995)
30. Sekara, V., Stopczynski, A., Lehmann, S.: Fundamental structures of dynamic social networks. Proc. Natl. Acad. Sci. **113**(36), 9977–9982 (2016)
31. Semeráth, O., Nagy, A.S., Varró, D.: A graph solver for the automated generation of consistent domain-specific models. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 969–980. ACM (2018). https://doi.org/10.1145/3180155.3180186
32. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, 10–14 November 1996, pp. 220–227. IEEE Computer Society/ACM (1996). https://doi.org/10.1109/ICCAD.1996.569607
33. 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
34. Stuckey, P.J.: Lazy clause generation: combining the power of sat and CP (and MIP?) solving. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 5–9. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13520-0_3
35. 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). https://doi.org/10.1137/15M1030054
36. Zampelli, S., Deville, Y., Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints **15**(3), 327–353 (2010)