A Partitioning Algorithm for Maximum Common Subgraph Problems*

Ciaran McCreesh, Patrick Prosser, James Trimble
University of Glasgow, Glasgow, Scotland
j.trimble.1@research.gla.ac.uk

Abstract
We introduce a new branch and bound algorithm for the maximum common subgraph and maximum common connected subgraph problems which is based around vertex labelling and partitioning. Our method in some ways resembles a traditional constraint programming approach, but uses a novel compact domain store and supporting inference algorithms which dramatically reduce the memory and computation requirements during search, and allow better dual viewpoint ordering heuristics to be calculated cheaply. Experiments show a speedup of more than an order of magnitude over the state of the art, and demonstrate that we can operate on much larger graphs without running out of memory.

1 Introduction
To determine the similarity or difference between two graphs, we must first find what they have in common [Bunke, 1997; Fernández and Valiente, 2001; Kriege, 2015]. The maximum common subgraph family of problems involves finding a large graph which is isomorphic to subgraphs of two given graphs simultaneously. Because graphs are widely used to model real-world phenomena, maximum common subgraph problems have arisen in molecular science (where graphs often represent molecules) [Raymond and Willett, 2002; Ehrlich and Rarey, 2011; Gay et al., 2014; Grindley et al., 1993], and also in other domains including malware detection [Park et al., 2013], source code analysis [Djoko et al., 1997], and computer vision [Cook and Holder, 1994].

Maximum common subgraph problems are NP-hard, and remain challenging computationally. Recent practical progress has been made by using constraint programming [Vismara and Valery, 2008; Ndiaye and Solnon, 2011; McCreesh et al., 2016] and mathematical programming [Bahiene et al., 2012], by reducing to the maximum clique problem [Levi, 1973; McCreesh et al., 2016], and by adapting subgraph isomorphism algorithms [Hoffmann et al., 2017].

Some special cases also have practical polynomial time algorithms [Droschinsky et al., 2016; 2017]. This paper considers the maximum common induced subgraph problem, in which the objective is to find a graph with as many vertices as possible which is an induced subgraph of each of two input graphs. (The maximum common partial subgraph problem instead asks for a common non-induced subgraph with as many edges as possible [Ndiaye and Solnon, 2011]; we discuss only the induced variant in this paper.) We introduce a new branch and bound algorithm which exploits special properties of the problem to allow a much faster exploration of the search space, whilst retaining the filtering and bounding benefits of the constraint programming approach. We describe the algorithm for the basic maximum common subgraph problem, and discuss how it may be adapted to handle vertex labels, edge labels, and the requirement that the found subgraph be connected. We then present an empirical study of the algorithm, demonstrating that it improves the state of the art by more than an order of magnitude on the unlabelled variant of the problem, and showing that it can handle much larger instances than earlier constraint programming or clique approaches due to lower memory usage.

2 The MC-SPLIT Algorithm
We initially assume that graphs are unlabelled, undirected and without loops (Section 2.2 describes how these restrictions may be relaxed). The vertex and edge sets of a graph $\mathcal{G}$ are denoted $V(\mathcal{G})$ and $E(\mathcal{G})$. The set of vertices adjacent to vertex $v$ in graph $\mathcal{G}$ is called the neighbourhood of $v$, denoted $N(\mathcal{G}, v)$. We denote by $\overline{N}(\mathcal{G}, v)$ the inverse neighbourhood of $v$, being the set of the vertices not adjacent to $v$ (excluding $v$ itself). A subgraph of a graph $\mathcal{G}$ is a graph consisting of some of the vertices of $\mathcal{G}$, and all of the edges between these vertices. (All subgraphs in this paper are induced subgraphs.) A common subgraph of two graphs is a graph which is isomorphic to a subgraph of two graphs simultaneously, and a maximum common subgraph is one with as many vertices as possible.

Throughout, $\mathcal{G}$ and $\mathcal{H}$ will be the two input graphs to our maximum common subgraph problem. The orders (number of vertices) of these graphs are denoted $g$ and $h$ respectively.

With these definitions established, we now present MC-SPLIT. This algorithm finds a maximum-cardinality mapping $M^* = \{(v_1, w_1), \ldots, (v_m, w_m)\}$ with $|M^*| = m$ vertex pairs,

---

*This work was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/K503058/1 and EP/M508056/1]
where the \( v_i \) are distinct vertices from \( V(G) \) and the \( w_i \) are distinct vertices from \( V(H) \), such that \( v_i \) and \( v_j \) are adjacent in \( G \) if and only if \( w_i \) and \( w_j \) are adjacent in \( H \). Given such a mapping, the subgraph of \( G \) induced by \( \{ v_1, \ldots, v_m \} \) and the subgraph of \( H \) induced by \( \{ w_1, \ldots, w_m \} \) are isomorphic and correspond to a maximum common subgraph.

**Walkthrough** Before discussing the algorithm in detail, we illustrate the main concepts using the graphs \( G \) and \( H \) in Figure 1. These graphs have a maximum common subgraph with four vertices; one example is the mapping \( \{1a, 2f, 3d, 5b\} \) where vertex 1 is assigned to vertex \( a \), 2 is assigned to \( f \), 3 to \( d \), and 5 to \( b \).

The algorithm builds up a mapping \( M \) using a depth-first search, starting with the empty mapping \( \emptyset \) and adding a \((v_i, w_i)\) pair or choosing to leave a vertex in \( V(G) \) unmatched at each level of the search tree. Select a vertex in \( V(G) \) as the first vertex to be mapped; in our example we will arbitrarily choose vertex 1. Each of the vertices in \( V(H) \) to which vertex 1 may be mapped will be tried in turn, and finally the possibility where vertex 1 remains unmatched will be tried.

We begin by mapping vertex 1 to vertex \( a \), giving \( M = \{1a\} \). Now label each unmatched vertex in \( V(G) \) according to whether it is adjacent to vertex 1, and label each unmatched vertex in \( V(H) \) according to whether it is adjacent to vertex 1, as shown in Figure 2(a). Adjacent vertices have label 1; non-adjacent vertices have label 0. We can extend \( M \) with a mapping \( vw \), with \( v \in V(G) \) and \( w \in V(H) \), if and only if \( v \) and \( w \) have the same label. This property, that two vertices may be mapped together if and only if they share a label, is the algorithm’s main invariant.

Next, extend the mapping by pairing a vertex in \( G \) with a vertex in \( H \) of the same label; we will choose to map vertex 2 to vertex \( d \), giving \( M = \{1a, 2d\} \) (Figure 2(b)). Each unmatched vertex \( v \in V(G) \) is labelled with a two-character bit string, indicating its adjacency to each of the two mapped vertices in \( V(G) \) (vertices 1 and 2). For example, vertex 3 is labelled 01, indicating that it is not adjacent to vertex 1 but is adjacent to vertex 2. Labels are given to unmapped vertices in \( V(H) \) in a similar fashion, showing adjacency to matched vertices \( a \) and \( d \). Our invariant is maintained: we can extend \( M \) by a vertex pairing if and only if the two vertices have the same label.

The algorithm backtracks when the incumbent (the largest mapping found so far) is at least as large as a calculated bound given \( M \) and the current labelling. To demonstrate how this bound is calculated, we consider the situation one level deeper in the search tree shown in Figure 2(c).

Three vertex labels are used: 100, 101, and 111. The first two of these only appear in one graph, and therefore there is no way to add a pair of vertices with label 100 or 101 to the mapping. The final label, 111, appears once in \( G \) and twice in \( H \), and therefore at most one pair with this label can be added to \( M \). Thus, the upper bound on mapping size is \(|M| + 1 = 4\). The general formula for the upper bound is

\[
\text{bound} = |M| + \sum_{l \in L} \min\big(|\{v \in V(G) : \text{label}(v) = l\}|, |\{v \in V(H) : \text{label}(v) = l\}|ig),
\]

where \( L \) is the set of labels used in both graphs.

**Label classes** We require only \( O(g + h) \) space per level of the search tree to store labelling information. This is done by storing a label class as a pair \((G, H)\) for each label \( l \) that is used, where \( G \) is the set of vertices in \( V(G) \) labelled \( l \), and \( H \) is the set of vertices in \( V(H) \) labelled \( l \). Since there are \( g + h \) vertices in the two graphs, at most \( g + h \) label classes can exist at once, and there are at most \( g + h \) vertices in the union of all of the \( G \) and \( H \) sets. Furthermore, we do not actually need to store the bits making up a label—we care only that like-labelled vertices are kept together, and the label itself is not used. Nor do we need to store any label class which is
Algorithm 1: Finding a maximum common subgraph.

1. Search(future, M)
2. begin
3. if |M| > |incumbent| then incumbent ← M
4. bound ← |M| + \(\sum_{G,H \in \text{future}} \min(|G|, |H|)\)
5. if bound ≤ |incumbent| then return
6. (G, H) ← SelectLabelClass(future)
7. v ← SelectVertex(G)
8. for w ∈ H do
9. future′ ← \(\emptyset\)
10. for \((G', H') \in \text{future}\) do
11. \(G'' \leftarrow G' \cap N(G, v) \setminus \{v\}\)
12. \(H'' \leftarrow H' \cap N(H, w) \setminus \{w\}\)
13. if \(G'' \neq \emptyset \) and \(H'' \neq \emptyset\) then
14. \(\text{Search}(\text{future}', M \cup \{(v, w)\})\)
15. \(G' \leftarrow G \setminus \{v\}\)
16. future ← future \(\setminus \{(G', H)\}\)
17. if \(G' \neq \emptyset\) then future ← future \(\cup \{(G', H)\}\)
18. Search(future, M)
19. McSplit(G, H)
20. begin
21. global incumbent ← \(\emptyset\)
22. Search(\(\{(V(G), V(H))\}\), incumbent)
23. return incumbent

present only in one graph but not the other (or which is not present at all). Together, these facts allow us to store all the necessary information in three arrays. The first stores a permutation of \(V(G)\) in which like-labelled vertices appear consecutively. The second array, similarly, stores a permutation of \(V(H)\) with like-labelled vertices together. The third array contains a record for each label class \((G, H)\). Each record contains start and end pointers to the portion of the first array that contains \(G\), and start and end pointers to the portion of the second array that contains \(H\). This representation has similarities to data structures used in partition backtracking for graph isomorphism [López-Presa and Anta, 2009; McKay and Piperno, 2014] and the Bron and Kerbosch [1973] clique enumeration algorithm.

Algorithm 1 in detail The recursive procedure, Search, has two parameters. The parameter future is a list of label classes, each represented as a \((G, H)\) pair as described above. The parameter \(M\) is the current mapping of vertices. On each call to Search, the invariant holds that a \((v, w)\) pair may be added to \(M\) if and only if \(v\) and \(w\) belong to the same label class in future.

Line 3 stores the current mapping \(M\) if it is large enough to unseat the incumbent. Lines 4 and 5 prune the search when a calculated upper bound is not larger than the incumbent.

The remainder of the procedure performs the search. A label class \((G, H)\) is selected from future using some heuristic (line 6); from this label class, a vertex \(v\) is selected from \(G\) (line 7). We now iterate over all vertices \(w\) in \(H\), exploring the consequences of adding \((v, w)\) to \(M\) (lines 8 to 19). A new set of label-classes, future′, is created (line 9); this will be the labelling that results from adding \((v, w)\) to our mapping. Every label-class in future can now be split (lines 10 to 18) into two new classes. The first of these classes (lines 11 to 14) contains vertices in \(G\) adjacent to \(v\) and vertices in \(H\) adjacent to \(w\). This is added to future′ if both sets contain at least one vertex. This is then repeated symmetrically for non-adjacency (lines 15 to 18). A recursive call is made (line 19), on return from which we remove the mapping \((v, w)\). Having explored all possible mappings of \(v\) with vertices in \(H\) we now consider what happens if \(v\) is not matched (lines 20 to 23).

We start our search at the function McSplit (line 24), with graphs \(G\) and \(H\) as inputs. This function returns a mapping of maximum cardinality. In line 27 the initial call is made to Search; at this point we have a single label-class containing all vertices, and the mapping \(M\) is empty.

2.1 Heuristics

Small scale experiments (not presented here) were performed to identify suitable heuristics for the SelectLabelClass and SelectVertex functions. Our SelectLabelClass function chooses a label class with the smallest \(\max(|G|, |H|)\), breaking ties by selecting a class containing a vertex in \(G\) with the largest degree. From the selected class, SelectVertex chooses a vertex in \(G\) with maximum degree. We further discuss the effectiveness of our SelectLabelClass function in Section 4.

2.2 Extensions

Maximum common subgraph problems come in many variants. Often vertices or edges have labels (for example, denoting the kind of atom or bond they represent in a molecule [Ehrlich and Rarey, 2011]), and the induced subgraphs of the two input graphs are required to have identical labels. Directed edges are used in an application to systems of biochemical reactions [Gay et al., 2014]. We now outline how to adapt Algorithm 1 to handle these cases.

Vertex labels and loops If vertices in the input graphs have labels (as distinct from the bit-string labels described in Section 2), replace \(\{(V(G), V(H))\}\) in line 27 with a set of label classes, one for each label that appears on at least one vertex of both \(G\) and \(H\). If some vertices have loops, we create two label classes for each input label: one class containing vertices with loops, and the other containing vertices without loops.

Directed graphs without edge labels Before running the algorithm, we create two-dimensional arrays \(A_G\) and \(A_H\) representing adjacencies in \(G\) and \(H\) respectively. These store
described by Vismara and Valery [2008]. We can represent non-zero elements in its bit-string label, following the scheme permitting branching only on a vertex complexity at each search node, by storing an extra bit with each subgraph be connected [Ehrlich and Rarey, 2011]. We consider the complications, it is sometimes desirable to require the common edge. This case is similar to its undirected case. That resembles the merging step of merge sort. We now perform a four-way split where each vertex is classified according to the label on its array entry indexed by \( v \) and \( w \). This is shown in Algorithm 2, where \( L = \{0, 1, 2, 3\} \).

**Algorithm 2**: Replacement for lines 11 to 18 of Algorithm 1 to handle directed and edge-labelled cases.

```
for \( l \in L \) do
    \( G'' \leftarrow \{u \in G' : u \neq v \land A_G[v][u] = l\} \)
    \( H'' \leftarrow \{u \in H' : u \neq w \land A_H[w][u] = l\} \)
    if \( G'' \neq \varnothing \) and \( H'' \neq \varnothing \) then
        \( \text{future}' \leftarrow \text{future}' \cup \{(G'', H'')\} \)
```

the same information as the graphs’ adjacency matrices, but allow us to determine in a single memory access which of the two possible edges exist between a pair of vertices. We now describe the entries of \( A_G \). For each vertex pair \((t, u)\) in \( G \), \( A_G[t][u] \) takes the value 0 if \( t \) and \( u \) are not adjacent, 1 if the two vertices share a single edge in the direction \( t \rightarrow u \), 2 if they share a single edge in the direction \( u \rightarrow t \), and 3 if there are edges in both directions. Where lines 11 to 18 of the basic algorithm split the label class \((G', H')\) into two, we now perform a four-way split where each vertex is classified according to the label on its array entry indexed by \( v \) and \( w \).

### Undirected with edge labels
Each entry of \( A_G \) or \( A_H \) contains an edge label, or a null entry 0 indicating that no edge is present. We use Algorithm 2, by letting \( L \) be the union of \( \{0\} \) with the set of all labels that appear in the input graphs. Since there may be up to \( g + h \) distinct labels, the loop in Algorithm 2 may execute up to \( g + h \) times, resulting in \( O((g + h)^2) \) time complexity per search node. To achieve \( O((g + h) \log(g + h)) \) time complexity per search node, we can modify the algorithms to use sorting rather than explicitly looping over all label classes, as follows. First, run lines 17-19 of Algorithm 1 to create a new label-class of vertices that are not adjacent to \( v \) or \( w \), and remove these vertices from \((G', H')\). Next, sort \( G' \) and \( H' \) in ascending order of the label on the edge from \( v \) or \( w \) to each vertex. We can then create the label classes corresponding to each edge label by simultaneously traversing \( G' \) and \( H' \) from left to right, in a manner that resembles the merging step of merge sort.

### Directed with edge labels
This case is similar to its undirected counterpart, except that each element \( A_G[u][v] \) or \( A_H[u][v] \) is a pair \((l_1, l_2)\), where \( l_1 \) is the label on the edge \( u \rightarrow v \) (or 0 if no edge exists) and \( l_2 \) is the label on the reverse edge.

### Maximum common connected subgraph
In chemistry applications, it is sometimes desirable to require the common subgraph be connected [Ehrlich and Rarey, 2011]. We consider only undirected graphs. We may modify MC\textsc{Split} by permitting branching only on a vertex \( v \) that has at least one non-zero element in its bit-string label, following the scheme described by Vismara and Valery [2008]. We can represent this information compactly, and without increasing time complexity at each search node, by storing an extra bit with each label class. This bit takes the value 1 if and only if the vertices in the class are adjacent to at least one vertex in \( M \).

### 3 Experimental Evaluation
Experiments were performed on machines with dual Intel Xeon E5-2640 v2 CPUs and 64GBytes RAM. Our algorithm was implemented in C++ and compiled using g++ 5.3.0. We compare against the best constraint programming implementations of Ndiaye and Solnon [2011] and McCreesh et al. [2016] (CP-FC in the unlabelled cases, and CP-MAC in the labelled cases, using both branching and filtering for connected subgraphs), the clique encodings of McCreesh et al. [2016], and the \( k \)-\textsc{split} algorithm of Hoffmann et al. [2017] (which only supports unlabelled, undirected, unconnected instances). Each of these comparator programs is an optimised, dedicated implementation and does not use a general-purpose constraint programming toolkit. We used the original authors’ code in each case.

Our first set of experiments uses a database of randomly-generated maximum common subgraph instances [Santo et al., 2003; Conte et al., 2007]. For unlabelled instances, we selected the first ten instances from each family whose members have no more than 50 vertices, for a total of 4,100 instances. For labelled instances, we selected the first ten instances from every family, for a total of 8,140 instances with up to 100 vertices; like McCreesh et al. [2016], we use the labelling scheme in which the number of distinct vertex labels and the number of distinct edge labels is approximately equal to 33 percent of the number of vertices in each graph.

### Unlabelled, undirected
Figure 3(a) shows a plot of cumulative number of instances solved against runtime. We may compare the speed of two algorithms using the horizontal distance between their curves. For example, we could solve 2,000 of the 4,110 unlabelled undirected instances using the MC\textsc{Split} algorithm if a time limit of 0.5 seconds per instance were imposed. Its nearest competitor, CP-FC, would require a time limit of over 24 seconds per instance to solve the same number of instances. For any given number of instances, MC\textsc{Split} is comfortably more than an order of magnitude faster than its nearest competitor. Moreover, MC\textsc{Split} is the fastest algorithm on 87% of the 3,506 instances that could be solved by at least one of the four algorithms in less than 1,000 seconds.

### Vertex and edge labels, directed
Cumulative runtimes for this class of instances are in Figure 3(b). Again, MC\textsc{Split} is over an order of magnitude faster than the best existing CP algorithm, which is CP-MAC in this case. Matching the conclusions of McCreesh et al. [2016], we see that the clique encoding outperforms the other algorithms—including MC\textsc{Split}—on these labelled instances, except in the very easy region of instances that can be solved in well under 100 ms.

\(^1\)Source code, instances, experimental scripts and raw results are available at https://github.com/jamestrimble/sjcai2017-partitioning-common-subgraph
Unlabelled, undirected, connected This class of instances is shown in Figure 3(c). These results are very similar to the corresponding experiment in Figure 3(a) in which the subgraph is not required to be connected: McSplit is the clear winner by more than an order of magnitude.

Vertex and edge labels, undirected, connected For the labelled, connected case, clique slightly outperforms McSplit on harder instances (Figure 3(d)). However, the gap between the two algorithms is very narrow, and is probably down to minor implementation details; indeed, the cumulative curve for McSplit briefly rises above the curve for clique at a runtime just below 100 seconds. Additionally, McSplit is the clear winner for easier instances, where the clique encoding is relatively expensive to construct but trivial to solve.

Large subgraph isomorphism instances We also ran the algorithms on a set of 5,725 larger instances used in recent studies of subgraph isomorphism [Kotthoff et al., 2016] and maximum common subgraph [Hoffmann et al., 2017]. This benchmark set includes real-world graphs and graphs generated using random models. Pattern graphs range from 4 vertices to 900 with a median of 80; target graphs range from 10 vertices to 6,671 with a median of 561. Cumulative runtimes on these instances are shown in Figure 4. This is a challenging set of instances, and more than half of the instances cannot be solved within a timeout of 1,000 seconds by any solver. Furthermore, the CP-FC algorithm and the clique encoding run out of memory on many of the instances (these are treated as timeouts, following Hoffmann et al. [2017]).

The basic McSplit is beaten by the $k\downarrow$ algorithm of Hoffmann et al. [2017] on this dataset. However, we can modify the McSplit algorithm to use a top-down strategy similar to that used by $k\downarrow$ by calling the main McSplit method once per goal size ($g, g-1, g-2, \ldots$); we backtrack (line 5 of Algorithm 1) when the bound is strictly less than the goal size, and terminate when a solution of the goal size is found. We expect that this could do well because in many cases the maximum common subgraph covers nearly all of the smaller graph—

![Figure 3: Cumulative numbers of instances solved over time for maximum common subgraph problems.](image)

![Figure 4: Cumulative numbers of instances solved over time for the maximum common connected subgraph problem on the large subgraph isomorphism benchmark suite.](image)
Indeed, Figure 4 shows that this approach is the strongest on these instances, and MCSP↓ is the best algorithm for every choice of timeout. (By contrast, the optimal solutions for the instances in Figure 3 typically cover a smaller proportion of the input graphs, and MCSP↓ is often more than a magnitude slower than the plain MCSP algorithm for these instances; the MCSP↓ results are not shown for these instances.)

Overall, we find that MCSP↑ improves on the previous state of the art by more than an order of magnitude for small, unlabelled graphs. On a benchmark suite of larger instances, the MCSP↓ variant of the algorithm comfortably outperforms the state of the art. On labelled instances, the clique encoding remains the strongest solver, except on the most trivial instances—particularly where there is no requirement for the common subgraph to be connected.

4 Comparison with Existing Algorithms

Our experimental results suggest that MCSP↑ has broadly similar performance trends to the constraint programming, forward-checking (CP-FC) algorithm of Ndiaye and Solnon [2011], but with much lower constant factors and memory usage. Indeed, in Figures 5(a) and 5(b) we plot the number of recursive calls made by our algorithm versus the number made by CP-FC, for the unlabelled and labelled, unconnected problem instances. (Rather than a simple scatter plot, we use darker colours to indicate a higher density of points around a location.) We see a close correlation: MCSP↑ typically does slightly less work, and sometimes does more, but instances with more than one order of magnitude difference in search tree size are rare.

Why is this? We do not see a similar correlation between our search tree size and that of the clique approach. The key observation is that MCSP↑ may be considered to be a different version of the CP-FC algorithm, using an unconventional domain store and more efficient filtering algorithms. We now explore this relationship further.

In the CP-FC algorithm, each vertex \( v \in V(G) \) is represented by a variable, whose domain corresponds to the set of vertices in \( V(H) \) to which \( v \) may currently be mapped, with an additional special \( \perp \) value representing an unmapped vertex. Given a label class \((G, H)\) in MCSP↑ the vertices in \( G \) correspond to variables in CP-FC, and the vertices in \( H \) to domain values. The label-class representation of domains is possible because throughout the CP-FC algorithm for maximum common subgraph, the domains of any two variables are either identical or disjoint (excluding \( \perp \), which is either present in all domains or in none). To the best of our knowledge, this observation has not been made previously, and it is not exploited in other algorithms for maximum common subgraph.

CP-FC uses a soft all-different constraint to compute a bound, which requires running a matching algorithm on a supporting compatibility graph. We now show that MCSP↑ computes the same bound, but using a simple counting loop (Algorithm 1, line 4)—this is possible because of the disjoint nature of the domains.

To illustrate the method used by CP-FC to calculate a bound, we consider the input graphs \( G \) and \( H \) in Figure 1. Suppose that the variable corresponding to vertex 1 has been matched to the value corresponding to vertex \( a \), and that no other assignments have been made. In the terminology of MCSP↑, we now have two label classes: \( \{\{2, 3\}, \{d, f\}\} \) and \( \{\{4, 5\}, \{b, c, e\}\} \). CP-FC represents this state by storing the domain of each remaining variable: 2 and 3 each have the domain \( \{d, f\} \); and 4 and 5 each have the domain \( \{b, c, e\} \).

As described previously, the MCSP↑ algorithm finds the size of the smaller set in each label class, and adds these sizes to the size of the current mapping, giving a bound of 5. CP-FC uses the compatibility graph \( B \) in Figure 6 to calculate an upper bound: variables are represented by vertices on the left, values are represented by vertices on the right, and a variable may take a value if and only if the corresponding vertices are adjacent. The upper bound is calculated by CP-FC by computing a maximum matching on this bipartite graph, and adding the size of this matching to the size of the current mapping.

The vertices in the bipartite graph are coloured to emphasise how they correspond to the two label classes. Each label class corresponds to a complete bipartite graph, and there are no edges between the components corresponding to different label classes. Clearly, a maximum matching in \( B \) must be
the union of the maximum matching in each complete bipartite subgraph, and each of these matchings in turn have the same size as the smaller of the left and right sets of vertices. It follows that the CP-FC bound is identical to the McSplit bound.

This bound is incomparable to the clique bound, which is given by a greedy colouring of a microstructure-like graph [McCreesh et al., 2016]. The clique bound is potentially stronger, in that it can reason about compatibilities between individual variable-vertex assignments. The clique bound is also able to make use of edge label information, which CP-FC and our equivalent bound cannot—this comes at the expense of much higher memory requirements, and also longer calculation times for unlabelled graphs. The clique bound is also potentially weaker: it is not even guaranteed to be less than or equal to the number of remaining variables, in the CP-FC sense.

A further advantage of our encoding is that it gives us efficient access to a better branching heuristic. The CP-FC algorithm uses smallest domain first, which in our algorithm corresponds to branching on a label class with smallest \( |H| \). We instead branch on the label class with smallest \( \max(|G|, |H|) \). This is empirically better, and accounts for much of the difference between the number of recursive calls made; branching on the smallest \( |G|/|H| \) gives very similar results. This can be viewed as exploiting both smallest domain first, and the dual viewpoint [Geelen, 1992] of smallest domain first, simultaneously, but we do not have the overheads of having to maintain and channel between the dual viewpoint that would be required when using a conventional domain store.

What about our relationship to the \( k \downarrow \) of Hoffmann et al. [2017]? Figure 5(c) plots the number of recursive calls made by \( k \downarrow \) and McSplit\( \downarrow \) on each of the subgraph isomorphism instances. Although McSplit\( \downarrow \) is the faster of the two algorithms overall, it explores more search nodes than \( k \downarrow \) for most instances (even taking into account that \( k \downarrow \) uses a unit propagation loop, and so measures the search tree slightly differently). This is the classic tradeoff between speed and cleverness. A hybrid algorithm could be beneficial here: it could use \( k \downarrow \) initially, switching to McSplit\( \downarrow \) when the extra filtering is ineffective, and finally switching to a clique encoding when fewer than some threshold number of vertices remain to be selected. This might deliver the benefits of the clique encoding for labelled graphs, while avoiding the high memory cost and colouring time of encoding the full instance.

5 Conclusion

We have introduced the McSplit algorithm for maximum common subgraph problems. This algorithm is more than an order of magnitude faster than the previous state of the art for unlabelled and undirected instances. We have shown how the algorithm can be extended for graphs with labels on edges, labels on vertices, loops, directed edges and the requirement that the resultant graph be connected.

We believe there is more to be discovered about branching heuristics. There is also the potential to branch on both sides, that is instead of branching on vertices in \( V(G) \), it would be equally valid to branch on a vertex in \( V(H) \), since our data structure treats the two graphs symmetrically. More interestingly, we could choose which graph to branch on at each search node using some heuristic (perhaps choosing based on whether the \( G \) or \( H \) set is smaller).

It would be interesting to see whether these techniques are more broadly applicable—we suspect that some other problems may have a similar branching structure which would also benefit from a partitioning domain store representation. Most obviously, we could solve the induced subgraph isomorphism problem in the same way (and with nearly no changes to the code), and our connected variant shows that certain side constraints can also be handled. However, we cannot solve non-induced subgraph isomorphism this way, nor can we handle certain richer labelling schemes such as those used in temporal subgraph isomorphism [Redmond and Cunningham, 2013].

References


[Droschinsky et al., 2016] Andre Droschinsky, Nils M. Kriege, and Petra Mutzel. Faster algorithms for the maximum common subtree isomorphism problem. In 41st


