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 [**?**; **?**; **?**]. 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) [**?**; **?**; **?**; **?**], and also in other domains including malware detection [**?**], source code analysis [**?**], and computer vision [**?**].

Maximum common subgraph problems are NP-hard, and remain challenging computationally. Recent practical progress has been made by using constraint programming [**?**; **?**; **?**] and mathematical programming [**?**], by reducing to the maximum clique problem [**?**; **?**], and by adapting subgraph isomorphism algorithms [**?**]. Some special cases also have practical polynomial time algorithms [**?**; **?**].

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 [**?**]; 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.

*This work was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/K503058/1 and EP/M508056/1]

2 The McSplit 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 $\mathrm{V}(\mathcal{G})$ and $\mathrm{E}(\mathcal{G})$. The set of vertices adjacent to vertex v in graph $\mathcal{G}$ is called the *neighbourhood* of v, denoted $\mathrm{N}(\mathcal{G}, v)$. We denote by $\overline{\mathrm{N}}(\mathcal{G}, v)$ 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 McSplit. This algorithm finds a maximum-cardinality mapping $M^* = \{(v_1, w_1), \ldots, (v_m, w_m)\}$ with $|M^*| = m$ vertex pairs, where the v_i are distinct vertices from $\mathrm{V}(\mathcal{G})$ and the w_i are distinct vertices from $\mathrm{V}(\mathcal{H})$, such that v_i and v_j are adjacent in $\mathcal{G}$ if and only if w_i and w_j are adjacent in $\mathcal{H}$. Given such a mapping, the subgraph of $\mathcal{G}$ induced by $\{v_1, \ldots, v_m\}$ and the subgraph of $\mathcal{H}$ induced by $\{w_1, \ldots, w_m\}$ are isomorphic and correspond to a maximum common subgraph.

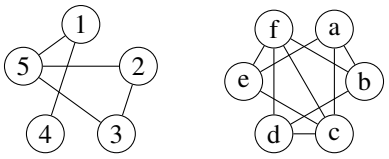

Figure 1: Example graphs $\mathcal{G}$ and $\mathcal{H}$.

Walkthrough Before discussing the algorithm in detail, we illustrate the main concepts using the graphs $\mathcal{G}$ and $\mathcal{H}$ 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 $\varnothing$ and adding a (v_i, w_i) pair or choosing to leave a vertex in $\mathrm{V}(\mathcal{G})$ unmatched at each level of the search tree. Select a vertex in $\mathrm{V}(\mathcal{G})$ as the first vertex to be mapped; in our example we will arbitrarily choose vertex 1. Each of the vertices in $\mathrm{V}(\mathcal{H})$ 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 $\mathrm{V}(\mathcal{G})$ according to whether it is adjacent to vertex 1, and label each unmatched vertex in $\mathrm{V}(\mathcal{H})$ according to whether it is adjacent to vertex a, 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 $v \in \mathrm{V}(\mathcal{G})$ and $w \in \mathrm{V}(\mathcal{H})$, 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 $\mathcal{G}$ with a vertex in $\mathcal{H}$ of the same label; we will choose to map vertex 2 to vertex d, giving $M = \{1a, 2d\}$ (Figure 2(b)). Each unmapped vertex $v \in \mathrm{V}(\mathcal{G})$ is labelled with a two-character bit string, indicating its adjacency to each of the two mapped vertices in $\mathrm{V}(\mathcal{G})$ (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 $\mathrm{V}(\mathcal{H})$ 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 $\mathcal{G}$ and twice in $\mathcal{H}$, and therefore at most one pair with this label can be added to M. Thus, the upper bound on mapping size is $|M| + 1 = 4$.

(a) After mapping 1 to a

Mapping $\{1a\}$

Labelling of $\mathcal{G}$		Labelling of $\mathcal{H}$	
Vertex	Label	Vertex	Label
2	0	b	1
3	0	c	1
4	1	d	0
5	1	e	1
		f	0

(b) After mapping 2 to d

Mapping $\{1a, 2d\}$

Labelling of $\mathcal{G}$		Labelling of $\mathcal{H}$	
Vertex	Label	Vertex	Label
3	01	b	11
4	10	c	11
5	11	e	10
		f	01

(c) After mapping 3 to f

Mapping $\{1a, 2d, 3f\}$

Labelling of $\mathcal{G}$		Labelling of $\mathcal{H}$	
Vertex	Label	Vertex	Label
4	100	b	111
5	111	c	111
		e	101

Figure 2: Mapping M and vertex labels during search on example graphs $\mathcal{G}$ and $\mathcal{H}$ from Figure 1. Labels represent adjacencies; for example, the label 101 on vertex e in the final table signifies that e is adjacent to the first and third mapped vertices of $\mathcal{H}$ (a and f) but not adjacent to the second mapped vertex (d).

The general formula for the upper bound is

$$bound = |M| + \sum_{l \in L} \min\big(|\{v \in \mathrm{V}(\mathcal{G}) : \mathrm{label}(v) = l\}|, |\{v \in \mathrm{V}(\mathcal{H}) : \mathrm{label}(v) = l\}|\big),$$

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

Label classes We require only $\mathrm{O}(g + h)$ space per level of the search tree to store labelling information. This is done by storing a *label class* as a pair $\langle G, H \rangle$ for each label l that is used, where G is the set of vertices in $\mathrm{V}(\mathcal{G})$ labelled l, and H is the set of vertices in $\mathrm{V}(\mathcal{H})$ 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 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 $\mathrm{V}(\mathcal{G})$ in which like-

Algorithm 1: Finding a maximum common subgraph.

```
1  Search(future, M)
2  begin
3      if |M| > |incumbent| then incumbent ← M
4      bound ← |M| + Σ_{⟨G,H⟩∈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′ ← ∅
10         for ⟨G′, H′⟩ ∈ future do
11             G″ ← G′ ∩ N(G, v) ∖ {v}
12             H″ ← H′ ∩ N(H, w) ∖ {w}
13             if G″ ≠ ∅ and H″ ≠ ∅ then
14                 future′ ← future′ ∪ {⟨G″, H″⟩}
15             G″ ← G′ ∩ N̄(G, v) ∖ {v}
16             H″ ← H′ ∩ N̄(H, w) ∖ {w}
17             if G″ ≠ ∅ and H″ ≠ ∅ then
18                 future′ ← future′ ∪ {⟨G″, H″⟩}
19         Search(future′, M ∪ {(v, w)})
20     G′ ← G ∖ {v}
21     future ← future ∖ {⟨G, H⟩}
22     if G′ ≠ ∅ then future ← future ∪ {⟨G′, H⟩}
23     Search(future, M)

24 McSplit(G, H)
25 begin
26     global incumbent ← ∅
27     Search({⟨V(G), V(H)⟩}, ∅)
28     return incumbent
```

labelled vertices appear consecutively. The second array, similarly, stores a permutation of $\mathrm{V}(\mathcal{H})$ with like-labelled vertices together. The third array contains a record for each label class $\langle G, H\rangle$. 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 [**?**; **?**] and the **?** [**?**] 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 $\langle G, H\rangle$ 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 $\langle G, H\rangle$ 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 $\mathcal{G}$ and $\mathcal{H}$ 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 [**?**]), 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 [**?**]. 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 $\langle V(\mathcal{G}), V(\mathcal{H})\rangle$ in line 27 with a set of label classes, one for each label that appears on at least one vertex of both $\mathcal{G}$ and $\mathcal{H}$. 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 $A_{\mathcal{G}}$ and $A_{\mathcal{H}}$ representing adjacencies in $\mathcal{G}$ and $\mathcal{H}$ respectively. These store 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 $A_{\mathcal{G}}$. For each vertex pair (t, u) in $\mathcal{G}$,

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

```
1 for l ∈ L do
2     G'' ← {u ∈ G' : u ≠ v ∧ A_G[v][u] = l}
3     H'' ← {u ∈ H' : u ≠ w ∧ A_H[w][u] = l}
4     if G'' ≠ ∅ and H'' ≠ ∅ then
5         future' ← future' ∪ {⟨G'', H''⟩}
```

$A_{\mathcal{G}}[t][u]$ takes the value 0 if t and u are not adjacent, 1 if the two vertices share a single edge in the direction $t \to u$, 2 if they share a single edge in the direction $u \to t$, and 3 if there are edges in both directions. Where lines 11 to 18 of the basic algorithm split the label class $\langle G', H' \rangle$ in 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. This is shown in Algorithm 2, where $L = \{0, 1, 2, 3\}$.

Undirected with edge labels Each entry of $A_{\mathcal{G}}$ or $A_{\mathcal{H}}$ 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 $\mathrm{O}\left((g+h)^2\right)$ time complexity per search node. To achieve $\mathrm{O}\left((g+h)\log(g+h)\right)$ 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 $\langle G', H' \rangle$. 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 $A_{\mathcal{G}}[u][v]$ or $A_{\mathcal{H}}[u][v]$ is a pair (l_1, l_2), where l_1 is the label on the edge $u \to v$ (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 [**?**]. We consider only undirected graphs. We may modify McSplit 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 **?** [**?**]. 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[1] in C++ and compiled using g++ 5.3.0. We compare against the best constraint programming implementations of **?** [**?**] and **?** [**?**] (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 **?** [**?**], and the $k\downarrow$ algorithm of **?** [**?**] (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 [**?**; **?**]. 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 **?** [**?**], 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 McSplit 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, McSplit is comfortably more than an order of magnitude faster than its nearest competitor. Moreover, McSplit 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 than 1,000 seconds.

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

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.

[1] Source code, instances, experimental scripts and raw results are available at https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph

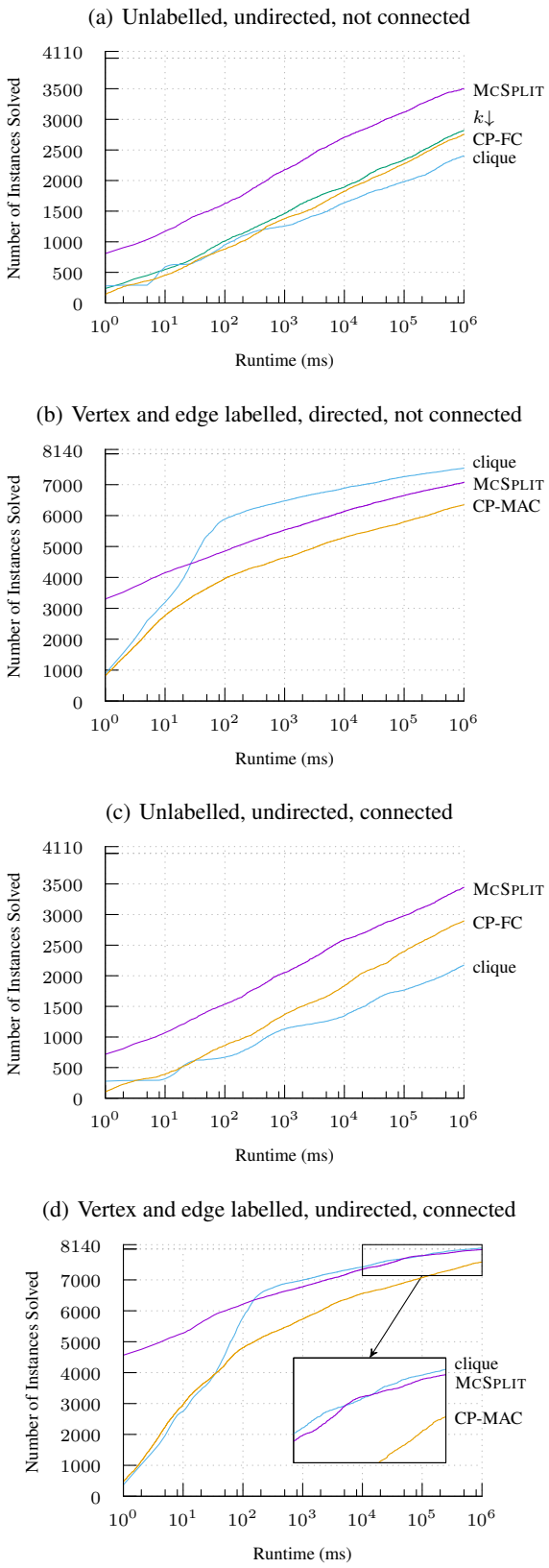

Figure 3: Cumulative numbers of instances solved over time for maximum common subgraph problems.

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 [**?**] and maximum common subgraph [**?**]. 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 **?** [**?**]).

The basic McSplit is beaten by the $k\downarrow$ algorithm of **?** [**?**] 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 $(g, g-1, g-2, \dots)$; 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—indeed, Figure 4 shows that this approach is the strongest on these instances, and McSplit↓ 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 McSplit↓ is often more than a magnitude slower than the plain McSplit algorithm for these instances; the McSplit↓ results are not shown for these instances.)

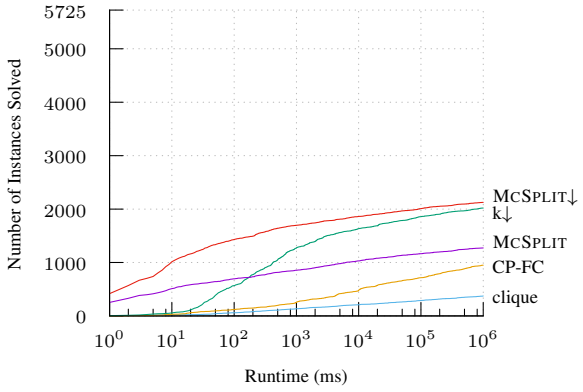

Figure 4: Cumulative numbers of instances solved over time for the maximum common connected subgraph problem on the large subgraph isomorphism benchmark suite.

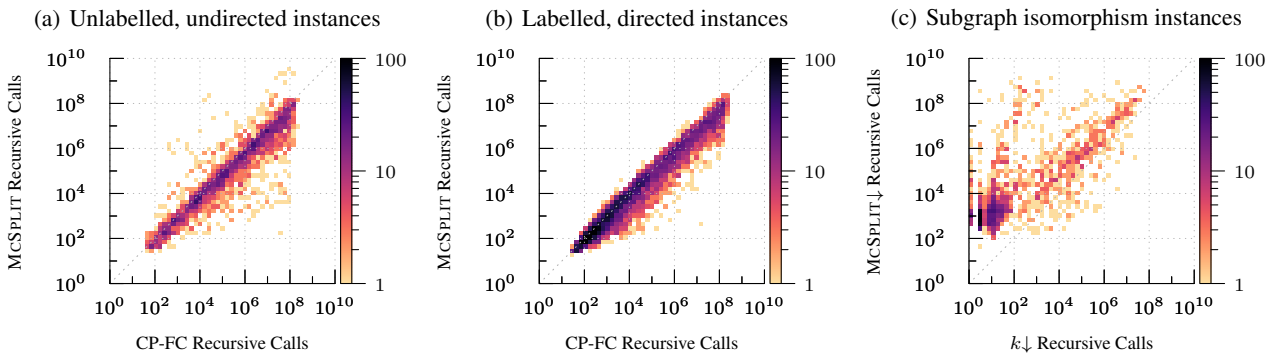

Figure 5: Relative search space sizes for instances which were solved by both algorithms within the timeout.

Overall, we find that McSplit 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 McSplit↓ 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 McSplit has broadly similar performance trends to the constraint programming, forward-checking (CP-FC) algorithm of **?** [**?**], 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: McSplit 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 McSplit 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 $v \in \mathrm{V}(\mathcal{G})$ is represented by a variable, whose domain corresponds to the set of vertices in $\mathrm{V}(\mathcal{H})$ to which v may currently be mapped, with an additional special $\perp$ value representing an unmapped vertex. Given a label class $\langle G, H \rangle$ in McSplit 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 McSplit 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 $\mathcal{G}$ and $\mathcal{H}$ 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 McSplit, we now have two label classes: $\langle \{2,3\}, \{d,f\} \rangle$ and $\langle \{4,5\}, \{b,c,e\} \rangle$. CP-FC represents this state by storing the domain of each remaining variable: 2 and 3 each have the domain $\{d,f\}$; 4 and 5 each have the domain $\{b,c,e\}$.

As described previously, the McSplit 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 $\mathcal{B}$ 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 $\mathcal{B}$ 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 [**?**]. 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

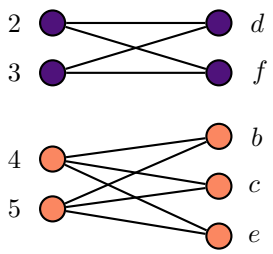

Figure 6: A bipartite compatibility graph of the type used to calculate a bound in the CP-FC algorithm.

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 [**?**] 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 **?** [**?**]? Figure 5(c) plots the number of recursive calls made by $k\downarrow$ and McSplit↓ on each of the subgraph isomorphism instances. Although McSplit↓ 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↓ 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 $\mathrm{V}(\mathcal{G})$, it would be equally valid to branch on a vertex in $\mathrm{V}(\mathcal{H})$, 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 [**?**].

References

[Bahiense *et al.*, 2012] Laura Bahiense, Gordana Manic, Breno Piva, and Cid C. de Souza. The maximum common edge subgraph problem: A polyhedral investigation. *Discrete Applied Mathematics*, 160(18):2523–2541, 2012.

[Bron and Kerbosch, 1973] Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[Bunke, 1997] Horst Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8):689–694, 1997.

[Conte *et al.*, 2007] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *J. Graph Algorithms Appl.*, 11(1):99–143, 2007.

[Cook and Holder, 1994] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res. (JAIR)*, 1:231–255, 1994.

[Djoko *et al.*, 1997] Surnjani Djoko, Diane J. Cook, and Lawrence B. Holder. An empirical study of domain knowledge and its benefits to substructure discovery. *IEEE Trans. Knowl. Data Eng.*, 9(4):575–586, 1997.

[Droschinsky *et al.*, 2016] Andre Droschinsky, Nils M. Kriege, and Petra Mutzel. Faster algorithms for the maximum common subtree isomorphism problem. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*, pages 33:1–33:14, 2016.

[Droschinsky *et al.*, 2017] Andre Droschinsky, Nils Kriege, and Petra Mutzel. Finding largest common substructures of molecules in quadratic time. In *SOFSEM 2017: Theory and Practice of Computer Science - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings*, pages 309–321, 2017.

[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.

[Fernández and Valiente, 2001] Mirtha-Lina Fernández and Gabriel Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 22(6/7):753–758, 2001.

[Gay *et al.*, 2014] Steven Gay, François Fages, Thierry Martinez, Sylvain Soliman, and Christine Solnon. On the subgraph epimorphism problem. *Discrete Applied Mathematics*, 162:214–228, 2014.

[Geelen, 1992] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *ECAI*, pages 31–35, 1992.

[Grindley *et al.*, 1993] Helen M. Grindley, Peter J. Artymiuk, David W. Rice, and Peter Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *Journal of Molecular Biology*, 229(3):707–721, 1993.

[Hoffmann *et al.*, 2017] Ruth Hoffmann, Ciaran McCreesh, and Craig Reilly. Between subgraph isomorphism and maximum common subgraph. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 3907–3914, 2017.

[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, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers*, pages 107–122, 2016.

[Kriege, 2015] Nils Kriege. *Comparing graphs*. PhD thesis, Technische Universität Dortmund, 2015.

[Levi, 1973] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO*, 9(4):341–352, 1973.

[López-Presa and Anta, 2009] José Luis López-Presa and Antonio Fernández Anta. Fast algorithm for graph isomorphism testing. In *Experimental Algorithms, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings*, pages 221–232, 2009.

[McCreesh *et al.*, 2016] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 350–368, 2016.

[McKay and Piperno, 2014] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014.

[Ndiaye and Solnon, 2011] Samba Ndojh Ndiaye and Christine Solnon. CP models for maximum common subgraph problems. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, pages 637–644, 2011.

[Park *et al.*, 2013] Young Hee Park, Douglas S. Reeves, and Mark Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*, 39:419–430, 2013.

[Raymond and Willett, 2002] John W. Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[Redmond and Cunningham, 2013] Ursula Redmond and Pádraig Cunningham. Temporal subgraph isomorphism. In *Advances in Social Networks Analysis and Mining 2013, ASONAM '13, Niagara, ON, Canada - August 25 - 29, 2013*, pages 1451–1452, 2013.

[Santo *et al.*, 2003] Massimo De Santo, Pasquale Foggia, Carlo Sansone, and Mario Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079, 2003.

[Vismara and Valery, 2008] Philippe Vismara and Benoît Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In *Modelling, Computation and Optimization in Information Systems and Management Sciences, Second International Conference, MCO 2008, Metz, France - Luxembourg, September 8-10, 2008. Proceedings*, pages 358–368, 2008.