Partitioning Algorithms for Induced Subgraph and Supergraph Problems

James Trimble

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

Month Year

Abstract

This dissertation introduces the McSplit family of algorithms for two closely-related NP-hard problems that involve finding a large induced subgraph contained by each of two input graphs: the induced subgraph isomorphism problem and the maximum common induced subgraph problem.

The McSplit algorithms resemble forward-checking constrant programming algorithms, but use problem-specific data structures that allow multiple, identical domains to be stored without duplication. These data structures enable fast, simple constraint propagation algorithms and very fast calculation of upper bounds. Versions of these algorithms for both sparse and dense graphs are described and implemented. The resulting algorithms are over an order of magnitude faster than the best existing algorithm for maximum common induced subgraph on unlabelled graphs, and outperform the state of the art on several classes of induced subgraph isomorphism instances.

A further advantage of the McSplit data structures is that variables and values are treated identically; this allows us to choose to branch on variables representing vertices of either input graph with no overhead. An extensive set of experiments shows that such two-sided branching can be particularly beneficial if the two input graphs have very different orders or densities.

Finally, we turn from subgraphs to supergraphs, tackling the problem of finding a small graph that contains every member of a given family of graphs as an induced subgraph. Exact and heuristic techniques are developed for this problem, in each case using a McSplit algorithm as a subroutine. These algorithms allow us to add new terms to two entries of the On-Line Encyclopedia of Integer Sequences.

Table of Contents

Chapter 1

Introduction

1.1 Background and Motivation

A *graph* is a collection of items, some pairs of which are related; we call the items *vertices* and the pairwise relationships *edges*. This simple abstraction is used to model systems from a huge variety of domains: a molecule is a collection of atoms joined together by bonds (Sussenguth, 1965); an image may be summarised using a vertex for each coloured region with the vertices for adjacent regions joined by edges (Olatunbosun et al., 1996). We may wish to determine whether a copy of all or a large part of a given object—such as a molecule or image—is contained in another object. This dissertation introduces new, practical algorithms for such problems.

We first consider the *maximum common induced subgraph* problem, in which we seek a large graph that appears as part (that is, is a *subgraph*) of each of two given graphs. The problem has a number of variants, such as a version where the vertices have labels—atomic number in the case of molecules—and the subgraphs must have the same labels. The McSplit family of algorithms, which are introduced in this dissertation, can handle many of these variants. These algorithms use a simple, fast and space-efficient data structure to keep track of the set of vertices in the second graph to which a given vertex in the first graph may be mapped.

The second problem we consider is *induced subgraph isomorphism*: to determine whether a given graph (the "target graph") contains another given graph (the "pattern graph") as an induced subgraph. We again use a variant of the McSplit algorithm—McSplit-SI—to solve the problem. This version of the algorithm has a specialised version of McSplit's data structure that enables very fast processing of sparse graphs.

Finally, we study the problem of finding, for a given family of graphs, an *induced universal graph*—that is, a graph that contains every member of the family as an induced

subgraph—with as few vertices as possible. Although much progress has been made on asymptotic results on the size of induced universal graphs, almost no work has been done on developing algorithms to solve the problem exactly. This dissertation presents an algorithm for finding minimal induced universal graphs using McSplit-SI as a subroutine, and presents new terms of integer sequences generated using the program. Further, we present a hill-climbing method for finding small (although possibly not optimal) induced universal graphs.

1.2 Structure of the Dissertation

The remainder of this dissertation is structured as follows.

Chapter 2: Background provides context for this dissertation's contributions: definitions, a survey of related work, and a description of the experimental setup.

Chapter 3: Maximum Common Induced Subgraph: the McSplit Algorithm introduces the McSplit algorithm for the maximum common induced subgraph problem and its variants, and carries out experimental comparisons with existing state of the art solvers. We find that the algorithm is more than an order of magnitude faster than prior solvers for unlabelled graphs, including in the variant of the problem where the subgraph must be connected. The chapter also has an empirical study of ordering strategies for the vertices of the two input graphs, and a detailed study—both in theory and practice—of the similarities and differences between McSplit on one hand, and constraint programming and clique approaches on the other.

Chapter 4: Swapping Graphs and Two-Sided Branching in McSplit asks whether and when it is useful to swap the two graphs given as input to McSplit. On the basis of a detailed empirical study, we develop two simple and effective rules for determining when to swap the graphs, and introduce the algorithm McSplit-2S which generalises the idea of swapping by branching on vertices of both input graphs.

Chapter 5: McSplit-SI: An Algorithm for the Induced Subgraph Isomorphism Problem introduces the McSplit-SI algorithm for the induced subgraph isomorphism problem. This uses special data structures designed to work well if the input graphs are sparse. We present an extensive set of experiments comparing the algorithm to state of the art solvers; these experiments demonstrate that the algorithm runs faster than the best existing solvers on several classes of random and structured graphs. In addition, we demonstrate

that McSplit-SI achieves generalised arc consistency on the all-different constraint, and we introduce two variants of the algorithm including one that is optimised for dense graphs. Finally, we return to the maximum common induced subgraph problem, and show that a modified version of McSplit-SI outperforms the state of the art solver for that problem on large, sparse graphs.

Chapter 6: Induced Universal Graphs introduces exact and heuristic algorithms for the minimal induced universal graph problem, and uses these algorithms to add new terms to entries in the On-Line Encyclopedia of Integer Sequences. The chapter includes an experimental study of ordering heuristics for these algorithms.

Chapter 7: Conclusion summarises progress made in the dissertation and outlines directions for future work.

1.3 Publications and Authorship

During the course of my PhD I carried out work on other graph algorithms—in particular, maximum weight clique and treedepth—that is not presented in this thesis. For completeness, the following is a full list of publications since the start of my PhD.

1. McCreesh, C., Prosser, P., and Trimble, J. (2017b). A partitioning algorithm for maximum common subgraph problems. In Sierra, C., editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 712–719. `ijcai.org`

2. McCreesh, C., Prosser, P., Simpson, K. A., and Trimble, J. (2017a). On maximum weight clique algorithms, and how they are evaluated. In Beck, J. C., editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 206–225. Springer

3. Hoffmann, R., McCreesh, C., Ndiaye, S. N., Prosser, P., Reilly, C., Solnon, C., and Trimble, J. (2018). Observations from parallelising three maximum common (connected) subgraph algorithms. In van Hoeve, W. J., editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings*, volume 10848 of *Lecture Notes in Computer Science*, pages 298–315. Springer

4. McCreesh, C., Prosser, P., Solnon, C., and Trimble, J. (2018). When subgraph isomorphism is really hard, and why this matters for graph databases. *J. Artif. Intell. Res.*, 61:723–759

5. Archibald, B., Dunlop, F., Hoffmann, R., McCreesh, C., Prosser, P., and Trimble, J. (2019). Sequential and parallel solution-biased search for subgraph algorithms. In Rousseau, L. and Stergiou, K., editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 20–38. Springer

6. Trimble, J. (2020a). An algorithm for the exact treedepth problem. In Faro, S. and Cantone, D., editors, *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*, volume 160 of *LIPIcs*, pages 19:1–19:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik

7. Trimble, J. (2020b). PACE solver description: Bute-plus: A bottom-up exact solver for treedepth. In Cao, Y. and Pilipczuk, M., editors, *15th International Symposium on Parameterized and Exact Computation, IPEC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 180 of *LIPIcs*, pages 34:1–34:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik

8. Trimble, J. (2020c). PACE solver description: Tweed-plus: A subtree-improving heuristic solver for treedepth. In Cao, Y. and Pilipczuk, M., editors, *15th International Symposium on Parameterized and Exact Computation, IPEC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 180 of *LIPIcs*, pages 35:1–35:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik

9. McCreesh, C., Prosser, P., and Trimble, J. (2020). The Glasgow Subgraph Solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In Gadducci, F. and Kehrer, T., editors, *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, volume 12150 of *Lecture Notes in Computer Science*, pages 316–324. Springer

10. Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., and Trimble, J. (2020a). Certifying solvers for clique and maximum common (connected) subgraph problems. In Simonis, H., editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer

11. Delorme, M., García, S., Gondzio, J., Kalcsics, J., Manlove, D. F., Pettersson, W., and Trimble, J. (2022). Improved instance generation for kidney exchange programmes. *Comput. Oper. Res.*, 141:105707

Chapter 3 of this dissertation contains material from the first publication on the list. I designed and implemented the McSplit algorithm and was the lead author of this paper, which was co-authored by Ciaran McCreesh and Patrick Prosser. Ciaran McCreesh carried out the experimental comparison of McSplit with existing solvers. Section 3.7, which uses the data from these experiments, is a revised and greatly extended version of the experimental section of the paper; the original section was jointly written by all three of the paper's authors.

I am the sole author of all other chapters of this dissertation.

1.4 Earlier Publications by the Author

The following is a list of the author's publications prior to starting work on this dissertation.

1. Dickerson, J. P., Manlove, D. F., Plaut, B., Sandholm, T., and Trimble, J. (2016). Position-indexed formulations for kidney exchange. In Conitzer, V., Bergemann, D., and Chen, Y., editors, *Proceedings of the 2016 ACM Conference on Economics and Computation, EC '16, Maastricht, The Netherlands, July 24-28, 2016*, pages 25–42. ACM

2. McCreesh, C., Prosser, P., and Trimble, J. (2016b). Heuristics and really hard instances for subgraph isomorphism problems. In Kambhampati, S., editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 631–638. IJCAI/AAAI Press

3. McCreesh, C., Prosser, P., and Trimble, J. (2016c). Morphing between stable matching problems. In Rueher, M., editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 832–840. Springer

4. Manlove, D. F., McBride, I., and Trimble, J. (2017). "Almost-stable" matchings in the hospitals / residents problem with couples. *Constraints An Int. J.*, 22(1):50–72

Chapter 2

Background

2.1 Introduction

This chapter presents necessary background material for the remainder of the dissertation. Sections 2.2 and 2.3 introduce topics from graph theory and complexity theory. Section 2.4 formally introduces the problems tacked by this dissertation, and Section 2.5 discusses closely related problems. Section 2.6 is a brief introduction to constraint programming, a field of research that led to many existing algorithms for subgraph isomorphism and maximum common subgraph, and has important connections to the new algorithms described in this dissertation. Sections 2.7 and 2.8 review existing algorithms for subgraph isomorphism and maximum common subgraph. Section 2.9 gives details on how experiments were performed and how the results will be displayed in the following chapters.

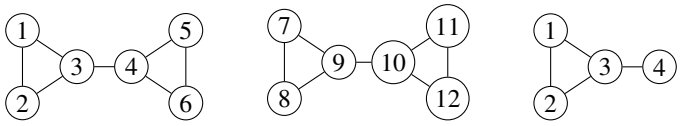
2.2 Graph Theory Terminology

A graph G is a pair (V, E) whose *vertex set* $V = V(G)$ is an arbitrary finite set and whose *edge set* $E = E(G)$ comprises two-elements subset of V. The elements of each edge $e \in E$ are called its *endpoints*, and two vertices that share an edge are said to be *adjacent*.

For example, Figure 2.1(a) shows the graph $G_1 = (V, E)$, where

$$V = \{1, 2, 3, 4, 5, 6\}, E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{4, 6\}, \{5, 6\}\}.$$

A *loop* is an edge from a vertex to itself. We assume throughout this dissertation that graphs have no loops, except in clearly marked sections where we discuss extensions of algorithms to graphs with loops.



(a) G_1 (b) G_2 (c) G_3

Figure 2.1: Example graphs G_1 to G_3

The set of vertices adjacent to vertex $v \in V(G)$ is called the *neighbourhood* of v, denoted $\mathrm{N}_G(v)$. Each element of $N_G(v)$ is said to be a *neighbour* of v. The *degree* of a vertex is size of its neighbourhood. We denote by $\overline{\mathrm{N}}_G(v)$ the *inverse neighbourhood* of v: the set of the vertices other than v itself that are not adjacent to v. We omit the subscript G where there is no ambiguity. In Figure 2.1(a), for example, we have $\mathrm{N}(3) = \{1, 2, 4\}$ and $\overline{\mathrm{N}}(3) = \{5, 6\}$.

We say that graphs G and H are *isomorphic* if they have the same number of vertices and we can relabel the vertices of G to produce graph H. For example, graphs G_1 and G_2 in Figure 2.1 are isomorphic. Formally, an *isomorphism* between graphs G and H is a bijection $f : V(G) \to V(H)$, such that $E(H) = \{\{f(u), f(v)\} \mid \{u, v\} \in E(G)\}$. If such an f exists, we say that G and H are isomorphic.

The *line graph* of a graph G is a graph with a vertex for each element of $E(G)$, such that two vertices are adjacent if and only if their corresponding edges in G share an endpoint.

An *induced subgraph* of $G = (V, E)$ is a graph that has a subset $W \subseteq V$ as its vertex set, and $\{\{u, v\} \in E \mid \{u, v\} \in W\}$ as its edge set; that is, the subgraph includes all edges of G whose endpoints both appear in W. We say that this subgraph is *induced by* W. Graph G_3 of Figure 2.1 is thus the subgraph of G_1 induced by the set $\{1, 2, 3, 4\}$. We say that G *contains* H *as an induced subgraph* if H is isomorphic to an induced subgraph of G. A *subgraph* (without the "induced" qualifier) is an induced subgraph with zero or more edges deleted.

A *common induced subgraph* of graphs G and H is an induced subgraph of G which is isomorphic to an induced subgraph of H. In Figure 2.2, a common induced subgraph of the two graphs (which is induced by $\{1, 2, 3, 4\}$) is highlighted on the first graph, and its isomorphic subgraph is highlighted on the second graph. It is often convenient to summarise a common induced subgraph and its associated isomorphism f by the *mapping* $\{(v, f(v)) \mid v \in W\}$, where $W \subseteq V$ is the set of vertices that induces the common subgraph. In our example, $M = \{(1, 6), (2, 7), (3, 8), (4, 9)\}$.

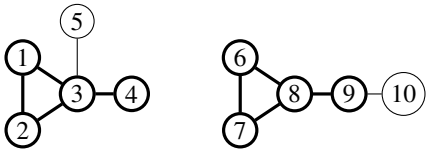


Figure 2.2: A pair of graphs with a maximum common induced subgraph highlighted

A *maximum common induced subgraph (MCIS)* of G and H is a common induced sub-

graph of G and H whose vertex set is as large as possible; the common induced subgraph in Figure 2.2 is an example of an MCIS.

A *connected graph* is a graph $G = (V, E)$ such that for all $u, v \in V$, we can reach vertex v by beginning at vertex u and traversing a sequence of edges.

A *clique* is a set of vertices that are mutually adjacent, and the *maximum clique problem* is, given a graph G, to find a clique in G with as many vertices as possible.

A *labelled graph* (or *network*) is a triple (G, f_v, f_e) where G is a graph, the *vertex label function* f_v is a function with domain $V(G)$, and the *edge label function* f_e is a function with domain $E(G)$. An isomorphism between labelled graphs G and H is required to map each vertex of G to a vertex of H with the same label, and each edge of G to an edge of H with the same label.

A *directed graph* is a pair (V, A), where V is the vertex set and A, a set of ordered pairs of elements of V, is the arc set. The definitions of *induced subgraph*, *isomorphism* and *maximum common induced subgraph* for directed graphs are analogous to their counterparts for graphs. Figure 2.3 shows two directed graphs with a common induced subgraph highlighted.

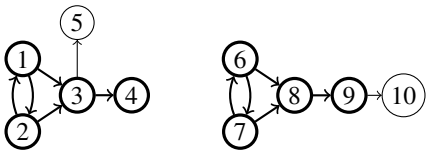


Figure 2.3: A pair of directed graphs with a maximum common induced subgraph highlighted

For arc (u, v), we say that u is the *source* vertex and v is *target* vertex; both u and v are called endpoints.

The two most common representations of a graph in computer memory are an *adjacency matrix* and *adjacency lists*. For simplicity, we assume that the vertices of a graph G are numbered sequentially from 0 to $|V(G)| - 1$. The adjacency matrix representation is an $n \times n$ array A of Boolean values, such that $A[v][w]$ is *true* if and only if v and w are adjacent. The adjacency list representation is a list of n lists, with list v containing the neighbours of vertex v. The former representation allows us to test for the existence of an edge in constant time. The latter representation is space-efficient for sparse graphs and allows us to iterate over the neighbourhood of a vertex in linear time.

There are numerous models for generating graphs randomly. Several are used in this dissertation, and in general we will define these models at the point of use. A very common random graph model is the Erdős-Rényi $G(n, p)$ model: for a given order $n \geq 0$ and edge probability p ($0 \leq p \leq 1$), the graph is generated by creating a set of n nodes and adding an edge between each pair of nodes with independent probability p.

2.3 Computational Complexity

This section gives a very brief, informal description of NP-*hardness* and related concepts. A more complete introduction can be found in the classic monograph of Garey and Johnson (1979), while Moore and Mertens (2011) give a broader, and very entertaining, treatment.

Consider the CLIQUE problem: given a graph G and a natural number k, does G contain a k-vertex clique? If the answer is "yes", we can give a proof of this fact by listing the k vertices in this clique. A sceptic may verify in $k(k-1)/2$ steps that the k vertices are indeed pairwise adjacent using an adjacency matrix representation of the graph. Any decision problem—like CLIQUE—such that there is a proof of "yes" instances that can be verified in polynomial time is said to be in the complexity class NP.

Although we can verify a "yes" instance of CLIQUE in polynomial time, there is no known way to solve the problem (deterministically) in polynomial time. Indeed CLIQUE is, in an important sense, one of the hardest problems in NP (Karp, 1972): every instance of a problem in NP can be transformed, or *reduced*, to a CLIQUE instance in polynomial time, such that the original instance and the reduced instance give the same answer. Any problem in NP that has this property is called NP-*complete*.

A problem H is NP-*hard* if it is at least as hard as any problem in NP; that is, if any problem in NP can be solved in polynomial time under the assumption that we have an oracle that can solve H in constant time. We will see in the next section that all three problems considered in this dissertation are NP-hard.

2.3.1 Coping With Complexity

There are several broad options for dealing with the complexity of NP-hard problems. Here we briefly describe three common approaches: approximate algorithms, FPT algorithms, and off-the-shelf solvers.

In the case of optimisation problems, it is sufficient for some applications to seek a solution that is good but not necessarily optimal. Such algorithms may be divided into *approximation algorithms* that guarantee that the solution's objective value will be within a constant factor of the optimum (Vazirani, 2001), and *heuristics* that provide no such guarantee (Talbi, 2009). These classes of algorithms are out of scope for this disseration, as we will consider only exact solutions.

One approach for finding an exact solution is to use a *fixed-parameter-tractable (FPT) algorithm*, whose run time is exponential in some parameter of the input (such as treewidth or vertex cover number if the input is a graph) but polynomial in the size of the input (Niedermeier, 2006). We will discuss an existing FPT algorithm for maximum common induced

subgraph in Section 2.8.3, but this will not be included in our experimental comparisons as the author is not aware of any existing implementation.

An alternative approach is to model a problem as a constraint program or integer program, and solve it using an off-the-shelf solver. Integer programming solvers, for example, have achieved state of the art results on graph problems such as maximum weight clique on dense random graphs (Segundo et al., 2019) and kidney exchange (Dickerson et al., 2016). However, off-the-shelf solvers run much more slowly and use more memory than dedicated solvers for subgraph isomorphism and maximum common subgraph problems (McCreesh, 2017; Trimble et al., 2018) — at least when modelling the problems in "obvious" ways. We therefore do not include such solvers in our experimental comparisons.

The algorithms introduced in this dissertation, like the existing state-of-the-art algorithms for the problems we consider, do not use any of the approaches described in this subsection. Rather, we use dedicated, exact solvers for each problem.

2.4 Problems Covered by This Dissertation

This dissertation presents algorithms for the following three problems. The first and third are optimisation problems; the second is a decision problem.

- **Maximum common induced subgraph (MCIS).** Find a maximum common induced subgraph of two given graphs. We have seen an example in Figure 2.2: the two graphs have a maximum common induced subgraph with four vertices.
- **Induced subgraph isomorphism (ISIP).** Given graphs G (the "pattern graph") and H (the "target graph"), determine whether H contains G as an induced subgraph; if so, return the mapping corresponding to an isomorphism from G to an induced subgraph of H. Figure 2.4 gives two examples of instances. The first is unsatisfiable; the solution $\{(1, a), (2, b), (3, d)\}$ for the second instance is shown.
- **Minimal induced universal graph.** Given a family of graphs $\mathcal{F}$, find a graph with as few vertices as possible that contains each member of $\mathcal{F}$ as an induced subgraph. Figure 2.5(a) shows an example instance: the family of graphs $\{C_3, C_4, C_5\}$. Figure 2.5(b) shows a corresponding solution with six vertices. This is shown three times, with an induced copy of each input graph highlighted.

Each of the three problems can be seen to be NP-hard by a simple reduction from CLIQUE, which may be seen as follows. Given graph G and natural number k, we can determine whether G contains a clique of size k by solving either of the first two problems

(a) Unsatisfiable (b) Satisfiable

Figure 2.4: Two instances of the induced subgraph isomorphism problem

(a) Instance (b) A solution

Figure 2.5: An instance of the minimal induced universal graph problem, and a solution

with the complete graph K_k as the first graph and G as the second graph, or by solving the third problem with the family $\{K_k, G\}$ as the input.

All three problems remain NP-hard if the input graphs are both forests (Garey and Johnson, 1979), or are both series-parallel graphs (Syslo, 1982).

Each problem has a natural extension to labelled graphs where we require the isomorphism to preserve labels on vertices and edges. Likewise, each problem has a variant for directed graphs. For the first two problems, this dissertation presents algorithms for both labelled graphs and directed graphs; for the third problem we only consider unlabelled, undirected graphs.

All three problems have natural enumeration counterparts: to find all maximum common induced subgraphs, all induced subgraph isomorphisms, and all minimal induced univeral graphs. As we will see in the following chapters, our solvers can be straightforwardly extended to handle these enumeration versions.

2.5 Related Problems

This section briefly introduces three problems that are closely related to maximum common induced subgraph and induced subgraph isomorphism.

The *graph isomorphism problem* is to determine whether two given graphs are isomorphic. It can be viewed as a version of the induced subgraph isomorphism problem that returns "no" if the input graphs do not have the same number of vertices. It is unknown whether the problem is NP-complete, and it is also unknown whether it can be solved in polynomial time, although a quasipolynomial-time algorithm has been proposed (Babai, 2016). Most state of the art algorithms for graph isomorphism, such as Nauty and Traces (McKay and Piperno, 2014), solve the problem by computing a canonical representation of each graph and compar-

ing these representations. This method has the advantage that the canonical representations may then be re-used for comparisons with other graphs. Considering only pairwise comparisons of graphs, a 2001 study found that the induced subgraph isomorphism solver VF2 outperformed Nauty on several classes of graphs (Foggia et al., 2001). While we do not consider graph isomorphism further in this dissertation, it would be an interesting topic of future work to test whether the current state of the art subgraph isomorphism solvers (including the McSplit-SI algorithm that will be introduced in Chapter 5) outperform the best graph isomorphism solvers on some interesting classes of graphs.

Induced subgraph isomorphism and maximum common induced subgraph each have a non-induced counterpart. The *subgraph monomorphism problem* is to determine whether graph G is isomorphic to a subgraph of graph H, without the requirement that the subgraph be induced. Some papers use the name *subgraph isomorphism* to refer to subgraph monomorphism (McCreesh and Prosser, 2015), while others use the same name to refer to induced subgraph isomorphism (Carletti et al., 2018). To avoid confusion, the ambiguous term "subgraph isomorphism" (without the "induced" qualifier) will not be used in the sequel.

The *maximum common edge subgraph (MCES)* problem is to find a subgraph of graph G with as many edges (rather than vertices) as possible that is isomorphic to a subgraph of graph H, again without requiring that the subgraphs be induced. There is a non-trivial but close connection between this problem and MCIS: by a result due to Whitney (1932), we can find the MCES two graphs by searching for an MCIS on their line graphs. This approach has a small complication: the triangle graph K_3 and the claw graph $K_{1,3}$ (Figure 2.6) both have K_3 as their line graph, and we must therefore check during search that no strongly connected component of G that is a triangle (resp. claw) is mapped to a claw (resp. triangle) in H.

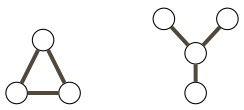


Figure 2.6: The graphs K_3 and $K_{1,3}$.

Several algorithms for MCES, such as RASCAL (Raymond et al., 2002a), use this line graph method, as we will discuss further in the next section. Vismara and Valery (2008) describe how this technique can be used on labelled graphs.

The present author's preliminary work on using the McSplit algorithm described in this thesis for MCES using line graphs shows promising results (Trimble et al., 2018). This would be a useful direction for future work, but it is not pursued further in this dissertation.

A generalisation that can be applied to any variant of the maximum common subgraph problem is to find a maximum common subgraph of an arbitrary set of graphs, rather than

only of a pair of graph. The MCES version of this problem has been studied in the context of sets of molecules (Dalke and Hastings, 2013).

2.6 Constraint Programming

We now turn from problems to solution methods. As discussed previously, each problem studied in this dissertation is NP-hard. This section reviews *constraint programming*, a very general framework for modelling and solving NP-hard problems which is the basis of many existing solvers for maximum common subgraph and induced subgraph isomorphism, and which has close connections to the McSplit algorithms that we will introduce.

2.6.1 Introduction to Constraint Programming

A *constraint satisfaction problem (CSP)* is a problem of finding a value for each of a finite set of variables with finite domains, subject to a set of constraints. This framework is extremely broad, and has applications in fields including logistics, scheduling, bioinformatics, and discrete mathematics. The field of *constraint programming (CP)* is concerned with modelling and solving CSPs and related problems.

Many algorithms for MCIS and induced subgraph isomorphism model the problems as CSPs. Moreover, the McSplit family of algorithms that will be introduced in the next three chapters are closely related to a constraint programming approach. This section gives a brief introduction to constraint programming; a detailed treatment can be found in the *Handbook of Constraint Programming* (Rossi et al., 2006).

Formally, a CSP is a triple $\langle X, D, C \rangle$, where

- X is a tuple of *variables* $X = \langle x_1, x_2, \ldots, x_n \rangle$,
- D is a corresponding tuple of finite *domains* $D = \langle D_1, D_2, \ldots, D_n \rangle$ such that each x_i must take a value in D_i,
- C is a set of constraints, each of which acts on a subset of X and restricts the values that may be taken simultaneously by those variables.

A solution to a CSP is an assignment of domain values to each variable in X satisfying the constraints C.

Broadly, algorithms for solving CSPs can be classed as *complete* (guaranteed to find a solution if one exists) or *incomplete*. The former category is the focus of this dissertation.

All of the constraint programming approaches to MCIS and induced subgraph isomorphism use some form of backtracking search, and therefore that will be the focus of this section.

To illustrate the process of solving a CSP by backtracking, we use a canonical introductory problem, n-queens. For our example, we will consider the case $n = 4$. The problem is to place four queens on a 4×4 chessboard such that no two queens are in the same row, column, or diagonal.

The first step is to model the problem as a CSP. Many models have been proposed for the n-queens problem (Smith, 2006); we will use a simple model with one variable per row (x_1 to x_4) and four values, $\{a, b, c, d\}$, for the four columns. There are two types of constraint. First, there is a single *allDifferent* constraint to ensure that the four variables take different values (and thus the queens are in different columns). Second, there is a constraint for each of ten diagonals ensuring that at most one queen is placed on that diagonal. (The one-queen-per-row requirement is met without further constraints, since each variable must take exactly one value.)

Figure 2.7 shows a search tree for this problem. This tree is traversed in depth-first order, and is not explicitly stored. At node (A), no assignments of values to variables have been made. At node (B), a first tentative assignment has been made: $x_1 = a$. A key part of most backtracking CP solvers is the interleaving of search (trying possible values for a variable in turn) with *constraint propagation* (deleting values that can be determined not to appear in any solution that extends the current set of assignments). For our example, we will use the simplest form of propagation: *forward checking*. This removes from the domains of the unassigned variables every value that conflicts with some existing assignment. The × symbol in a square in the figure indicates that a value has been removed from a domain.

At node (C), the assignment $x_2 = c$ has been made. After forward checking, no values are left in D_3; thus there is no solution with $x_1 = a$ and $x_2 = c$. The algorithm backtracks and tries $x_2 = d$ (node (D)). This process of search, forward checking, and backtracking when a domain becomes empty is continued (nodes (E) to (I)) until reaching the solution shown in node (I).

For many problems, more costly propagation algorithms can remove more values from domains than forward checking, achieving stronger *levels of consistency* (a level of consistency is a guarantee that a value will be removed from a domain if some condition is met). It is common for constraint programming solvers to maintain *arc consistency*, guaranteeing that no value will remain in a domain if there is some constraint involving two variables that cannot be satisfied if that value is taken. *Generalised arc consistency* extends this to constraints involving arbitrarily many variables. The canonical example is Régin's polynomial-time filtering algorithm for the allDifferent constraint, which uses a matching algorithm (Régin, 1994).

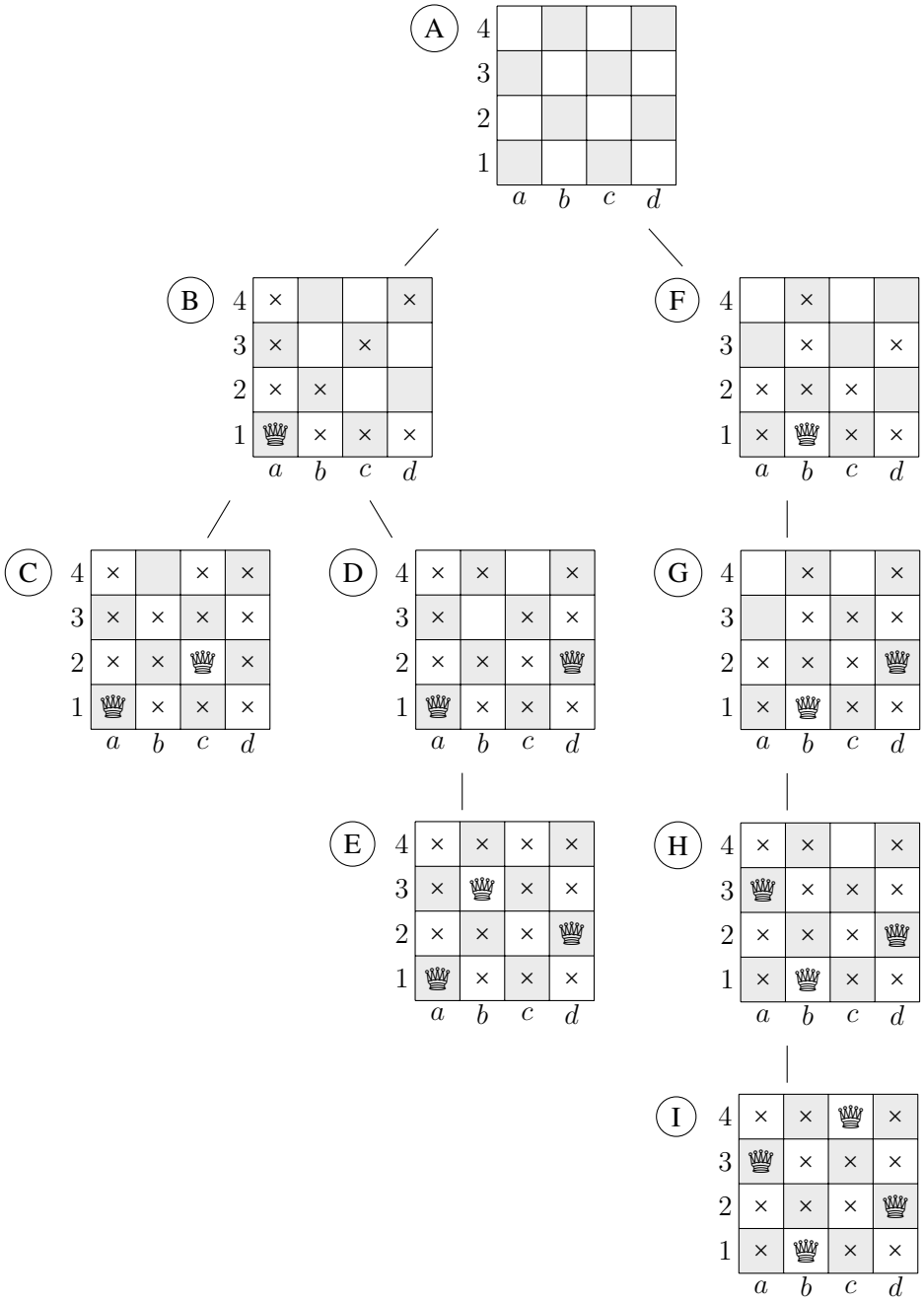


Figure 2.7: The search tree of a backtracking CP algorithm for the 4-queens problem.

The forward-checking search described above can be modified in numerous ways, many of which are significant research topics in their own right. To give four examples:

- We could break symmetries (informally, to ignore parts of the search tree that are equivalent to ones already visited) (Gent et al., 2006). In the 4-queens example, we can disregard $x_1 = c$ and $x_1 = d$, since any solution with these assignments is the same as a solution with $x_1 = b$ and $x_1 = a$, respectively, after reflecting the chessboard horizontally.

- We could use a *variable ordering heuristic* to choose which variable to branch on at each search node, and a *value ordering heuristic* to choose the order in which the child nodes of a search node are visited. For example, a very common variable ordering heuristic is to choose the variable with fewest remaining values (Golomb and Baumert, 1965).
- We could learn new constraints during search that do not affect the solution, and use these to prune the search tree (Katsirelos and Bacchus, 2005).
- We could traverse the search tree in an order other than depth-first search to quickly explore diverse regions of the search space. An early example is limited discrepancy search (Harvey and Ginsberg, 1995); another strategy is to combine the learning of constraints with periodic restarts of the backtracking search and randomised variable- and value-ordering heuristics (Lecoutre et al., 2007; Archibald et al., 2019).

2.6.2 A CP Model for Induced Subgraph Isomorphism

We now introduce a simple CSP formulation for induced subgraph isomorphism based on Ullmann (1976). Let G and H be the pattern and target graphs. We have a variable x_v for each $v \in V(G)$. The domain of each variable is $V(H)$, representing the vertices in H to which each vertex in G may be mapped. We have an allDifferent($\{x_v \mid v \in V(G)\}$) constraint to ensure that all of the vertices in the pattern graph are mapped to different vertices in the target graph. Finally, we have two sets of constraints to ensure that edges in the pattern graph are mapped to edges in the target graph and non-edges are mapped to non-edges:

- For all distinct $v, w \in V(G)$ such that $v \in N_G(w)$, we have $x_v \in N_H(x_w)$.
- For all distinct $v, w \in V(G)$ such that $v \notin N_G(w)$, we have $x_v \notin N_H(x_w)$.

Figure 2.9 shows one branch of the search tree for this CSP, using graphs G and H in Figure 2.8. Again, we show each domain as a row, with the symbol × indicating that a value has been removed. The letter M indicates an assignment.

At node (B) of the search tree, variable x_1 has been assigned the value a, corresponding to vertex 1 of graph G being mapped to vertex a of graph H. The allDifferent propagator removes a from the remaining four domains. Values b to e are divided between those four domains—variables x_4 and x_5 keep the vertices adjacent to a in graph H because vertices 4 and 5 are adjacent to vertex 1 in graph G. At this and all of the other search nodes, any two variables have either identical domains or disjoint domains; in the next chapter this property will be exploited to develop what is essentially a compressed domain store in the McSplit algorithm.

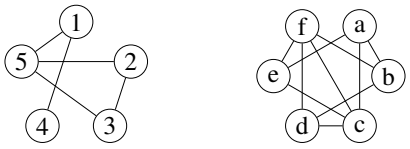


Figure 2.8: Example graphs G and H

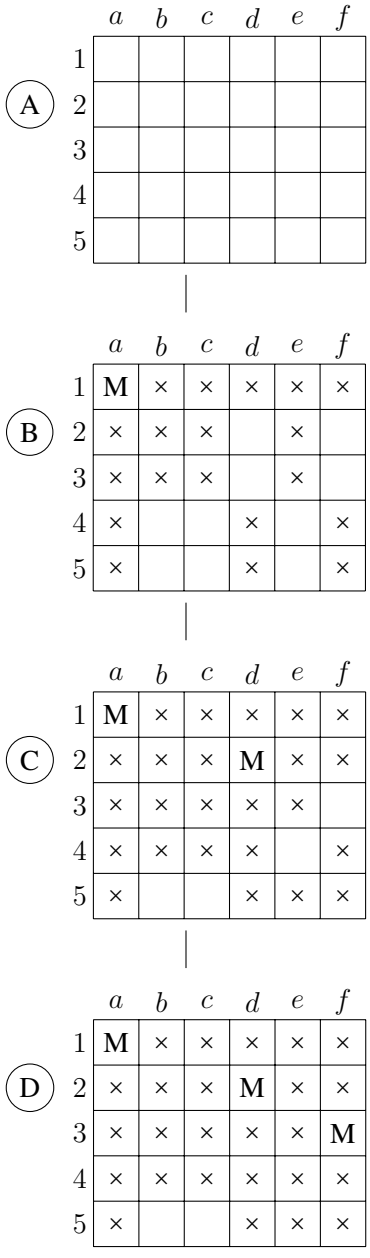


Figure 2.9: A branch of the search tree using forward checking for a simple constraint programming model of induced subgraph isomorphism

2.6.3 Constraint Optimisation Problems

The previous subsection described constraint satisfaction problems, which are a class of decision problem. We now turn to *constraint optimisation problems (COP)*, in which we

must optimise some objective function. The term "constraint optimisation problem" has multiple definitions in the literature; we will define a COP simply as a modified CSP in which the goal is to minimise or maximise a specific variable rather than determine satisfiability.

Branch and bound (Land and Doig, 2010) is a general technique for solving optimisation problems by backtracking search. We will describe branch and bound in the context of maximisation problems. During search, we maintain an incumbent—the best solution found so far. Each time a solution S whose objective value is larger than the incumbent is found, the incumbent's value is updated to S. At each search node we calculate an upper bound on the best objective value that can be obtained by extending the current partial solution. If this upper bound is not greater than the incumbent's objective value, we know that exploring the subtree below the current search node cannot possibly increase the incumbent, and it is therefore safe to backtrack.

An alternative to branch and bound is to solve the maximisation problem by a sequence of decision problems. A simple way to do this, assuming that the variable to be optimised is an integer variable, is to start with an optimistic target and decrement this each time our search fails. Beginning at a known upper bound b for the optimal objective value, we use search to answer the question "does a solution with objective value b exist?" If the answer is "yes", the algorithm terminates. Otherwise, a solution with objective value $b - 1$ is sought, then $b - 2$, and so on until a solution is found.

In the next chapter, we compare these two techniques empirically for the McSplit algorithm.

2.7 Algorithms for Induced Subgraph Isomorphism

This section reviews the two main lineages of practical algorithms for induced subgraph isomorphism. Section 2.7.1 reviews algorithms based on constraint programming. Many CP algorithms for induced subgraph isomorphism are based on algorithms for subgraph monomorphism, and this subsection therefore touches on the latter problem. Section 2.7.2 reviews algorithms from the pattern recognition community. These are in general simple backtracking algorithms without the domain store used by constraint programming algorithms. Finally, Section 2.7.3 reviews other approaches.

2.7.1 Algorithms Based on Constraint Programming

The algorithm of Ullmann (1976) solves only the subgraph monomorphism problem. The constraint programming model it uses is essentially the one described in Section 2.6.2, with

the non-neighbour constraints removed. (We could trivially add these constraints to Ullmann's algorithm to solve the induced problem.) The algorithm uses forward checking for the allDifferent constraint and maintains arc consistency on the adjacency constraints. Bitsets—Boolean arrays packed into arrays of larger words—are used to represent domains and rows of each adjacency matrix, allowing very fast updates to domains, particularly if the pattern and target graphs are dense.

McGregor (1979) introduces a modified version of Ullmann's algorithm. McGregor finds experimentally that the cost of maintaining arc consistency throughout search outweighs the benefit; as a compromise, McGregor enforces arc consistency only after assigning the first pattern-graph vertex, and uses forward checking at deeper levels of the search tree.

The subsequent trend has been largely in the direction of more sophisticated filtering at each search node. This trend begins with the RESYN system (Vismara et al., 1992; Régin, 1995), whose filtering algorithm achieves generalised arc consistency on the allDifferent constraint.

Solnon (2010) proposes an algorithm for LAD (local all different) filtering, which propagates the following constraint: if $v \in V(G)$ is mapped to $w \in V(H)$, then the neighbours of v in G must all be mapped to different values, and these values must all belong to the set $N_H(w)$. This strengthens the filtering of the nRF+ algorithm (Larrosa and Valiente, 2002), and can be performed in $O(|V(G)| \cdot |V(H)| \cdot \Delta_G^2 \cdot \Delta_H^2)$ time, where Δ_G and Δ_H are the maximum degrees of G and H.

The SND algorithm (Audemard et al., 2014) extends LAD by counting the number of length-k paths between pairs of vertices for small values of k, and enforcing within a LAD-style constraint that a pair of pattern vertices connected by p paths of length k may not be mapped to a pair of target vertices connected by fewer than p paths of length k. PathLAD (Kotthoff et al., 2016) incorporates a subset of the SND filtering into the LAD algorithm.

The Glasgow Subgraph Solver (McCreesh and Prosser, 2015; McCreesh et al., 2020) (henceforth Glasgow) reverses this trend towards stronger—and typically also slower—inference at each search node. The algorithm aims to achieve most of the domain filtering of LAD and SND with much less effort per search node. The Glasgow algorithm introduces *supplemental graphs*—graphs that encode the number of paths of a given length between pairs of vertices—which provide additional filtering very cheaply. Rather than achieving generalised arc consistency on the allDifferent constraint, Glasgow uses a new "counting" allDifferent propagator which runs very fast without giving any consistency guarantee. Like Ullmann's algorithm, Glasgow uses bitsets to represent domains and rows of adjacency matrices. Glasgow was the fastest solver for hard induced subgraph isomorphism instances in a recent experimental evaluation by Solnon (2019).

Symmetry breaking Zampelli et al. (2007) and Zampelli (2008) show that the variable and value symmetries in the subgraph isomorphism problem can be completely broken using the automorphism groups of the pattern and target graph graphs. Moreover, the authors show how to break local symmetries that emerge during search. A solver with symmetry breaking was shown to be able to solve more instances than the same solver without symmetry breaking. Techniques based on those of Zampelli et al. could be applied to any subgraph isomorphism solver based on constraint programming, and the use of such techniques is understudied; neither McSplit-SI nor any of the state-of-the-art algorithms to which we compare it in our experiments uses symmetry breaking.[1] This would be a fruitful area for future work.

2.7.2 Pattern Recognition Algorithms

Algorithms for induced subgraph isomorphism from the pattern recognition (PR) community take a lightweight approach—they do not store domains and perform minimal work at each search node, resulting in algorithms with very low memory requirements that can visit many search nodes per second, but often have to visit many more search nodes in total than CP algorithms.

Like the CP solvers, the VF3 (Carletti et al., 2018) and RI (Bonnici et al., 2013; Bonnici and Giugno, 2017) solvers explore a search tree, adding one pair of vertices to the mapping at each search node. However, the data structures maintained by these algorithms are much more lightweight than those of any CP solver. VF3 does not maintain a domain for each vertex in the pattern graph; it simply store a partition of the vertex set of each graph into three sets: (A) vertices that have been mapped already, (B) unmapped vertices that are adjacent to at least one mapped vertex, and (C) all other unmapped vertices. If the B (resp. C) set of the target graph is smaller than the B (resp. C) set of the pattern graph, the algorithm can backtrack. RI is simpler still; it does not distinguish between sets B and C.

RI-DS (Bonnici et al., 2013) is a version of RI that stores a domain for each vertex of G. These domains are filtered before search, but the algorithm does not perform additional filtering during search. Therefore, the algorithm is more akin to the backtracking VF3 and RI solvers than to a constraint programming solver.

2.7.3 Other Approaches

Sussenguth's (1965) backtracking algorithm for induced subgraph isomorphism pre-dates

[1] Recent versions of the Glasgow Subgraph Solver have a symmetry breaking option for the pattern graph which is described as "very experimental". We do not use this in our experiments.

the field of constraint programming but shares the principle of interleaving search and propagation. The algorithm uses a list of rules to generate pairs $\langle S_G, S_H \rangle$ such that $S_G \subseteq V(G)$, $S_H \subseteq V(H)$, and each vertex in S_G may only be mapped to vertices in S_H. For example, pattern-graph vertices of degree 3 may only be mapped to target-graph vertices of degree at least 3. Further pairs of sets are generated based on adjacency to members of these sets S_G and S_H. The search tree of Sussenguth's algorithm is incomparable to that of other CP algorithms, and unfortunately an implementation of Sussenguth's algorithm is not readily available. (To preview the family of algorithms presented in this dissertation, the McSplit algorithm shares with Sussenguth's algorithm the approach of storing pairs of sets $\langle S_G, S_H \rangle$ representing vertices that may be mapped to one another. However, McSplit uses a very different data structure to store these sets, taking advantage of the fact that the sets used by McSplit are guaranteed to be disjoint.)

Cortadella and Valiente (2000) represent an instance of the subgraph isomorphism problem as a binary decision diagram (BDD), and solve this BDD in order to find all solutions to the subgraph isomorphism problem. A limitation is that for large pattern graphs, the size of the BDDs quickly becomes unmanageable; the experiments in the paper only consider pattern graphs with 10 or fewer vertices.

Filter/verify Some graph database systems use an indexing technique to compute information about each target graph in a database, in the hope that this can be used to quickly prove unsatisfiability for some pattern graphs. We do not consider this technique further because it not intended for the one-shot problem of testing whether a given graph contains another graph as in induced subgraph. Moreover, it is unclear whether the filter/verify technique provides any additional benefit over state of the art subgraph isomorphism solvers; see McCreesh et al. (2018) for a sceptical review.

2.8 Algorithms for Maximum Common Induced Subgraph

Exact algorithms for the maximum common induced subgraph problem may be divided into four broad categories: algorithms that solve the maximum clique problem on a derived graph called the association graph, algorithms based on constraint programming, FPT algorithms, and algorithms that enumerate subgraphs of graph G and use a subgraph isomorphism algorithm to search for each subgraph in graph H.[2] The first two of these categories are the most commonly used ones for practical solvers. We now review each of the four categories.

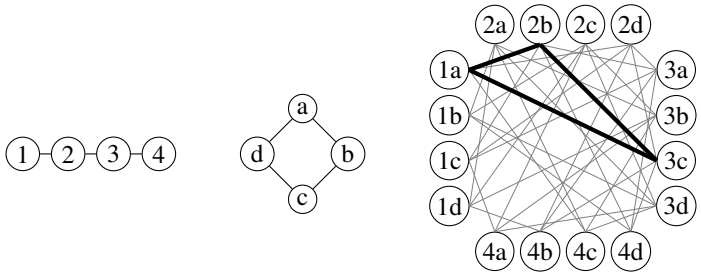
[2]The maximum common *edge* subgraph has additionally been solved using linear programming (Marenco, 1999; Bahiense et al., 2012).

2.8.1 Clique Algorithms

The *association graph* (also known as the weak modular product graph) of graphs G and H, which we will call A, is defined as follow. The vertex set of A is $V(G) \times V(H)$. Two vertices (v, w) and (v', w') of A are adjacent if and only if $v \neq v'$, $w \neq w'$, and either (1) v and v' are adjacent in G and w and w' are adjacent in H or (2) v and v' are not adjacent in G and w and w' are not adjacent in H.

Every common subgraph of G and H, which may be viewed as a mapping M, corresponds to a clique in A with vertex set M. Thus, we can find a maximum common subgraph of G and H by finding a maximum clique in A (Levi, 1973).

Figure 2.10 shows two graphs and their association graph. The thick edges in the association graph correspond to a maximum common induced subgraph.



(a) G (b) H (c) Association graph

Figure 2.10: Two graphs and their association graph. The edges of a maximum clique in the association graph are highlighted. This corresponds to a maximum common induced subgraph: the subgraph of G induced by $\{1, 2, 3\}$, which is isomorphic to the subgraph of H induced by $\{a, b, c\}$.

A number of early algorithms for finding a maximum common subgraph use a *maximal* clique algorithm (Bron and Kerbosch, 1973) for the maximisation problem (Durand et al., 1999; Vismara and Valery, 2008).

Bunke et al. (2002) use the maximum clique algorithm of Balas and Yu (1986) to find a maximum common subgraph. This algorithm greedily colours vertices during search to obtain a good upper bound. The RASCAL algorithm (Raymond et al., 2002b)—which solves MCES by finding a maximum clique in the modular product graph of line graphs—tries several other heuristic colouring methods in addition to greedy colouring, and uses the one that gives the tightest upper bound. Unfortunately the authors were unable to provide code for RASCAL.

McCreesh et al. (2016a) use a state of the art maximum clique algorithm, MCSa1 (Prosser, 2012; Tomita et al., 2013), to solve the maximum common induced subgraph problem. In addition to greedy colouring, this algorithm uses a static vertex order based on degree, and has very fast operations on sets due to the use of bitsets and the separate compilation of code

for different bitset sizes. The paper also introduces a modified version of the algorithm to solve the maximum common *connected* induced subgraph problem.

2.8.2 Constraint Programming Algorithms

McGregor (1982) introduced a branch and bound algorithm for the maximum common *edge* subgraph problem in which a vertex of G is mapped to a vertex of H at each node of the search tree. A matrix of Boolean values indicates the set of edges in H to which each edge in G may be mapped; this may be viewed as a simple domain store. Bunke et al. (2002) reports using McGregor's algorithm to solve MCIS, without giving full details of the modifications made to the algorithm.

Vismara and Valery (2008) give a constraint model for solving maximum common connected edge subgraph based on solving MCIS on the line graphs of the two input graphs; effectively, this is the first formal constraint program for MCIS. Like the CP model for induced subgraph isomorphism in Section 2.6.2, we have a variable x_v for each $v \in V(G)$. Each domain is $V(H) \cup \perp$, where $\perp$ is a special value indicating that a vertex is unmapped. The objective is to minimise the number of $\perp$ values. The model uses binary constraints to ensure that no two vertices in G are mapped to the same vertex in H. The adjacency and non-adjacency constraints from our induced subgraph isomorphism model are replaced by the following, in order to allow any vertex to be mapped to $\perp$.

- For all distinct $v, w \in V(G)$ such that $v \in N_G(w)$, we have $x_v = \perp$, $x_w = \perp$, or $x_v \in N_H(x_w)$.
- For all distinct $v, w \in V(G)$ such that $v \notin N_G(w)$, we have $x_v = \perp$, $x_w = \perp$, or $x_v \notin N_H(x_w)$.

Ndiaye and Solnon (2011) present a CP model that shares the variables of the Vismara and Valery model, but replaces the binary difference constraints with a single global "soft allDiff" constraint (Petit et al., 2001). This ensures that distinct vertices in G are mapped to distinct vertices in H, and uses a matching algorithm to calculate a stronger upper bound than the one given by the Vismara and Valery model. Ndiaye and Solnon carry out detailed experiments using different levels of consistency for the adjacency and soft allDiff constraints. The soft allDiff constraint, if it is used to calculate an upper bound but not to filter domains, causes the algorithm to run several times faster than the simple difference constraints used by Vismara and Valery. Comparing forward checking (FC) and maintaining arc consistency (MAC) on the adjacency constraints, FC outperforms MAC on unlabelled instances while MAC outperforms FC on labelled instances.

McCreesh et al. (2016a) add a connectedness constraint to the CP model of Ndiaye and Solnon (2011). This is found to perform better overall than ensuring connectedness simply by branching on vertices adjacent to some already-mapped vertex.

Finally, the $k\downarrow$ algorithm of Hoffmann et al. (2017) solves the optimisation problem as a sequence of decision problems, with each of these subproblems solved by a modified Glasgow algorithm (McCreesh and Prosser, 2015). It remains possible to filter domains using supplemental graphs, albeit with weaker propagation than for induced subgraph isomorphism.

2.8.3 An FPT Algorithm

Abu-Khzam et al. (2017) give an algorithm for maximum common induced subgraph parameterised by vertex cover number that runs in $2^{O(k \log k)}$ time, where k is the sum of the vertex cover numbers of the two input graphs.

2.8.4 Subgraph Enumeration Algorithms

Finally, we mention briefly a very different technique that has been used in a number of papers to find a maximum common connected subgraph between two or more graphs representing molecules (Armitage and Lynch, 1967; Takahashi et al., 1987; Dalke and Hastings, 2013). In this type of algorithm, connected subgraphs of the first graph are enumerated by backtracking search. A subgraph isomorphism solver is used to test whether each of these subgraphs appears in each of the other input graphs. In some of the algorithms in this category, the subgraphs of G are tested for isomorphism with previously generated subgraphs in order to break symmetries. Unfortunately, we have been unable to find an implementation of this technique for MCIS in general graphs to use in our experimental evaluation.

2.9 Experimental Details

2.9.1 Experimental Setup

The primary experimental setup used for this dissertation was a cluster of machines with dual Intel Xeon E5-2697A v4 CPUs with thermal scaling disabled and 512GBytes RAM. All code was compiled with GCC version 9.4.0. All algorithms used are sequential; 32 instances (one per CPU core) were solved concurrently. This setup was used except where indicated otherwise: namely, in Section 3.7 and Chapter 6.

All of the new algorithms presented were implemented by the present author in C++ (Chapters 3 to 5) and Python (Chapter 6). All solvers by other authors were implemented either in C or C++. Throughout, the compiler flag `-O3` (optimization level 3) was used.

2.9.2 Conventions for Plots

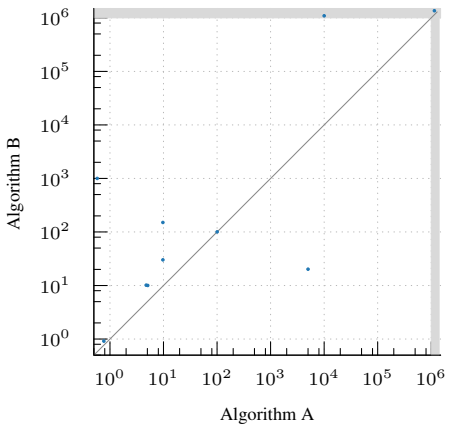
Scatter plots and cumulative plots are used frequently in this dissertation to compare the run times of algorithms. To illustrate how these will be used, consider the run times in milliseconds for ten hypothetical instances, with one row per instance, in Figure 2.11(a).

Figure 2.11(b) shows a scatter plot of these run times. Log scales are used for run times throughout this dissertation. Since run times are measured at a 1 millisecond resolution, the points are jittered by adding a uniform random number in the range $[-1/2, 1/2)$ to each time. This reduces over-plotting and thus helps to show the distribution of points, with only a negligible effect on the plots' precision. Run times of zero are plotted randomly in the range $[1/2, 1)$ because zero cannot appear on a log scale. Timeouts (exceeding 1000 seconds) appear in the grey band just beyond a run time of 1000 seconds; again, we jitter these within the band to minimise over-plotting. The line $x = y$, where the two algorithms have the same run time, is shown for convenience.

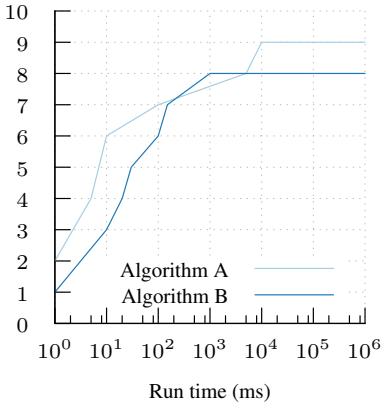
Figure 2.11(c) shows a less common type of figure: an *empirical cumulative distribution function plot* (henceforth *cumulative plot*). For a given run time t (on the horizontal axis), this answers the question "how many of the instances can be solved in at most t milliseconds per instance?" Equivalently, it answers the question "if the per-instance time limit were reduced to t, how many instances would each algorithm solve?" We can see, for example, that Algorithm A can solve six of the instances within 10 ms (the first six instances in the table), while Algorithm B can solve only three instances within that time.

Solver A	Solver B
0	0
1	1000
5	10
5	10
10	30
10	150
100	100
5000	20
10000	timeout
timeout	timeout

(a) Hypothetical run times per instance



(b) Scatter plot of run times



(c) Cumulative plot of run times

Figure 2.11: Hypothetical run times in ms for two solvers on ten instances: table, scatter plot, and cumulative plot. The time limit is 1000 seconds.

Chapter 3

Maximum Common Induced Subgraph: the McSplit Algorithm

3.1 Introduction

This chapter introduces the McSplit algorithm for the maximum common induced subgraph problem.[1] This algorithm, and the variants presented in this and the next chapter, use an adjacency-matrix representation of graphs. This representation makes the implementation of McSplit very simple, and is efficient for the small, dense graphs that are typical of MCIS benchmark instances. In Chapter 5, we will turn to a more intricate version using adjacency lists—first for induced subgraph isomorphism, then for MCIS.

The discussion of McSplit in this chapter proceeds as follows. Section 3.2 introduces the algorithm and its data structures. Section 3.3 proves the correctness of the algorithm and shows the close relationship between two methods for solving MCIS: McSplit on one hand and finding a maximum clique in the association graph on the other. Section 3.4 discusses variable and value ordering heuristics for McSplit. Sections 3.5 to 3.7 present experimental comparisons of McSplit with existing algorithms, and Section 3.8 explains the difference in run times that we observe in these experiments. Section 3.9 reviews papers that build on the McSplit algorithm, and Section 3.10 concludes.

3.2 The McSplit Algorithm

We initially assume that graphs are unlabelled, undirected and without loops; Section 3.2.2 will describe how these restrictions may be relaxed. Throughout, G and H will be the two

[1] The algorithm's name alludes to the maximum common subgraph (MCS) problem and to the partitioning step that will be described in detail in Section 3.2.

input graphs, and n_G and n_H will be their orders (that is, the sizes of their vertex sets).

McSplit finds a maximum-cardinality mapping $M^* = \{(v_1, w_1), \ldots, (v_m, w_m)\}$ with $|M^*| = m$ vertex pairs. In this mapping, the v_i are distinct members of $\mathrm{V}(G)$ and the w_i are distinct members of $\mathrm{V}(H)$; vertices 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 induced subgraph.

The behaviour of McSplit is similar to that of a CP solver, and the (v_i, w_i) pairs in M^* are analogous to assignments of value w_i to variable v_i in CP. A key difference between McSplit and a CP solver is that domains are not stored individually in McSplit; rather, sets of vertices from both graphs are stored together in "label classes". The remainder of this section explains the algorithm in detail.

Walkthrough Before giving full details of the algorithm, we illustrate its main concepts using the graphs G and H in Figure 3.1. These graphs have a maximum common subgraph with four vertices; one example is the mapping $\{(1, a), (2, f), (3, d), (5, b)\}$, which we abbreviate as $\{1a, 2f, 3d, 5b\}$, where vertex 1 is assigned to vertex a, 2 to f, and so on.

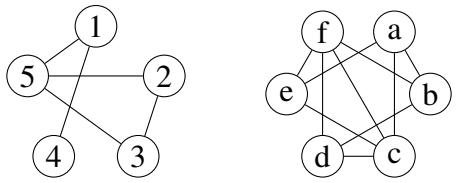


Figure 3.1: Example graphs G and H.

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 $\mathrm{V}(G)$ unmatched at each level of the search tree. Beginning at the root of the search tree, we select a vertex in $\mathrm{V}(G)$ as the first vertex to be mapped; in our example we will arbitrarily choose vertex 1. Each of the vertices in $\mathrm{V}(H)$ to which vertex 1 may be mapped will be tried in turn, and finally the decision to leave vertex 1 unmapped will be tried.

We begin by mapping vertex 1 to vertex a, giving $M = \{1a\}$. Now label each unmapped vertex in $\mathrm{V}(G)$ according to whether it is adjacent to vertex 1, and label each unmapped vertex in $\mathrm{V}(H)$ according to whether it is adjacent to vertex a, as shown in Figure 3.2(a). Vertices adjacent to 1 in G or to a in H have label 1; non-adjacent vertices have label 0. We can extend M with a mapping vw, with $v \in \mathrm{V}(G)$ and $w \in \mathrm{V}(H)$, 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.

Mapping $\{1a\}$

Labelling of G

Vertex	Label
2	0
3	0
4	1
5	1

Labelling of H

Vertex	Label
b	1
c	1
d	0
e	1
f	0

(a) After mapping 1 to a

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

Labelling of G

Vertex	Label
3	01
4	10
5	11

Labelling of H

Vertex	Label
b	11
c	11
e	10
f	01

(b) After mapping 2 to d

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

Labelling of G

Vertex	Label
4	100
5	111

Labelling of H

Vertex	Label
b	111
c	111
e	101

(c) After mapping 3 to f

Figure 3.2: Mapping M and vertex labels during search on example graphs G and H from Figure 3.1. 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 H (a and f) but not adjacent to the second mapped vertex (d).

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 3.2(b)). Each unmapped vertex $v \in \mathrm{V}(G)$ is now labelled with a two-character bit string, indicating its adjacency to each of the two mapped vertices in $\mathrm{V}(G)$—vertices 1 and 2. For example, vertex 3 is labelled 01, indicating that it is adjacent not to vertex 1 but to vertex 2. Labels are similarly given to unmapped vertices in $\mathrm{V}(H)$, showing adjacency to the mapped vertices a and d. Our invariant is maintained: we can extend M by a pair of vertices vw if and only if v and w 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 whose inputs are M and the current labelling. To see how this bound is calculated, consider the situation one level deeper in the search tree shown

in Figure 3.2(c). Here, three vertex labels are used: 100, 101, and 111. The first two of these labels 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 the size of a mapping is $|M| + 1 = 4$. The general formula for this upper bound is

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

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

Search tree Figure 3.3 shows the full search tree explored by McSplit with example graphs G and H as input. At the root node, no vertex-vertex assignments have yet been made. The large number at each node of the tree shows the computed bound. Nodes at which the incumbent is updated are circled. Each edge of the tree is labelled with the assignment added to the mapping, except those edges with red labels, which represent a decision not to map a vertex of G. The mapping at each search node contains all vertex-vertex assignments on the path from the root to that node; thus, for example, at the bottom-left node we have $M = \{1a, 2d, 3f, 5g\}$.

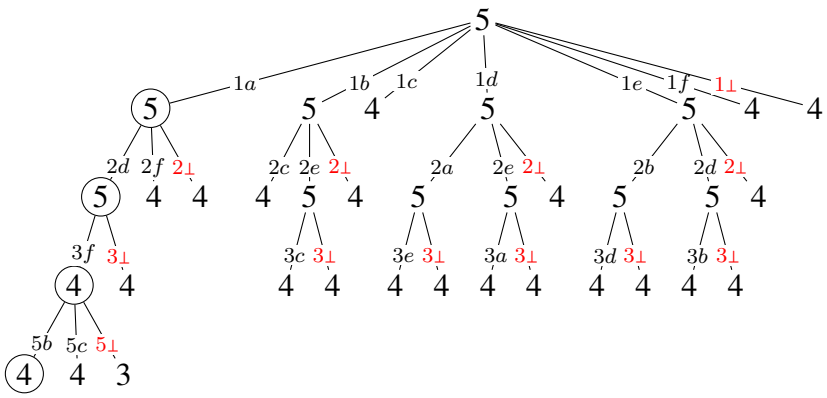


Figure 3.3: The search tree of McSplit on example graphs G and H

Label classes We require only $\mathrm{O}(n_G + n_H)$ space per level of the search tree to store labelling information. This is achieved by storing, for each label l that is used, a *label class*: a pair $\langle S_G, S_H \rangle$, where S_G is the set of vertices in $\mathrm{V}(G)$ labelled l, and S_H is the set of vertices in $\mathrm{V}(H)$ labelled l. Since there are $n_G + n_H$ vertices in the two graphs, at most $n_G + n_H$ label classes can exist at once, and there are at most $n_G + n_H$ vertices in the union of all of the S_G and S_H sets. Furthermore, we do not actually need to store the bits making up a label, since we care only that like-labelled vertices are kept together. Nor do we need to store any label class which is present 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 $\mathrm{V}(G)$ in which like-labelled vertices appear contiguously. The second array, similarly, stores a permutation of $\mathrm{V}(H)$ with like-labelled vertices together. We call these the *G-array* and *H-array* respectively. They are modified in-place and never copied.

The third array, which we call the *LC-array*, contains a record for each label class $\langle S_G, S_H \rangle$. Each record in an LC-array contains start and end pointers to the portion of the G-array that contains S_G, and start and end pointers to the portion of the H-array that contains S_H. This representation has similarities to data structures used in partition backtracking for graph isomorphism (McKay and Piperno, 2014; López-Presa and Anta, 2009). A new LC-array is created for each level of the search tree, and the LC-arrays for ancestor nodes of the search tree are kept in memory for use after backtracking.

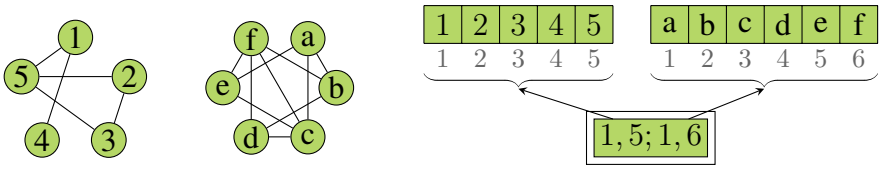
Figure 3.4 illustrates this data structure using our running example. The first subfigure shows the initial label classes when no vertex assignments have been made; the second and third subfigures correspond to the first two steps in Figure 3.2. Each subfigure shows graphs G and H with vertices colour-coded according to their label class. The three arrays of our data structure are shown on the right of each subfigure, with the LC-array below the G-array and H-array. The grey numbers below the G-array and H-array are the array indices, which are not stored explicitly.

When the algorithm begins, all vertices in the two graphs are in a single label class, as shown in Figure 3.4(a). The LC-array has a single element, $1, 5; 1, 6$; this indicates that the label class contains the vertices in positions 1 to 5 of the G-array and positions 1 to 6 of the H-array.

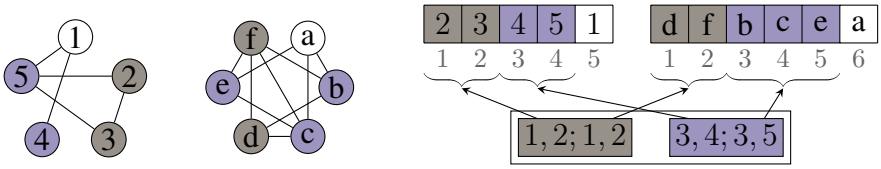
The label classes after mapping vertex 1 to vertex a are shown in Figure 3.4(b). The grey label class shows vertices that are not adjacent to vertex 1 or vertex a; the purple label class shows adjacent vertices. The LC-array contains an element for each of these label classes, with pointers into the G-array and H-array.

The situation after the additional mapping of 2 to d is shown in Figure 3.4(c). There are now three label classes: the mapped vertices 2 and d have been moved out of their label classes, and the purple label class of Figure 3.4(b) has been partitioned. The LC-array has three members, corresponding to the three label classes.

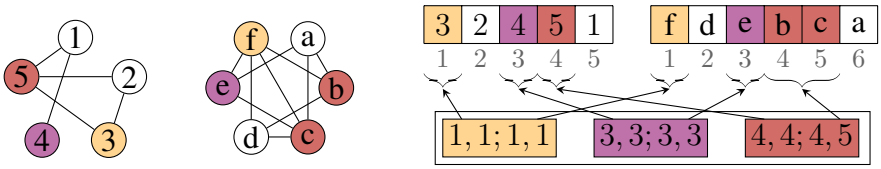
The McSplit algorithm in detail We start our search at the function `McSplit` (line 24 of Algorithm 1), with graphs G and H as inputs. This function returns a mapping of maximum cardinality. In line 27 the initial call is made to the recursive function `Search`; at this point the mapping M is empty, and we have a single label class containing all vertices.



(a) Before mapping any vertices



(b) After mapping 1 to a



(c) After mapping 2 to d

Figure 3.4: An illustration of McSplit's data structure for representing label classes. In each subfigure, graphs G and H are shown on the left and our data structure for representing label classes is shown on the right. Vertices in the same label class are shown in the same colour; white vertices do not belong to any label class but may belong to the mapping M, which is stored separately.

The `Search` function has two parameters. The parameter *future* is the list of label classes, each represented as a $\langle S_G, S_H \rangle$ 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 to 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 $\langle S_G, S_H \rangle$ is selected from *future* using a heuristic that we will describe in Section 3.4 (line 6); from this label class, a vertex v is selected from S_G (line 7). We iterate over all vertices w in S_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 is 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 S_G adjacent to v and vertices in S_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 S_H we now consider what happens if we make the decision not to map v (lines 20 to 23).

Algorithm 1: McSplit: a branch-and-bound algorithm to find a maximum common induced subgraph of two graphs.

```
1  Search(future, M)
2  begin
3      if |M| > |incumbent| then incumbent ← M
4      bound ← |M| + Σ_{⟨S_G,S_H⟩∈future} min(|S_G|, |S_H|)
5      if bound ≤ |incumbent| then return
6      ⟨S_G, S_H⟩ ← SelectLabelClass(future)
7      v ← SelectVertex(S_G)
8      for w ∈ S_H do
9          future' ← ∅
10         for ⟨S'_G, S'_H⟩ ∈ future do
11             S''_G ← S'_G ∩ N_G(v)
12             S''_H ← S'_H ∩ N_H(w)
13             if S''_G ≠ ∅ and S''_H ≠ ∅ then
14                 future' ← future' ∪ {⟨S''_G, S''_H⟩}
15             S''_G ← S'_G ∩ N̄_G(v)
16             S''_H ← S'_H ∩ N̄_H(w)
17             if S''_G ≠ ∅ and S''_H ≠ ∅ then
18                 future' ← future' ∪ {⟨S''_G, S''_H⟩}
19         Search(future', M ∪ {(v, w)})
20     S'_G ← S_G ∖ {v}
21     future ← future ∖ {⟨S_G, S_H⟩}
22     if S'_G ≠ ∅ then future ← future ∪ {⟨S'_G, S_H⟩}
23     Search(future, M)

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

Updating the data structure To describe our implementation of lines 10 to 18 of the algorithm, we now return to the data structure illustrated in Figure 3.4. We use a simple $\mathrm{O}\left(n_G + n_H\right)$-time procedure to create *future'*. First, v is removed from its label class by swapping it with the rightmost element of its label class in the G-array and decrementing the corresponding pointer in the LC-array. Vertex w is removed in a similar manner. Next, the following procedure, corresponding to lines 11 to 18, is carried out for each label class $\langle S_G, S_H \rangle$. The sub-array of the G-array corresponding to S_G is partitioned such that all vertices adjacent to v are moved to the rightmost part of the sub-array. Similarly, the sub-array of the H-array corresponding to S_H is partitioned in order to move all vertices adjacent to w to the right. If, after this partitioning procedure, the right-hand portion of both of these subarrays has nonzero length, we create a new LC-array element pointing to these right-

hand portions; this step corresponds to lines 13 to 14. If the left-hand portion of both of these subarrays has nonzero length, we create a new LC-array element pointing to these left-hand portions; this step corresponds to lines 17 to 18.

Although this partitioning step re-orders vertices within a label class to create new label classes, a convenient property is that the vertices of the larger label class that existed prior to partitioning remain contiguous. Therefore there is no need to copy the G-array or H-array, or to store additional bookkeeping information to restore label classes when backtracking; we can simply revert to the LC-array from the parent node in the search tree.

3.2.1 A Related Algorithm: Schmidt and Druffel (1976)

The algorithm of Schmidt and Druffel (1976), henceforth SD, is a backtracking algorithm for a different problem—graph isomorphism—that uses a method of partitioning similar to that of McSplit. Whereas McSplit stores the vertices of a label class contiguously, SD stores an array containing one integer for each vertex of G, and a similar array for H. This simple data structure corresponds almost directly to the tables of labels used in Figure 3.2 to explain McSplit, although SD uses an additional normalisation step at each search node to ensure that each label can fit in a machine word. Like McSplit, SD has an $O(n_G + n_H)$ partitioning step. Thus the data structures of SD could plausibly be used in McSplit with little change in speed, and this would be an interesting avenue for future work. However, the present author cannot see a way to adapt the SD data structures to handle sparse graphs efficiently as we do with McSplit in Chapter 5.

Beyond the difference in data structures, there are two other major differences between SD and McSplit. Because SD solves the graph isomorphism problem, it is unnecessary to calculate an upper bound; the algorithm can backtrack as soon as the number of vertices of G with a given label differs from the number of vertices of H with that label. A further difference from McSplit is that SD uses the distance matrix (a matrix of shortest path lengths) for each graph to give a good initial partition and to refine the partition more at each search node than McSplit's method based on adjacency matrices. Unfortunately, the distance matrix cannot be straightforwardly used for the maximum common subgraph problem.

3.2.2 Extensions for Problem Variants

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 Suppose the vertices in the input graphs have labels (as distinct from the bit-string labels described in Section 3.2 which were used to explain the algorithm but are not a property of the input graphs). We assume without loss of generality that these labels are natural numbers. Suppose further that we require that each vertex of G that appears in the common subgraph be mapped to a vertex of H with the same label.

Let $\ell_G : V(G) \to \mathbb{N}$ be the labelling function for graph G, and let $\ell_H : V(H) \to \mathbb{N}$ be the labelling function for graph H. Let $L \subset \mathbb{N}$ be the set of labels that are used in both graphs. To enforce the requirement of matching labels, we simply replace the initial label class $\{\langle V(G), V(H)\rangle\}$ in line 27 of Algorithm 1 with $\{\langle \ell_G^{-1}(l), \ell_H^{-1}(l)\rangle \mid l \in L\}$. Thus, we replace a single label class in which any vertex in G can be mapped to any vertex in H with a set of label classes ensuring that vertices can only be mapped to like-labelled vertices.

If some vertices have loops, we can use the same procedure by augmenting vertex labels with an extra bit with takes the value 1 if and only if the vertex has a loop. The vertex labels will then be members of the set $\mathbb{N} \times \{0, 1\}$.

Directed graphs without edge labels The next variant we consider has as inputs directed graphs with no 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 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. The earliest reference I have found to this data structure is in López-Presa and Anta (2009). 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 $t \to u$, 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 $\langle S'_G, S'_H\rangle$ 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 $L = \{0, 1, 2, 3\}$.

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

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

Undirected graphs with edge labels If the inputs are undirected graphs with edge labels, we create two-dimensional arrays A_G and A_H containing the 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 $n_G + n_H$ distinct labels, the loop in Algorithm 2 may execute up to $n_G + n_H$ times, resulting in $\mathrm{O}\left((n_G + n_H)^2\right)$ time complexity per search node. To achieve $\mathrm{O}\left((n_G + n_H)\log(n_G + n_H)\right)$ time complexity per search node—as we do in our implementation of McSplit—we 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 $\langle S'_G, S'_H \rangle$. Next, sort S'_G and S'_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 S'_G and S'_H from left to right, in a manner that resembles the merging step of merge sort.

Directed graphs with edge labels The case of *directed* graphs with edge labels 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 $u \to v$ (or 0 if no edge exists) and l_2 is the label on the reverse edge. Our implementation packs each pair of labels into a single machine word for efficiency.

Maximum common *connected* subgraph In chemistry applications, it is sometimes desirable to require that the common subgraph be connected (Ehrlich and Rarey, 2011). We consider only undirected graphs. We may modify McSplit to ensure connectedness by requiring that the mapped vertices of graph G, that is $\{v \mid (v, w) \in M\}$, induced a connected subgraph of G at every search node where M is non-empty. This simple strategy follow the branching scheme described by Vismara and Valery (2008). Returning to Figure 3.4, for example, it would not be permissible to map vertex 2 after making the initial mapping $(1, a)$, since vertices 1 and 2 are not adjacent in G. Instead, we would need to branch on vertex 4 or 5 of graph G. In McSplit, we can ensure connectedness by storing an extra Boolean flag in each label class object. This takes the value $true$ if and only if the vertices in the class are adjacent to at least one vertex in M. When choosing a vertex on which to branch, we then select only from label classes in which this flag is $true$.

We could, in addition, follow Ndiaye and Solnon (2011) by using a *connectedness constraint* to further filter domains during search. In McSplit this would work as follows: at each search node with at least one mapped vertex, perform a depth-first traversal of each graph visiting only those vertices that are mapped or that remain in some label class, then

remove unreached vertices from their label classes. This would be straightforward to implement and would work well with McSplit's data structures, but the current McSplit implementation does not yet include this constraint.

Finding all maximum common subgraphs With two trivial modifications, the McSplit algorithm can be used to find *all* maximum common subgraphs of a pair of graphs, rather than just a single example of a maximum common subgraph. First, we modify line 3 of Algorithm 1 so that a mapping M is stored if its size greater than or equal to — rather than strictly greater than — the size of the incumbent. Second, on line 5, we return from the function only if the bound is *strictly* less than the size of the incumbent.

McSplit↓ The final version that we introduce in this section does not solve a variant of the problem, but uses a different strategy that may be applied to any of the problem variants that we have described. Our inspiration is the $k\downarrow$ algorithm of Hoffmann et al. (2017).[2] Rather than using branch and bound to find increasingly large common subgraphs, $k\downarrow$ solves a sequence of decision problems: for $k = 0, 1, \ldots$, it tests whether G and H have a common induced subgraph of order $n_G - k$. The $k\downarrow$ algorithm is a modified subgraph isomorphism solver, and for low values of k it can perform stronger reasoning that McSplit, based on vertex degrees and the existence of paths between vertices, to eliminate candidate vertex-vertex mappings. This stronger reasoning is not compatible with McSplit's data structures, but our McSplit↓ variant uses the $k\downarrow$ strategy of solving a sequence of decision problems rather than using branch and bound. The modifications to the McSplit algorithm are straightforward: we call the main `McSplit` method once per goal size ($n_G, n_G - 1, n_G - 2, \ldots$), then 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.[3]

3.3 Proof of Correctness

This section proves the correctness of the variant of McSplit in Algorithm 3, which solves the decision variant of maximum common induced subgraph: given two graphs and a natural number t (the "target"), does there exist a common induced subgraph with at least t vertices?

[2]The $k\downarrow$ algorithm may be used either to find a maximum common induced subgraph or to a solve a problem closely related to the maximum common edge subgraph problem. Here, we consider only the former use of $k\downarrow$.

[3]Our McSplit implementation has an additional small optimisation: we store an incumbent, in the style of a branch and bound algorithm, even if that incumbent is smaller than the current goal size. For example, if the algorithm is searching for a common subgraph with 6 vertices and it finds a common subgraph with 5 vertices in the process, it records this. On a later iteration, with a goal size of 5, this stored subgraph can then be returned without any need for search. Thus, the subproblems are not solved completely independently, as they are by $k\downarrow$.

(The circled letters should be ignored for now.) This algorithm contains all of the ingredients of the full McSplit algorithm (Algorithm 1) other than the branch-and-bound technique; by removing branch and bound we can give a simpler proof that focuses on the details that are specific to McSplit. We will see at the end of the section how the proof can be extended to the full branch-and-bound McSplit algorithm.

Algorithm 3: A decision-problem variant of McSplit.

```
1  Search(future, M, t)
2  begin
3      if |M| = t then return true (A)
4      bound ← |M| + Σ_{⟨S_G,S_H⟩∈future} min(|S_G|, |S_H|)
5      if bound < t then return false (B)
6      ⟨S_G, S_H⟩ ← SelectLabelClass(future)
7      v ← SelectVertex(S_G)
8      for w ∈ S_H (C) do
9          future' ← ∅
10         for ⟨S'_G, S'_H⟩ ∈ future do
11             S''_G ← S'_G ∩ N_G(v)
12             S''_H ← S'_H ∩ N_H(w)
13             if S''_G ≠ ∅ and S''_H ≠ ∅ then
14                 future' ← future' ∪ {⟨S''_G, S''_H⟩}
15             S''_G ← S'_G ∩ N̄_G(v)
16             S''_H ← S'_H ∩ N̄_H(w)
17             if S''_G ≠ ∅ and S''_H ≠ ∅ then
18                 future' ← future' ∪ {⟨S''_G, S''_H⟩}
19         if Search(future', M ∪ {(v, w)}, t) then return true (D)
20     S'_G ← S_G ∖ {v}
21     future' ← future ∖ {⟨S_G, S_H⟩}
22     if S'_G ≠ ∅ then future' ← future' ∪ {⟨S'_G, S_H⟩}
23     if Search(future', M, t) then return true (E)
24     return false (F)

25 McSplitDecision(G, H, t)
26 begin
27     return Search({⟨V(G), V(H)⟩}, ∅, t)
```

For simplicity, we assume that the input graphs are unlabelled, undirected and loopless.

An algorithm for the clique decision problem Recall that a common subgraph with t vertices exists if and only if the association graph has a clique with t vertices; see Section 2.8.1. Our proof strategy is as follows: we will prove the correctness of a simple algorithm for the clique decision problem, then we will show that this algorithm, when applied to

the association graph, carries out exactly the same steps as the McSplit algorithm applied to graphs G and H.

The clique algorithm we will use, Algorithm 4, uses a similar upper-bounding strategy to MCSa1 (Prosser, 2012; Tomita et al., 2013; McCreesh et al., 2016a); both algorithms heuristically colour vertices and use the number of colours as an upper bound on the number of vertices that can be added to the growing clique. We will shortly describe the `ColouringUpperBound()` function called on line 6 of Algorithm 4; our proof requires the colouring to mimic McSplit's behaviour rather than to use the greedy colouring method of MCSa1.

Algorithm 4: A simple algorithm for the clique decision problem.

```
1  CliqueSearch(P, C, t)
2  Data: Candidate vertices P, clique C such that |C| ≤ t, and target size t
3  Result: true if and only if A has a clique of size t with C as a subset
4  begin
5      if |C| = t then return true (A)
6      bound ← |C| + ColouringUpperBound(P)
7      if bound < t then return false (B)
8      S ← a subset of P that is an independent set in A
9      for x ∈ S (C) do
10         if CliqueSearch(P ∩ N(x), C ∪ {x}, t) then return true (D)
11     if CliqueSearch(P ∖ S, C, t) then return true (E)
12     return false (F)

13 Clique(A, t)
14 Data: Graph A and target size t
15 Result: true if and only if a clique of size t exists in A
16 begin
17     return CliqueSearch(V(A), ∅, t)
```

The parameters of the `CliqueSearch` function of Algorithm 4 are vertex sets P and C and a natural number t. Set C—the current clique—is a subset of $V(A)$, and must be a clique of size no greater than t. Set P ("potential") is a set of candidate vertices for addition to the C; we require that $C \cap P = \varnothing$ and that each vertex in P is adjacent in A to every member of C. The parameter t is the target clique size. We now show the correctness of Algorithm 4.

Lemma 1. *The function* `CliqueSearch`(P, C, t) *returns* $true$ *if and only if* A *has a clique* K *of size* t *such that* $C \subseteq K \subseteq C \cup P$.

Proof. The proof is by induction on $|P|$.

Base case, $|P| = 0$. Since C is a clique by assumption and $P = \varnothing$, we must show that the function returns $true$ if and only if $|C| = t$. If $|C| = t$, line 5 returns $true$ as required. If $|C| < t$, then $bound = |C|$, since `ColouringUpperBound`(P) is zero. Therefore $false$ is returned on line 7.

Inductive case. Let natural number k be given, and assume that the lemma holds for $|P| < k$. We will show that the lemma also holds if $|P| = k$. Let C, P, and t be given, such that $|P| = k$.

For the first direction of the implication in the inductive case, suppose there is no clique K of size t such that $C \subseteq K \subseteq C \cup P$. Line 5 does not return $true$ since $|C| < t$. Now, either line 7 returns $false$ or we must show that neither line 10 nor line 11 returns $true$. First consider line 10. By the inductive assumption, for every clique K of size t we have either that $C \nsubseteq K$ or $K \nsubseteq C \cup P$. Since $C \subset C \cup \{x\}$ and $C \cup \{x\} \cup (P \cap N(x)) \subseteq C \cup P$, it follows for every clique K of size t that either $C \cup \{x\} \nsubseteq K$ or $K \nsubseteq C \cup \{x\} \cup (P \cap N(x))$. Hence the call to `CliqueSearch`() on line 10 returns $false$ for every value taken by x in the loop. It remains to show that line 11, if reached, never returns $true$; this is the case since $P \setminus S \subset P$ and therefore for every clique K of size t we have by the inductive assumption that either $C \nsubseteq K$ or $K \nsubseteq C \cup (P \setminus S)$.

For the second direction of the implication in the inductive case, suppose there exists a clique K of size t such that $C \subseteq K \subseteq C \cup P$. If $|C| = t$, line 5 returns $true$ as required. Otherwise, $bound \geq t$ since $K \setminus C$ is a clique and therefore each member of $K \setminus C$ must be in a different colour class in the colouring on line 6; therefore, line 7 does not return $false$. It remains to show that either line 10 or line 11 returns $true$. We consider two cases.

Case 1: $S \cap K \neq \varnothing$. Let x be the unique element of $S \cap K$. Since $C \subseteq K$ and $x \in K$, we have $C \cup \{x\} \subseteq K$. Since K is a clique containing x, we have $K \subseteq \{x\} \cup N(x)$ and therefore $P \cap K \subseteq \{x\} \cup (P \cap N(x))$. Combining this with the inductive assumption that $K \subseteq C \cup P$, we have $K \subseteq C \cup \{x\} \cup (P \cap N(x))$. The recursive call on line 10 thus returns $true$.

Case 2: $S \cap K = \varnothing$. Since $K \subseteq C \cup P$ by the inductive assumption, we have $K \subseteq C \cup (P \setminus S)$. Therefore line 11 returns $true$. □

Equivalence of Algorithm 3 and Algorithm 4. We will now show that Algorithm 3 and Algorithm 4 explore the same search tree. Suppose we have graphs G and H for which we wish to find a common subgraph, and let A be their association graph. We will consider calls to `Search`$(future, M, t)$ in the McSplit algorithm (Algorithm 3), and corresponding calls to `CliqueSearch`(P, C, t) in the clique algorithm (Algorithm 4), where $P = \{G \times H \mid \langle G, H \rangle \in future\}$ and $C = M$. That is, each call to the `CliqueSearch`() function takes as its argument P a set containing every node (v, w) in the association graph such that v

and w are in the same label class in *future*, and as its C argument every node (v, w) in the association graph corresponding to an assignment in M.

In order for the two algorithms to explore equivalent search trees, we must precisely specify the `ColouringUpperBound()` function and the procedure for selecting independent set S on line 8 of Algorithm 4. The `ColouringUpperBound`(P) call returns the size of a colouring of $P = \{G \times H \mid \langle G, H\rangle \in \mathit{future}\}$ in the association graph. For each $\langle G, H\rangle$ in *future*, we colour the corresponding subset of P, that is, $G \times H$, as follows. If $|G| \leq |H|$, we assign a colour to the set of association graph nodes $\{(v, w) \mid w \in H\}$ for each $v \in G$. If $|G| > |H|$, we assign a colour to the set of association graph nodes $\{(v, w) \mid v \in G\}$ for each $w \in H$. The subset of P corresponding to label class $\langle G, H\rangle$ therefore requires $\min(|S_G|, |S_H|)$ colours. The size of the colouring returned by `ColouringUpperBound`(P) is $\sum_{\langle S_G, S_H\rangle \in \mathit{future}} \min(|S_G|, |S_H|)$.

We now turn to line 8 of Algorithm 4, which chooses an independent set of nodes from P on which to branch. We choose $S = \{(v, w) \mid w \in H\}$, with H and v taking the values selected by `SelectLabelClass`(future) and `SelectVertex`(G) on lines line 6 and line 7 of Algorithm 3. This ensures that the branching decisions in the clique algorithm mirror those of the McSplit algorithm.

Theorem 1. *Let graphs G and H be given, and let A be their association graph. Let a set of label classes future, a mapping M, and a target size t be given. Let P be the set of association-graph nodes corresponding to future; that is, $P = \{G \times H \mid \langle G, H\rangle \in \mathit{future}\}$. Let $C = M$. The function* `CliqueSearch`(P, C, t) *returns the same value as* `Search`(future, M, t).

Proof. The proof is by induction on $|P|$.

Base case, $|P| = 0$. If $|M| = t$, both algorithms return *true* at the line marked A. Otherwise, $\mathit{bound} = |M|$ and both algorithms return *false* at the line marked B.

Inductive case. Both algorithms return *true* at the line marked A if and only if $|M| = t$. We have shown that `ColouringUpperBound`$(P) = \sum_{\langle S_G, S_H\rangle \in \mathit{future}} \min(|S_G|, |S_H|)$; therefore, both algorithms return *false* at the line marked B if and only if

$$|M| + \sum_{\langle S_G, S_H\rangle \in \mathit{future}} \min(|S_G|, |S_H|) < t.$$

Due to the way we have defined set S, the loop marked C in the clique algorithm iterates over the set $\{(v, w) \mid w \in H\}$, where H is the set iterated over in the corresponding loop of the McSplit algorithm. To show that the lines marked D are equivalent in the two algorithms, we must show that $P \cap N_A((v, w)) = \{G \times H \mid \langle G, H\rangle \in \mathit{future}'\}$ (where (v, w)

is the value of loop variable x in the clique algorithm), and that $C \cup \{(v,w)\} = M \cup \{(v,w)\}$. The latter equality is trivial.

To prove one direction of inclusion in the former equality, let (v',w') be an element of $P \cap N_A((v,w))$. Since, by assumption, $P = \{G \times H \mid \langle G,H\rangle \in \mathit{future}\}$, there exists a label class $\langle G',H'\rangle \in \mathit{future}$ such that $v \in G'$ and $w \in H'$. Since $(v',w') \in N_A((v,w))$, we have by the definition of the association graph that $v \neq v'$, $w \neq w'$, and either (1) v and v' are adjacent and w and w' are adjacent or (2) v and v' are non-adjacent and w and w' are non-adjacent. In the first of these cases, a label class containing v' and w' will be added to future' on line 14. In the second case, a label class containing v' and w' will be added to future' on line 18. In either case, (v',w') will be an element of $\{G \times H \mid \langle G,H\rangle \in \mathit{future}'\}$, as required.

To prove the second direction of inclusion, let v' and w' be such that for some element $\langle G'',H''\rangle$ of future', we have $v' \in G''$ and $w' \in H''$. It must hold that $\langle G'',H''\rangle$ was added to future' either on line 14 or line 18 of Algorithm 3. In the former case, by the construction of G'' and H'' in Algorithm 3, we have that v and v' are distinct and adjacent in G and that w are w' are distinct and adjacent in H. Therefore, (v',w') is an element of $N_A((v,w))$. Moreover, again by the construction of G'' and H'', there is some $\langle G',H'\rangle \in \mathit{future}$ such that $v' \in G'$ and $w' \in H'$, and therefore (v',w') is an element of P. Thus, (v',w') is an element of $P \cap N_A((v,w))$ as required. A similar proof applies in the case where $\langle G'',H''\rangle$ was added to future' on line 18 of Algorithm 3.

We now move to the lines marked E of the two algorithms. To prove the equivalence of the two recursive calls, we must show that $P \setminus S = \{G \times H \mid \langle G,H\rangle \in \mathit{future}'\}$. We have

$$P \setminus S = \tag{3.1}$$
$$\{(u,w) \in \{G \times H \mid \langle G,H\rangle \in \mathit{future}\} \mid u \neq v\} = \tag{3.2}$$
$$\{\{u \in G \mid u \neq v\} \times H \mid \langle G,H\rangle \in \mathit{future}\} = \tag{3.3}$$
$$\{G \times H \mid \langle G,H\rangle \in \mathit{future}'\}. \tag{3.4}$$

Finally, both function return false upon reaching the line marked F. □

It is clear from the proof of Theorem 1 not only that Algorithm 3 and Algorithm 4 give the same answer, but that they explore identically shaped search trees.

Branch and bound algorithms To simplify the exposition, this section has shown that a decision-problem variant of McSplit is equivalent to an algorithm for the clique decision problem. It would be straightforward to convert our clique algorithm into a branch and bound algorithm by introducing a global $\mathit{incumbent}$ variable in the style of Algorithm 1. The proof

of Theorem 1 could then be modified trivially to show the equivalence of the full McSplit algorithm to a branch-and-bound algorithm for the maximum clique problem.

Extensions of McSplit The definition of the association graph can easily be modified to handle loops, directed edges, and labelled vertices and edges (Levi, 1973). It would be possible to modify our correctness proof of McSplit to apply to any of these variants of the problem by using an appropriate definition of the association graph.

To solve the maximum common *connected* subgraph problem with a clique algorithm, we must distinguish between two types of edges in the association graph: c-edges corresponding to edges in graphs G and H, and d-edges corresponding to non-edges in the two graphs. We then search for a clique in the association graph that is spanned by c-edges (Koch, 2001; Vismara and Valery, 2008). McCreesh et al. (2016a) introduces a modified maximum-clique algorithm that solves this problem by only choosing to add those vertices to the clique C that share at least one c-edge with a vertex that is already in the clique. (The first vertex added to C is a special case and is not subject to this restriction.) It would be straightforward to modify our clique algorithm to solve this version of the problem, and we could then modify the proof of theorem 1 to prove the correctness of the connected version of McSplit.

3.4 Variable and Value Ordering Heuristics

We turn now to variable- and value-ordering heuristics for McSplit. The variable-ordering heuristic begins with the `SelectLabelClass` function; our implementation chooses a label class with the smallest $\max(|S_G|, |S_H|)$. As we will see in Section 3.6, this strategy performs slightly better than a smallest-domain-first strategy (select the label class with smallest $|S_H|$) on a set of benchmark instances.[4] Ties are broken by selecting a label class with the smallest-numbered vertex of graph G. The `SelectVertex` function selects the smallest-numbered vertex within this label class. (The next subsection will discuss strategies for assigning numbers to vertices.)

The value-ordering heuristic controls the iteration order at line 8 of Algorithm 1. The strategy in our implementation iterates in ascending order of vertex number.

[4]Patrick Prosser carried out further, unpublished experiments that found that minimising $|S_G| \times |S_H|$ performs similarly (Prosser, 2017). The fact that we can cheaply compute such heuristics is an advantage of McSplit over CP algorithms, although the reason for the success of these heuristics remains unclear.

3.4.1 Ordering Graphs by Vertex Degree

Our variable and value ordering strategies both depend on the order in which vertices are numbered. In this dissertation, we follow the Glasgow Subgraph Solver (McCreesh et al., 2018, 2020) and number vertices in order of degree. (For efficiency, we also generate new adjacency matrices for the reordered graphs before search.)

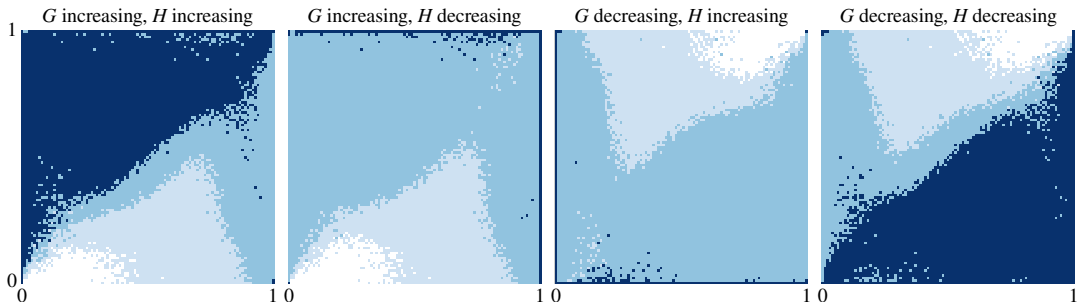
Is it better to order the vertices in each graph in ascending or descending order of degree? To shed light on this question, we present results of an experiment comparing increasing and decreasing degree orders for graphs G and H where both graphs are random Erdős-Rényi $G(n, p)$ graphs. As we will see, the answer depends in a complex way on the structure of the two graphs.

In the experiment, the p (density) parameter for both graphs was varied from 0 to 1 in steps of 0.01, for a total of 10201 pairs of graph. For each graph, we considered the two possible degree orders; we call these strategies "G increasing" (smallest degree first for graph G), "G decreasing", (largest degree first for graph G), "H increasing", and "H decreasing". Due to the large number of instances, $n = 10$ was used. Each run was repeated 32 times with different random graphs, and an average (mean) number of calls to the `Search` function was computed across the 32 runs.

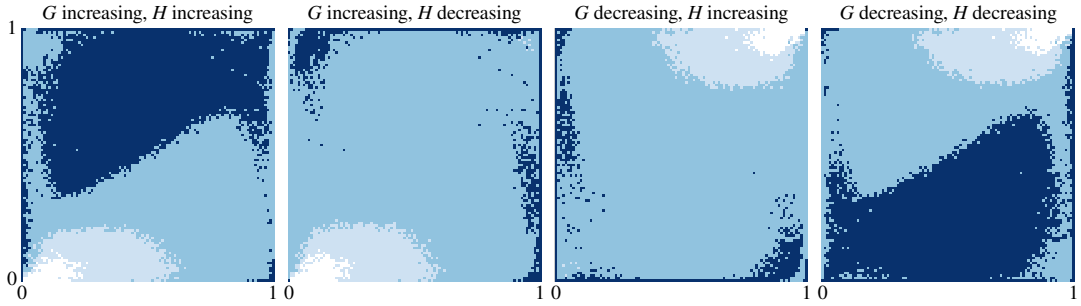
The heatmaps in Figure 3.5 show how effective each pairing of ordering strategies was for each G density / H density pair.[5] To explain the plots, we refer to the first of the four squares in Figure 3.5(a). On the x and y axes, we have values of the graph generator's p (density) parameter ranging from 0 to 1 for graphs G and H respectively. The plot shows, for each pair of density parameters, how the strategy "G increasing, H increasing" compares with the best of the four possible strategies. Dark blue indicates that "G increasing, H increasing" is not beaten by any of the other three strategies, mid blue indicates that it requires between 1 and 2 times the search effort of the best strategy, and light blue indicates that it requires between 2 and 10 times the search effort of the best strategy.

Figure 3.5(a) shows results for unlabelled, undirected graphs. The middle two of the four subplots on this row show that—at least for these random graphs—it is seldom optimal to order one graph by increasing degree and the other by decreasing degree. The first and last subplots show that it is best to sort both graphs in increasing degree order if the density of H is much greater than the density of G, while it is best to sort in decreasing degree order if the opposite is true. Yet the decision rule "sort in increasing order if and only if the density of H is greater than the density of G" often leads to poor results if the graphs are sparse and have similar density, as we can see in the large light-coloured region at the bottom left of the first plot.

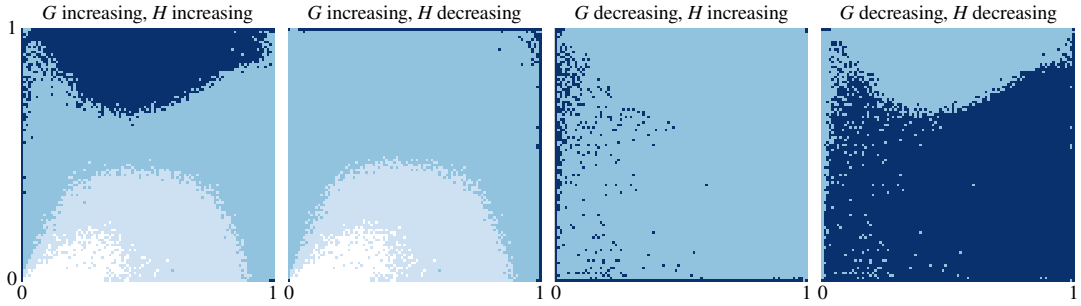
[5]The experiments in this section and Section 5.4 were inspired by McCreesh et al. (2018), which uses similar heatmaps to explore the phase transition and heuristics for subgraph isomorphism problems.



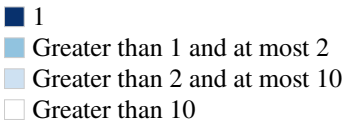
(a) Unlabelled, undirected



(b) Unlabelled, directed



(c) Edge labelled, undirected



(d) Colour key (search effort divided by search effort of best strategy)

Figure 3.5: Heatmaps of McSplit search effort compared to the best strategy, varying density of G (horizontal axis) and density of H (vertical axis)

Figure 3.5(b) shows results for directed graphs. The overall pattern is somewhat similar to the undirected case. Finally, Figure 3.5(c) shows results for undirected graphs with edge labels chosen uniformly at random from a set of five labels. In this case, the increasing order for both graphs tends to be best when H is very dense, and the decreasing order tends to be best otherwise.

As these figures show, the optimal choice of variable and value ordering heuristic depends on the characteristics of an instance in a complex way. If we were to pick a simple

decision rule for ordering based on the densities of G and H, a plausible candidate would be "use increasing order for both graphs if the density of H is greater than $\frac{1}{2}$; otherwise use decreasing order for both graphs". This is the rule we use throughout this chapter and the next chapter, with one exception: in the comparison of algorithms in Section 3.7, we use the results from a large-scale experiment in McCreesh et al. (2017b), which used a simpler approach—decreasing degree order on all instances. This makes no difference for the directed-graph instances, since all graphs in the benchmark set have density less than $\frac{1}{2}$ and therefore the decreasing degree order is always consistent with our proposed rule. For the undirected instances, the two rules agree for 96% of the instances.

3.5 Instances Used in Experiments

Our experiments use a database of randomly generated maximum common subgraph instances (Santo et al., 2003; Conte et al., 2007). Each instance satisfies $10 \leq n_G = n_H \leq 100$. The database's authors produced each instance by generating a graph G' of a given family to serve as the common induced subgraph, then adding additional vertices and edges at random to G' in order to produce graphs G and H. Several families of common subgraph are used: Erdős-Rényi $G(n, p)$ graphs; 2D, 3D, and 4D meshes; bounded valence graphs; and randomly perturbed ("irregular") versions of meshes and bounded valence graphs.

For unlabelled instances, the first ten instances from each family whose members have no more than 50 vertices are used, giving a total of 4,110 instances. For labelled instances, the first ten instances from every family are used, giving a total of 8,140 instances. Like McCreesh et al. (2016a), we use the labelling scheme in which the number of distinct vertex labels and the number of distinct edge labels is approximately equal to a third of the number of vertices in each graph.

3.6 Experimental Evaluation 1: Finding All Maximum Common Subgraphs

Krissinel and Henrick (2004) introduce a forward-checking algorithm, which we will refer to as KH, for finding all maximum common induced subgraphs of two given graphs. KH is a forward-checking constraint programming algorithm, with a simple upper bound that is computed by counting the number of unmapped vertices in G that have at least one vertex remaining in their domain. In this section we introduce a sequence of modifications to McSplit that make the algorithm progressively closer to KH, and show experimentally how these modifications progressively reduce the speed of McSplit until it is similar to that of

KH. (This section compares only with KH for two reasons. First, KH differs from most other MCIS algorithms in that it finds all maximum common subgraphs rather than a single one. Second, KH serves as a baseline forward-checking algorithm against which we can test the usefulness of McSplit's data structures and heuristics.)

We can emulate the search tree of KH by making the following three modifications to McSplit in addition to the modifications for finding all maximum common subgraphs described in Section 3.2.2.

1. Modify the variable selection heuristic so that it chooses a label class with as small a set S_H as possible—a smallest domain first strategy—rather than minimising $\max(|S_G|, |S_H|)$.

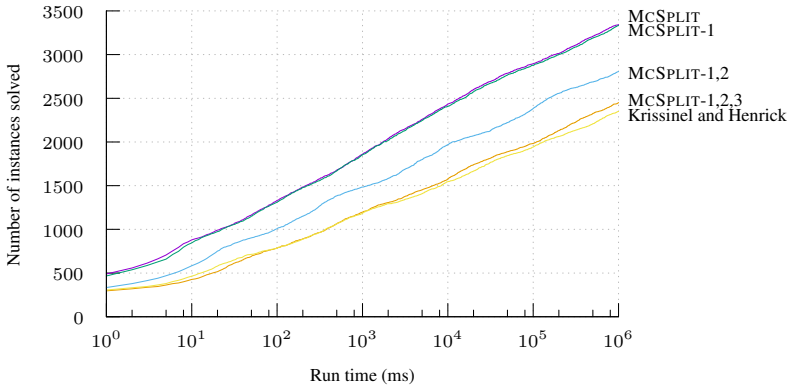
2. Rather than using $|M| + \sum_{\langle S_G, S_H \rangle \in future} \min(|S_G|, |S_H|)$ as the upper bound on line 4 of Algorithm 1, use $|M| + \sum_{\langle S_G, S_H \rangle \in future} |S_G|$. Clearly, this bound is greater than or equal to McSplit's bound.

3. Leave the vertices of G and H in their original order rather than sorting by degree.

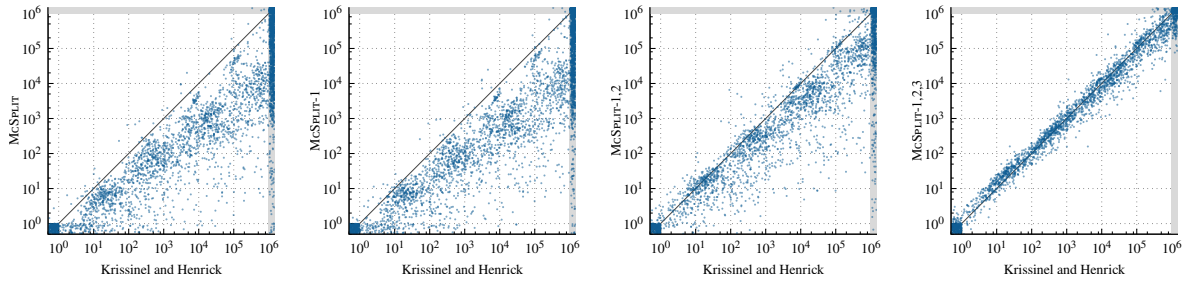
We hypothesise that each of these changes will make the McSplit algorithm slower, and that by making all three changes to McSplit we will have an algorithm with similar run times to the algorithm of KH (albeit with some variation due to the different data structures used by the two algorithms to represent domains). To test these hypotheses, we ran five algorithms on the set of 4110 unlabelled, undirected instances. In addition to McSplit and KH, we used three modified versions of McSplit: McSplit-1 has only the first change from the list above, McSplit-1,2 has the first two changes, and McSplit-1,2,3 (which we expect to be the slowest) has all three changes.

Figure 3.6 shows the results of our experiment. The cumulative plot shows, in line with our hypotheses, that McSplit is around two orders of magnitude faster than KH overall, and that McSplit-1,2,3 and KH have very similar overall performance. The first scatter plot shows that McSplit clearly outperforms KH, with KH running faster than McSplit on only a handful of instances. Moving from left to right across the four scatter plots we see a steady convergence between the speed of the McSplit variants and KH; the fourth scatter plot shows—as we expected—that McSplit-1,2,3 and KH have similar run times on an instance-by-instance basis. (It appears, furthermore, that KH has a slight advantage on the easiest instances and McSplit-1,2,3 has a slight advantage on the hardest instances.)

Returning to the cumulative plot, it is evident that the second and third changes (weakening the bound and not sorting by degree) have a large effect on the run time of McSplit but that the first change (using the sizes of both S_G and S_H in the variable-ordering heuristic) has only a minor effect.



(a) Cumulative plot of run times



(b) Per-instance run times, Krissinel and Henrick (KH) vs McSplit versions

Figure 3.6: Cumulative and scatter plots of run times (ms) of (Krissinel and Henrick, 2004) and four versions of McSplit on the 4110 unlabelled, undirected instances

3.7 Experimental Evaluation 2: Finding a Maximum Common Subgraph

This section presents our main performance comparison of McSplit against existing state of the art MCIS algorithms. A cluster of machines with dual Intel Xeon E5-2640 v2 CPUs and 64GBytes RAM was used for the evaluation; all code was compiled using GCC 5.3.0.[6] The experiments described in this section were first published in McCreesh et al. (2017b). Although I designed and implemented McSplit and was the main author of this paper, credit for running the experiments in this section is due to Ciaran McCreesh.

We compare against the following implementations.

- The best constraint programming implementations of Ndiaye and Solnon (2011) and McCreesh et al. (2016a): FC+Bound for unlabelled graphs, and MAC+Bound for labelled graphs. We refer to these as CP-FC and CP-MAC respectively. On the adjacency constraints, CP-FC uses forward checking while CP-MAC maintains arc consistency. Both algorithms use the soft allDifferent constraint to calculate an upper bound

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

but not to remove values from domains. Both branching and filtering are used for the connected variant of the problem.

- The clique encodings and solver of McCreesh et al. (2016a). This uses Prosser's MCSa1 algorithm (Prosser, 2012), which is based on an algorithm by Tomita et al. (2013) and uses a bitset representation to enable very fast operations on sets (San Segundo et al., 2011).[7]
- The $k\downarrow$ algorithm of Hoffmann et al. (2017) (which only supports unlabelled, undirected, unconnected instances).

Each program is an optimised, dedicated implementation and does not use a general-purpose CP toolkit. The original authors' code was used in each case.

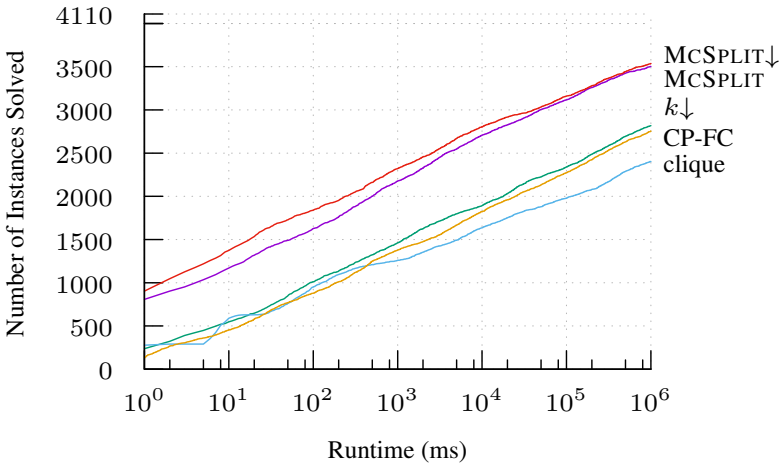
Unlabelled, undirected Figure 3.7(a) shows a plot of cumulative number of instances solved against runtime for unlabelled, undirected instances. We may compare the speed of two algorithms using the horizontal distance between their curves. For example, we could solve 2,000 of the 4,110 unlabelled undirected instances using the MCSPLIT algorithm if a time limit of 0.5 seconds per instance were imposed. 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 and MCSPLIT↓ are more than an order of magnitude faster than each of the non-MCSPLIT algorithms.

MCSPLIT↓ slightly outperforms MCSPLIT overall. But the similarity of the cumulative curves masks an asymmetric relationship between the run times of the two algorithms on an instance-by-instance basis, which is shown in Figure 3.7(b). On most instances, MCSPLIT is in fact slightly *faster* (but never much faster[8]) than MCSPLIT↓. Among the relatively small number of instances where MCSPLIT↓ is the faster of the two algorithms, however, it is often dramatically faster. The instances that favour MCSPLIT↓ tend to be ones where the solution is almost as large as the input graphs and thus few decision problems need to be solved; these are the dark red points that appear predominantly towards the bottom of the figure.

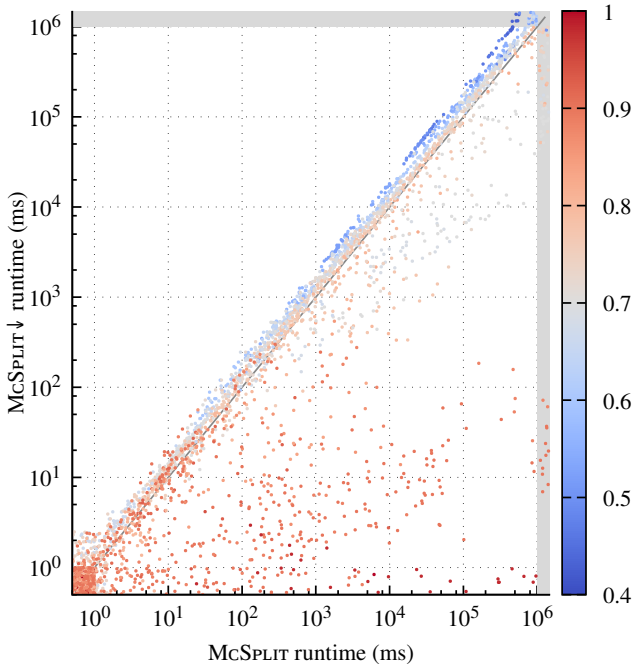
Why is MCSPLIT much slower than MCSPLIT↓ on many instances? Presumably the reason is that before the incumbent has reached the optimal value, the search may become "trapped" in areas of the search space with no good solutions, and the algorithm is not able

[7] After this experiment was run, the present author also tried using another state of the art maximum clique solver, IncMC2 (Li et al., 2018), which uses a technique from MaxSat solvers to achieve an upper bound that is tighter than the bound given by greedy colouring. This performed worse than MCSa1, and I therefore do not report its results.

[8] An additional unpublished experiment by the author shows that most of the run time of MCSPLIT↓ is spent on the last two decision problems. Since each individual run on a decision problem requires fewer search nodes than a full branch-and-bound search, it is unsurprising that the run time of MCSPLIT↓ is seldom more than twice that of MCSPLIT.



(a) Cumulative number of instances solved



(b) McSplit versus McSplit↓, with one plotted point per instance. The colour of each point shows the order (vertex count) of a maximum common subgraph as a proportion of the order of input graphs G and H.

Figure 3.7: Runtimes for maximum common induced subgraph (with no requirement for the common subgraph to be connected): unlabelled, undirected instances.

to prune the search tree effectively on line 5 of Algorithm 1 because of the small size of the incumbent.

Although the k↓ algorithm was slightly faster than CP-FC, it was slower than McSplit↓ on all of the instances that were solved within the time limit by k↓, and was more than an order of magnitude slower than McSplit↓ on most instances.

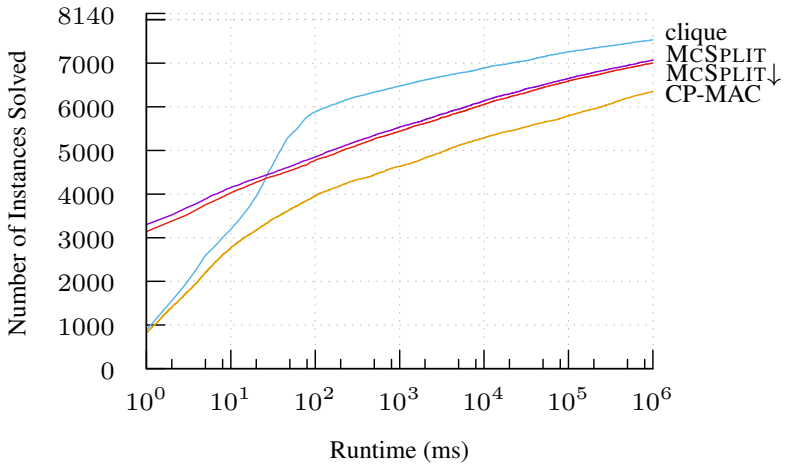
Vertex and edge labels, directed Cumulative runtimes for labelled, directed instances are shown in Figure 3.8(a). Again, McSplit is over an order of magnitude faster than the best existing CP algorithm, which is CP-MAC in this case. Yet the overall, clear winner (except in the very easy region of instances that can be solved in well under 100 ms) is the clique encoding. This is consistent with McCreesh et al. (2016a), who found that the clique encoding was by far the strongest solver of those tested for these instances.

Comparing only McSplit with McSplit↓ on these instances, the branch and bound algorithm is by a narrow margin the overall winner. The instance-by-instance comparison of the two algorithms in Figure 3.8(b) shows a similar pattern to the undirected, unlabelled instances, albeit without instances where McSplit↓ is many orders of magnitude faster than McSplit.

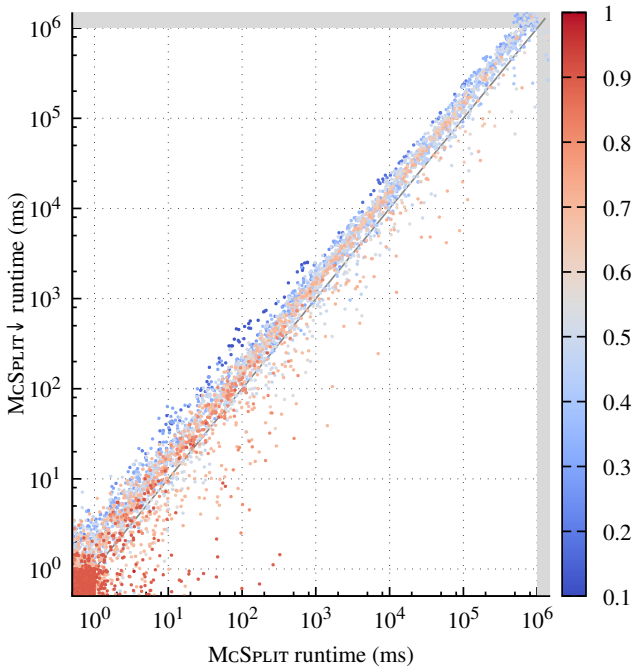
Unlabelled, undirected, connected We now turn to two classes of instance where we require the induced subgraph to be connected. Results for the first—unlabelled and undirected—class are shown in Figure 3.9(a). These results are very similar to the corresponding experiment in Figure 3.7(a) in which the subgraph is not required to be connected: McSplit↓ is the overall winner, and both McSplit variants are more than an order of magnitude faster than the best other algorithm.

Vertex and edge labels, undirected, connected For the labelled, connected case, the clique encoding slightly outperforms both McSplit variants on harder instances (Figure 3.9(b)). However, the gap between the two algorithms is very narrow, and could be 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.

Summary Overall, we find that McSplit improves on the previous state of the art by more than an order of magnitude for unlabelled graphs. On hard labelled instances, the clique encoding remains the strongest solver—particularly where there is no requirement for the common subgraph to be connected.

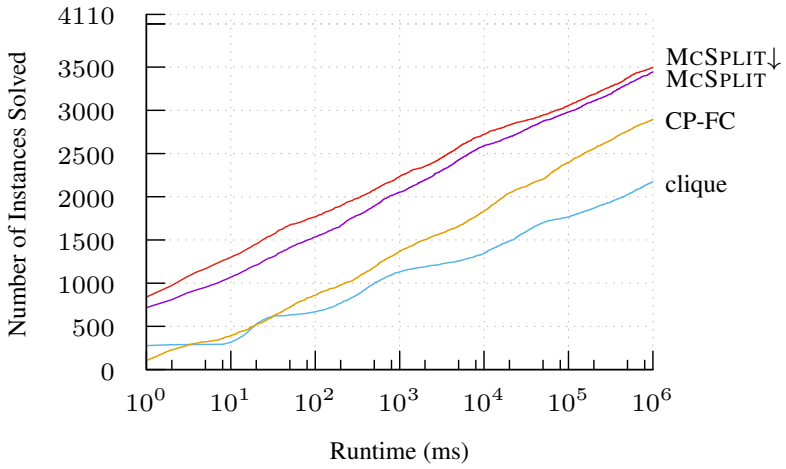


(a) Cumulative number of instances solved

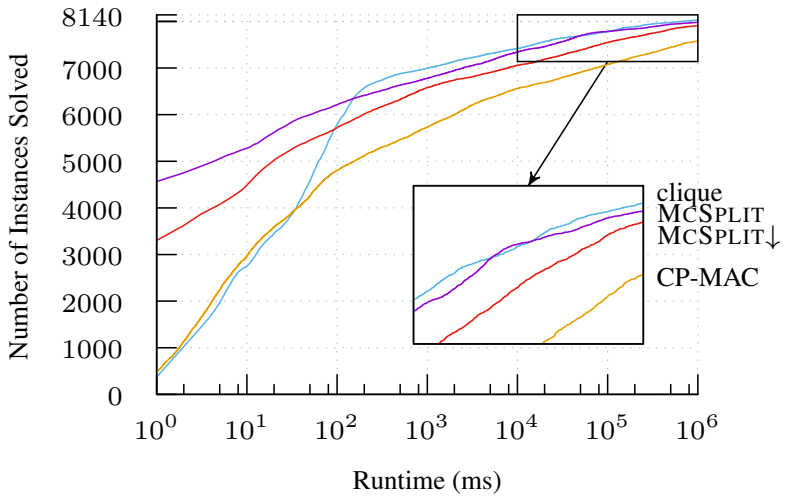


(b) McSplit versus McSplit↓, with one plotted point per instance. The colour of each point shows the order (vertex count) of a maximum common subgraph as a proportion of the order of input graphs G and H.

Figure 3.8: Runtimes for maximum common induced subgraph (with no requirement for the common subgraph to be connected): vertex and edge labelled, directed instances.



(a) Unlabelled, undirected, connected



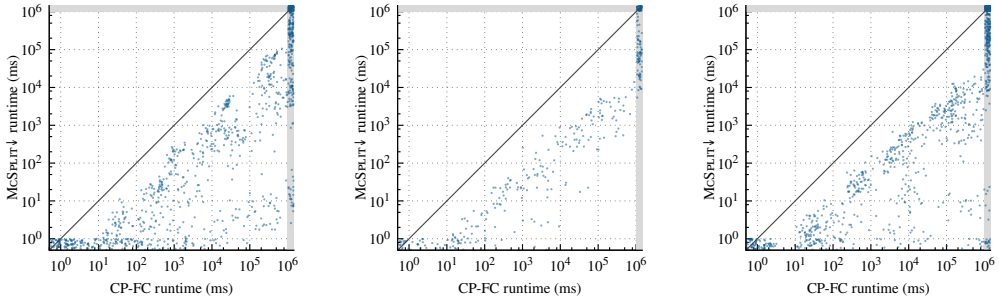
(b) Vertex and edge labelled, undirected, connected

Figure 3.9: Cumulative numbers of instances solved over time for maximum common *connected* subgraph problems.

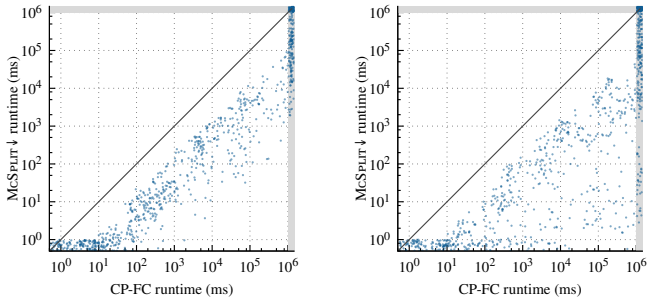
3.7.1 Run Time Comparisons by Family: Unlabelled Instances

We have seen so far that McSplit and McSplit↓ are faster than both the clique encoding and k↓ for the unlabelled instances. This section presents more detailed scatter plots of per-instance run times, with one plot for each of the five families of instances: Erdős-Rényi random graphs, regular and irregular meshes, bounded valence (BV) graphs, and irregular BV graphs. In each case, the run time of McSplit↓ is on the vertical axis.

Figure 3.10 compares CP-FC and McSplit↓. On each of the families of instances, McSplit↓ is substantially faster than CP-FC. Moreover, there is not a single instance where CP-FC is faster than McSplit↓. (Some points appear above the $y = x$ line at the bottom left of the plots due to jittering.)



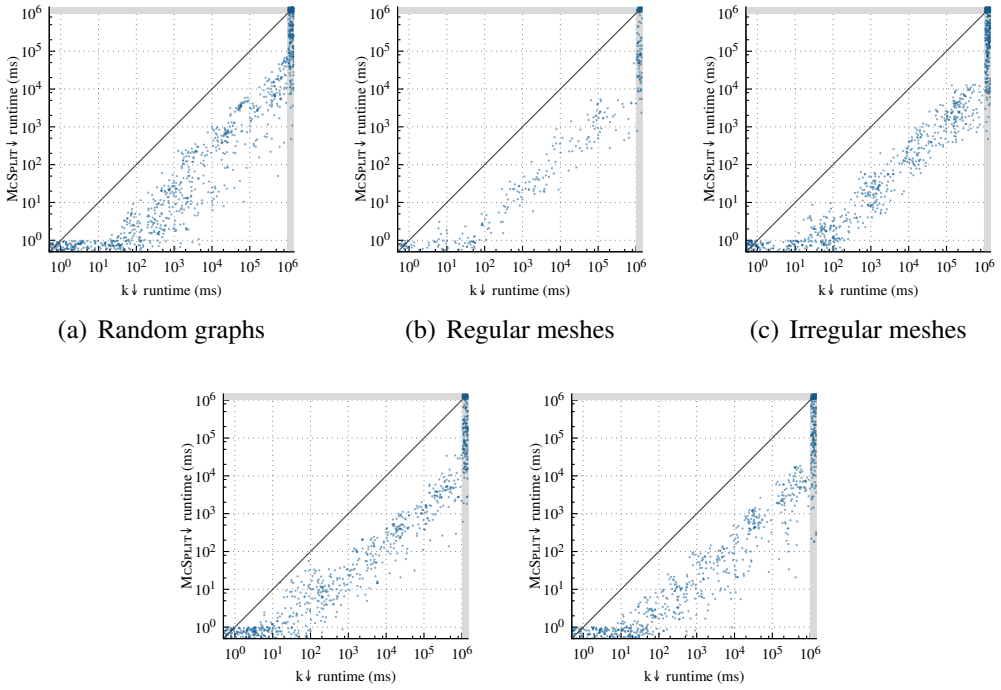
(a) Random graphs (b) Regular meshes (c) Irregular meshes



(d) Bounded valence graphs (e) Irregular BV graphs

Figure 3.10: McSplit↓ and CP-FC run times for unlabelled instances, with one plot per family of instances.

Figure 3.11 compares k↓ and McSplit↓. Again, McSplit↓ is consistently the faster algorithm across all five instance families.

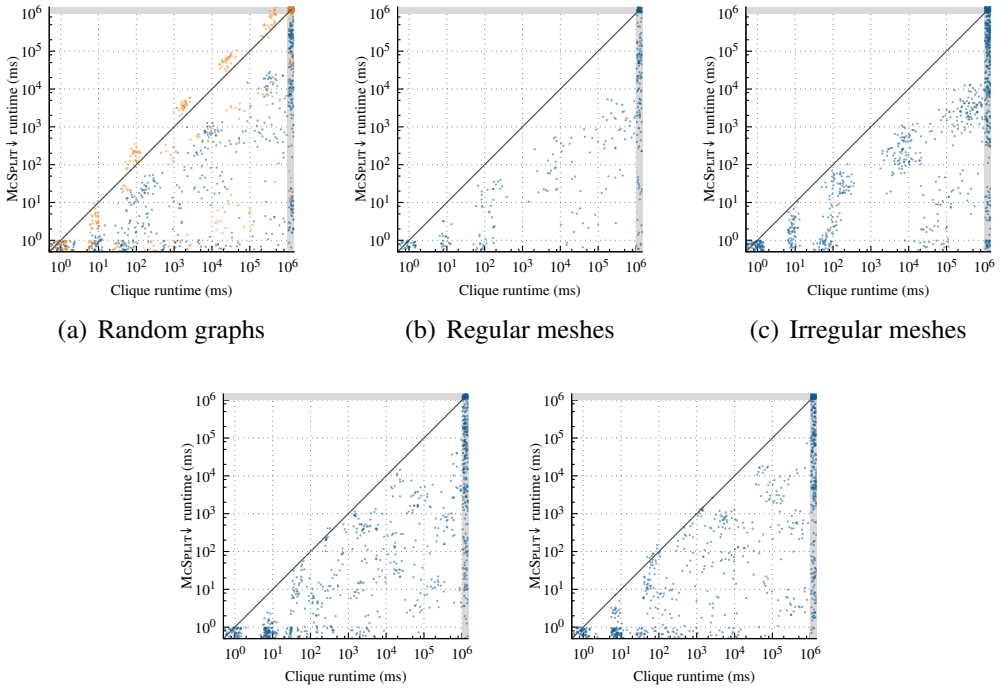


(a) Random graphs (b) Regular meshes (c) Irregular meshes

(d) Bounded valence graphs (e) Irregular BV graphs

Figure 3.11: McSplit↓ and k↓ run times for unlabelled instances, with one plot per family of instances.

Figure 3.12 compares the clique encoding and McSplit↓. In each instance class, McSplit↓ outperforms the clique algorithm overall. However, the results are not as clear-cut as in the CP-FC and k↓ comparisons. In particular, a number of instances in the Random family were solved more quickly by the clique solver than by McSplit↓. Of the 2398 instances in this family that were solved by at least one of these two solvers within the time limit, the clique encoding was faster than McSplit↓ on 125 instances. All but 11 of these 125 instances were in the densest subset of the Random instances: graphs with each edge generated with probability 0.2, shown as orange points on the plot. (Other instances in the Random family had edges generated with probabilities 0.01, 0.05, and 0.1.) This suggests that, for random instances, the advantage of McSplit↓ is greatest in sparse graphs.



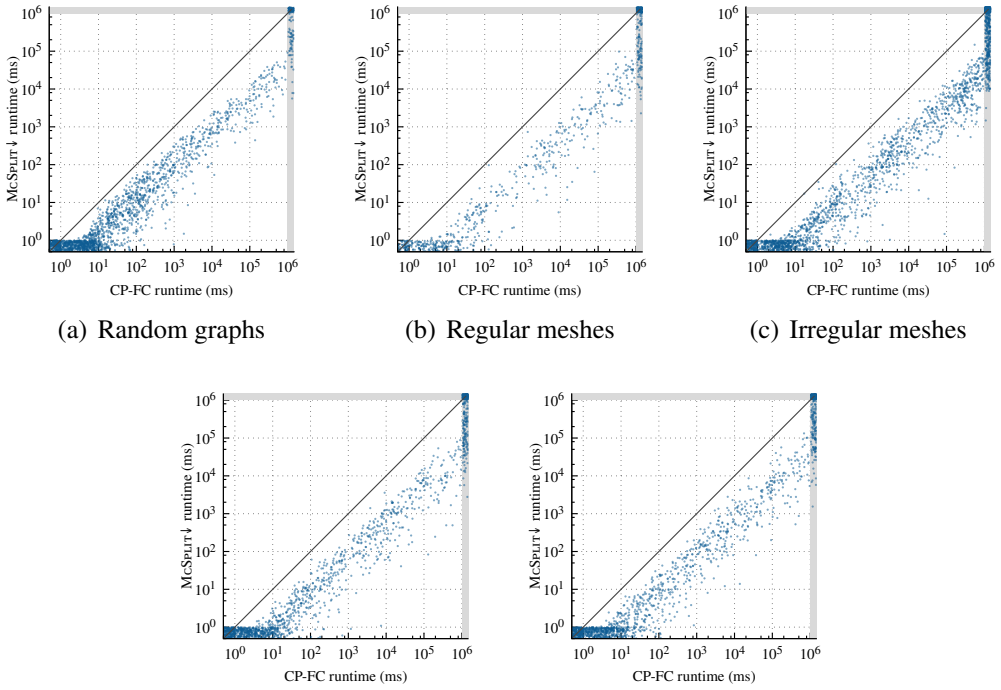
(a) Random graphs (b) Regular meshes (c) Irregular meshes

(d) Bounded valence graphs (e) Irregular BV graphs

Figure 3.12: McSplit↓ and clique run times for unlabelled instances, with one plot per family of instances. In the first plot, the densest instances ($p = 0.2$) are highlighted in orange.

3.7.2 Run Time Comparisons by Family: Labelled Instances

Figure 3.13 compares McSplit↓ and CP-FC run times on the five families of instances with labels on vertices and edges. As in the unlabelled case, McSplit↓ is the clear winner for each of the five families.

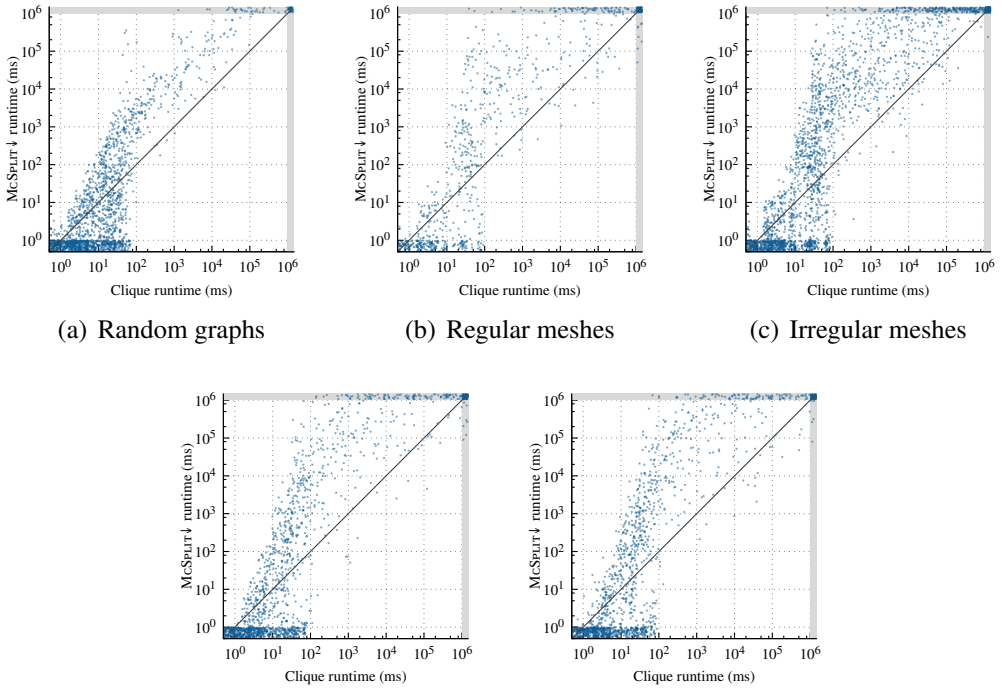


(a) Random graphs (b) Regular meshes (c) Irregular meshes

(d) Bounded valence graphs (e) Irregular BV graphs

Figure 3.13: McSplit↓ and CP-FC run times for vertex and edge labelled instances, by family.

Figure 3.14 compares McSplit↓ and the clique encoding on the labelled instances. In each of the five families, the clique algorithm is the clear winner for all but the most trivial instances.



(a) Random graphs (b) Regular meshes (c) Irregular meshes

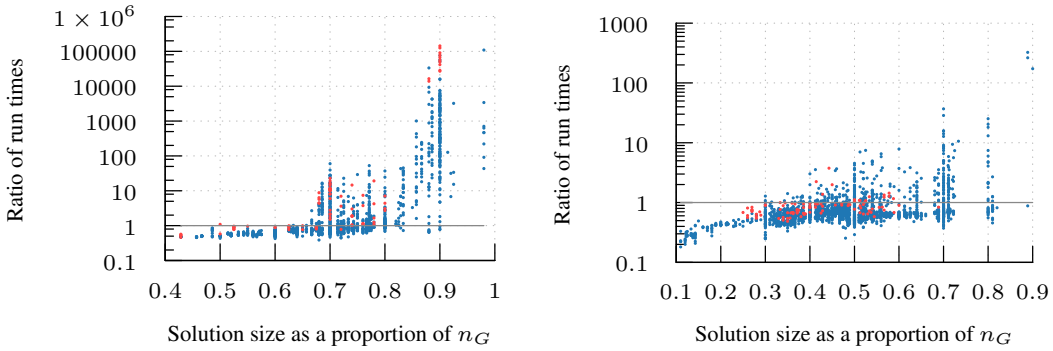
(d) Bounded valence graphs (e) Irregular BV graphs

Figure 3.14: McSplit↓ and clique run times for vertex and edge labelled instances, by family.

3.7.3 On Which Instances Should we Choose McSplit↓ Over McSplit?

On which instances is it better to use McSplit↓ than McSplit? We have already seen from the colour coding in Figures 3.7 and 3.8 that McSplit is often slightly faster than McSplit↓ on instances where the maximum common subgraph is much smaller than the input graphs, and that McSplit↓ is often much faster than McSplit on instances with a large maximum common subgraph. Figure 3.15 examines this relationship in more detail. In each plot, we have one point per instance. The vertical axis measures the ratio of run times; values above 1 indicate that McSplit↓ is faster than McSplit. The horizontal axis measures the number of vertices in a maximum common subgraph as a proportion of the number of vertices in each of the two input graphs.

For both unlabelled and labelled instances, there is a positive relationship between these two measures. Above the point where the maximum common subgraph is 70% as large as each input graph, McSplit↓ becomes the faster algorithm.



(a) Unlabelled instances

(b) Vertex and edge labelled

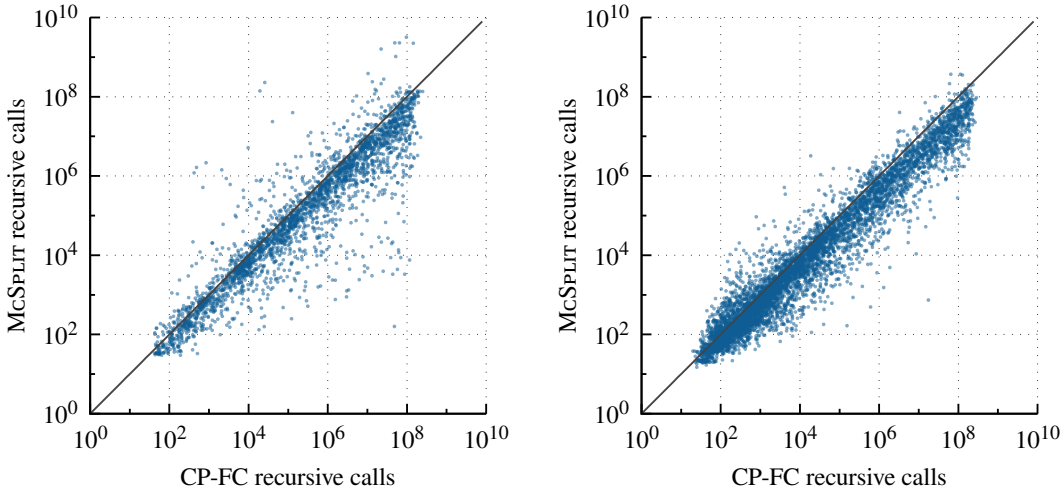
Figure 3.15: McSplit↓ outperforms McSplit on instances with a large maximum common subgraph. The vertical axis on each plot measures the ratio of run times (McSplit time divided by McSplit↓ time) The horizontal axis measures the solution size divided by the number of vertices in an input graph. (For each instance, the two input graphs had the same number of vertices). Trivial instances that could be solved by either algorithm in less than 1 ms or by both algorithms in less than 100 ms are excluded. Red points show instances on which at least one algorithm timed out; timeouts are counted as 1000 seconds.

3.8 Comparison With Existing Algorithms

The CP-FC and McSplit bounds are equal Our experimental results suggest that McSplit 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. In Figures 3.16(a) and 3.16(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. 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 similarly close correlation between the search tree size of McSplit and that of the clique approach (Figure 3.17). 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 $v \in \mathrm{V}(G)$ is represented by a variable, whose domain corresponds to the set of vertices in $\mathrm{V}(H)$ to which v may currently be mapped, with an additional special $\perp$ value representing an unmapped vertex. Given a label class $\langle S_G, S_H \rangle$ in McSplit the vertices in S_G correspond to variables in CP-FC, and the vertices in S_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).



(a) Unlabelled, undirected instances

(b) Labelled, directed instances

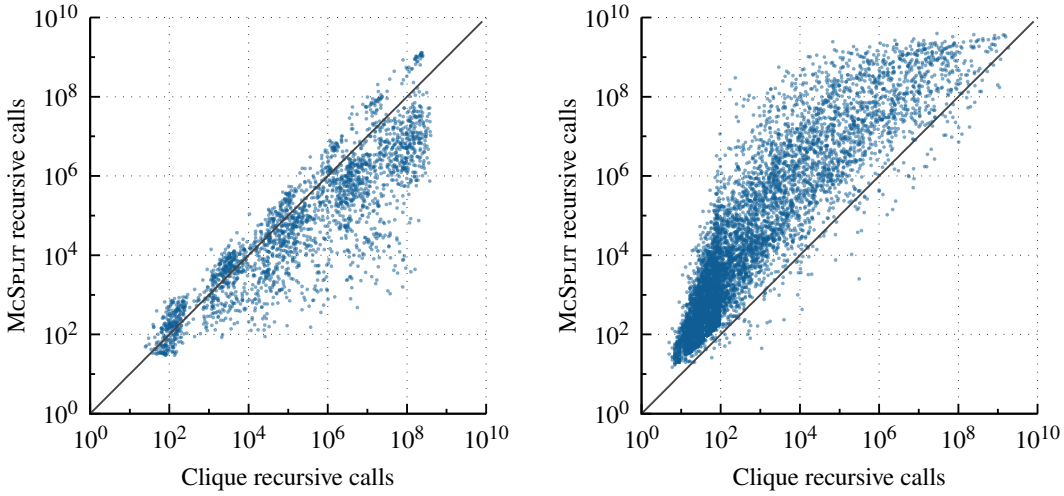
Figure 3.16: Search space sizes for instances which were solved by both algorithms within the timeout: CP-FC and McSplit.

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 G and H in Figure 3.1. 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 B in Figure 3.18 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 of the compatibility graph 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 subgraph, and there are no edges between 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



(a) Unlabelled, undirected instances

(b) Labelled, directed instances

Figure 3.17: Search space sizes for instances which were solved by both algorithms within the timeout: clique algorithm and McSplit.

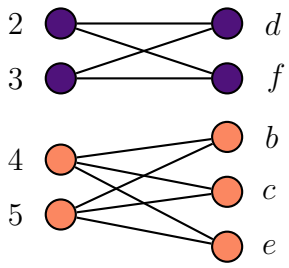


Figure 3.18: A bipartite compatibility graph of the type used to calculate a bound in the CP-FC algorithm.

the left and right sets of vertices. It follows that the CP-FC bound is identical to the McSplit bound.

Comparison of CP and clique bounds The upper bound shared by CP-FC and McSplit can be greater or less than the clique bound. To compare the clique bound and the McSplit bound and to give some insight into why the clique algorithm is so effective for labelled graphs, we consider two extreme cases: unlabelled input graphs and input graphs in which all vertex labels are distinct in each graph. To preview the result: in the unlabelled case, the bounds are incomparable, while in the all-labels-distinct case, the clique bound is always at least as strong as the McSplit bound.

We begin with the unlabelled case (or equivalently, the case in which all vertex labels are equal). Figure 3.19 shows an example in which the clique bound is at least as strong as the McSplit bound. The input graphs are K_2 and I_2. The association graph is the independent set I_4, and therefore a greedy colouring will yield a single colour class, resulting in an upper

bound of 1 on the clique number. McSplit gives an upper bound of 2, since there is a single label class containing two vertices from each graph.

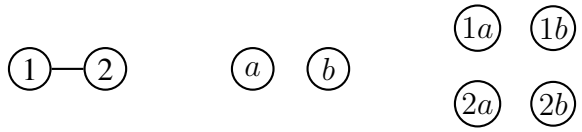


Figure 3.19: Graphs G and H and their association graph A. McSplit gives an upper bound of 2 on the order of a maximum common induced subgraph of G and H, whereas a clique algorithm computing a greedy colouring of A gives a tighter upper bound of 1.

Figure 3.20 shows an example in which McSplit's bound may be stronger than or equal to the clique bound depending on the order in which the greedy colouring is carried out. The input graphs are K_2 and K_3. Clearly, the McSplit bound is 2. A colouring of the association graph has size either 2 (if we assign $1a$, $1b$, and $1c$ to one colour class, and $2a$, $2b$, and $2c$ to another) or 3 (if the colour classes are $\{1a, 2a\}$, $\{1b, 2b\}$, and $\{1c, 2c\}$).

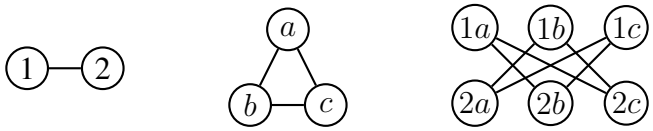


Figure 3.20: Graphs G and H and their association graph A. McSplit gives an upper bound of 2 on the order of a maximum common induced subgraph of G and H. A greedy colouring of A gives an upper bound of 2 or 3, depending on the order in which vertices are coloured.

We now turn to the extreme case in which all vertex labels are distinct in each of the two input graphs. Figure 3.21 shows two graphs, each with three distinctly labelled vertices; the labels are represented by three different shapes. The McSplit upper bound is 3, since there are three label classes, each containing one vertex from each graph. A greedy colouring of the association graph gives a stronger bound, 2, irrespective of the order in which the nodes are coloured, since nodes $1a$ and $3c$ will be assigned to the same colour class.

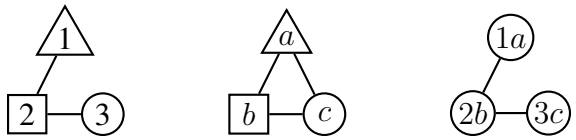


Figure 3.21: Graphs G and H and their association graph A. Vertex labels are represented by shapes. McSplit gives an upper bound of 3 on the order of a maximum common induced subgraph of G and H. A greedy colouring of A gives an upper bound of 2.

Clearly, this example generalises: if all of the vertices in each input graph have distinct labels, then the clique bound is less than or equal to the McSplit bound.

We have established that the McSplit and clique upper bounds are incomparable for unlabelled graphs, and that the clique algorithm's bound is stronger than the McSplit bound if there are very many vertex labels. In fact, a bound calculated using greedy colouring can

always be at least as good as the McSplit bound if we colour vertices in the right order; for example, if we colour the vertices as described in the proof in Section 3.3.

Experimental comparison of bounds Is the clique bound stronger than the McSplit bound in practice, even for unlabelled graphs? We will see shortly that the answer is "no"—at least if we use MCSa1 as the clique algorithm. To find the point at which the clique bound becomes preferable, I carried out a small experiment on $G(n, p)$ random graphs, varying the size of the set of vertex labels.

I generated 1800 pairs of random undirected, loopless graphs of order 20 without edge labels, with densities ranging from 0.1 to 0.9 in steps of 0.1. For each integer value of a parameter m (the maximum label) from 1 to 20, I generated 90 pairs of graphs with vertex labels assigned uniformly at random from the set $\{1, \ldots, m\}$. I used my implementation of McSplit and Prosser's implementation of the MCSa1 algorithm (Prosser, 2012) to find the McSplit and clique bounds respectively at the top of search (that is, on the first recursive call, before any vertex assignment decisions have been made).

Figure 3.22 plots the results of dividing the McSplit upper bound by the clique upper bound, with one point per instance. The value of m parameter is shown on the horizontal axis; points towards the right of the plot have more distinct vertex labels. Mean values are shown as horizontal lines. When $m = 1$—that is, when all vertices have the same label—the McSplit bound is strictly tighter than the clique-algorithm bound on all 90 of our instances. On average, the McSplit bound is tighter than the clique bound for $m < 4$, and worse than the clique bound for $m > 5$. Of the 900 instances with $m > 10$, McSplit had a tighter bound than the clique algorithm for only a single instance.

To summarise, the upper bound of the clique encoding dominates that of CP-FC and McSplit where vertices are "as labelled as possible", whereas neither bound dominates for unlabelled graphs. Experimentally, we see that the McSplit bound is tighter than the clique bound for unlabelled graphs, but the tightness of the clique bound steadily increases relative to that of the McSplit bound as the number of vertex labels is increased. This provides an explanation for the behaviour we saw in Section 3.7, where McSplit was the fastest solver for unlabelled instances while the maximum clique solver was fastest for instances with many distinct labels on vertices.

3.9 Publications Using McSplit

Since the McSplit algorithm was published at IJCAI 2017, a number of papers, including two that I co-authored, have further investigated the algorithm. This section briefly surveys

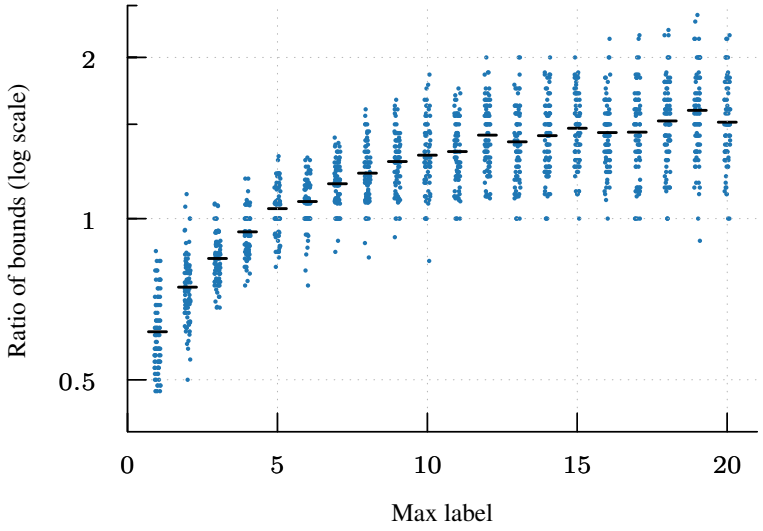


Figure 3.22: As the number of vertex labels is increased, the clique algorithm's bound becomes stronger than the McSplit bound. The vertical axis shows the ratio of initial McSplit upper bound to initial clique algorithm (MCSa1) upper bound for 1800 instances, with one dot per instance; for values less than 1, McSplit's bound is better than that of the clique algorithm. The horizontal axis shows the maximum label m; each vertex label is a random integer from the range $[1, m]$. Points are jittered horizontally to reduce overplotting.

this work.

3.9.1 Publications Co-Authored by the Present Author

Archibald et al. (2019) introduce *solution-biased search*, a technique using restarts and no-good learning that allows an algorithm to quickly explore diverse parts of the search tree rather than explore it in depth-first order (as a typical backtracking algorithm such as the standard version of McSplit does); this can often help the algorithm to find good solutions quickly. For the paper, I designed and implemented a version of McSplit using solution-biased search; this led to modestly improved run times on many instances.

In Gocht et al. (2020a), the present author added proof logging (Gocht et al., 2020b) to the C++ implementation of McSplit to create a *certified* version of the solver. This outputs a log of the solver's reasoning steps using the cutting plane proof system (Cook et al., 1987), such that the log can be verified using a simple and extensively tested checker program. The proof logging and verification process slows down the solver by around three orders of magnitude; therefore I did not use it for the experiments in the current thesis chapter. However, the program's logs were verified for all 16,300 instances considered for the paper, providing evidence of the implementation's correctness.

3.9.2 Other Publications

Liu et al. (2020) introduce improved variable- and value-ordering heuristics for McSplit using reinforcement learning. Specifically, when selecting a vertex in G or H to use for branching, the algorithm prefers to use vertices that have resulted in a large decrease to the upper bound in previous recursive calls. This strategy enabled the new solver, McSplit+RL, to solve more instances within a time limit than McSplit. Interestingly, a scatter plot in the paper shows that each of McSplit and McSplit+RL is faster than the other algorithm for many instances. This suggests that it could be useful to include both algorithms in a portfolio solver.

Quer et al. (2020) explore several enhancements of McSplit, including a massively parallel GPU implementation using CUDA, a recursion-free implementation, and a parallel portfolio solver containing these along with the original McSplit implementation. The GPU implementation is shown to be promising although not competitive with CPU implementations on the hardware used by the paper's authors; the portfolio solver, however, is able to solve more instances within a given time limit than any of the constituent solvers.

Finally, two theses — the undergraduate thesis of Paulius Dilkas and the Masters thesis of Jonathan Trummer — make contributions in both algorithm design and experimental work.

Dilkas (2018) performs a detailed experimental evaluation of McSplit and a clique algorithm for a range of labelling schemes, finding that McSplit outperforms the clique encoding even for labelled graphs, if only a small number of labels are used. Dilkas uses machine learning to construct a portfolio solver that chooses intelligently between McSplit and the clique encoding according to easily computed characteristics of the two input graphs such as density and standard deviation of degrees. The portfolio solver outperforms each of its component algorithms, giving overall performance close to that of the virtual best solver. Finally, Dilkas constructs a hybrid algorithm, Fusion, that switches from McSplit to the clique encoding at a specified depth of recursion.

Trummer (2021) presents several innovations for maximum common induced subgraph algorithms, including improved reinforcement learning for McSplit based on Liu et al. (2020), and a version of McSplit with symmetry breaking.

3.9.3 An Application to a Question in Probability Theory

Chatterjee and Diaconis (2021) consider taking two independent random $G(N, 1/2)$ graphs G and H, and finding the order of their maximum common induced subgraph. They prove that as $N \to \infty$, the maximum common subgraph of G and H will, with probability tending to 1, be in a range consisting of either one integer or two consecutive integers. This range is

given by $\{\lfloor x_N - \varepsilon_N \rfloor, \lfloor x_N + \varepsilon_N \rfloor\}$, where $x_N = 4\log_2 N - 2\log_2\log_2 N - 2\log_2(4/e) + 1$ and $\varepsilon_N = (4\log_2 N)^{-1/2}$. It is interesting to examine whether this precise prediction of the order of a maximum common induced subgraph comes close to holding, even for small values of N.

Figure 3.23 shows data provided by the present author to Chatterjee and Diaconis; a subset of this data is presented in their paper. The broad blue bands give the range of predicted MCIS orders for a given N; each band is either a single integer or a range of two consecutive integers. The orange points show the mean MCIS orders for randomly generated pairs of graphs, and the bands show the range of MCIS orders observed. For $N \leq 32$, 100 pairs of graphs were generated. Due to the difficulty of solving larger instances, one pair of graphs was generated for each N in the range $32 < N \leq 45$. For values of N above 27, the theorem of Chatterjee and Diaconis is a good predictor of the MCIS size, differing by at most 1 from the values observed in our generated data.

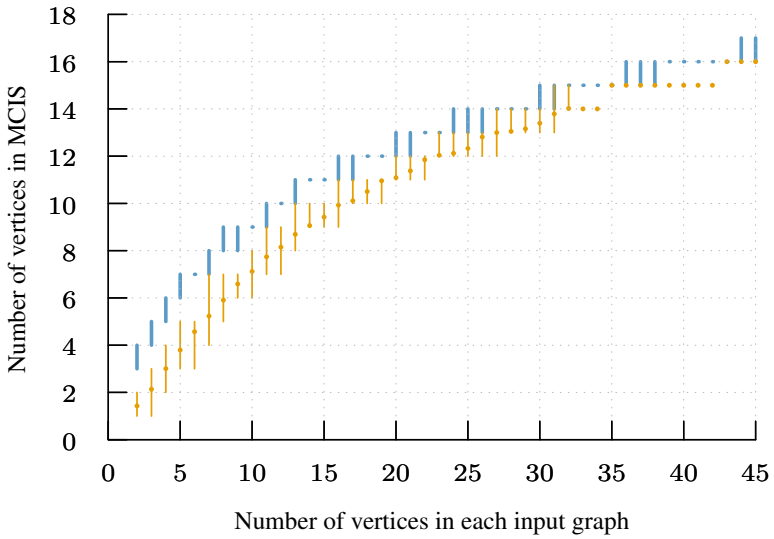


Figure 3.23: For each N, the blue bands give the range of predicted maximum common induced subgraph orders given by the asymptotic formula by Chatterjee and Diaconis Chatterjee and Diaconis (2021). For $N \leq 32$, 100 instances were generated; the orange bands give the range of MCIS orders observed, and the orange points give the means. For $32 < N \leq 45$, a single instance was generated; the MCIS order is shown as a single point.

3.10 Conclusion

This chapter has introduced the McSplit algorithm for maximum common induced subgraph problems. This algorithm is more than an order of magnitude faster than the previous state of the art for unlabelled, undirected instances. For instances with many distinct vertex and edge labels, however, the MCSa1 maximum clique algorithm remains the fastest solver.

We have seen how the McSplit 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 have explored the large effect that sorting graphs by degree can have on run times; experiments on random graphs suggest that it is helpful to sort both graphs in increasing order of degree if the second graph is dense, and in decreasing order otherwise. Finally, we have examined the relationship between the upper bounds calculated by McSplit and by clique algorithms.

The next chapter introduces new variants of McSplit that explore the effects of swapping graphs G and H. Section 5.9 will return once more to maximum common induced subgraph, introducing a variant of McSplit that is optimised for large, sparse graphs.

Chapter 4

Swapping Graphs and Two-Sided Branching in McSplit

4.1 Introduction

A maximum common subgraph of G and H is clearly also a maximum common subgraph of H and G. In this chapter we examine whether it can be useful to swap the two input graphs when calling an MCIS solver. We will see that swapping the graphs can have a very large effect on the run time of McSplit if the two graphs differ in density or order, and that simple rules can be devised to take advantage of this. The second contribution of this chapter is to introduce McSplit-2S, a modified version of McSplit that generalises the idea of swapping graphs by choosing at each search node which side of a label class to branch on. Our experimental evaluation finds that this algorithm improves upon McSplit, albeit modestly, for some classes of graphs.

Section 4.2 introduces two simple and very effective heuristics for deciding whether to swap graphs. Section 4.3 speculates about reasons for the success of the swapping rules. Section 4.4 introduces McSplit-2S, a version of McSplit that can branch on two sides. Section 4.5 discusses related work, and Section 4.6 concludes.

4.2 When Should we Swap the Graphs?

We refer to the swapping version of McSplit as McSplit-Swap; this is shown in Algorithm 5. McSplit-Swap simply calls McSplit with the two graphs interchanged, then returns the resulting mapping with its elements reversed. We begin by plotting run times of McSplit with and without swapping graphs, to give an idea of the difference that swapping can make.

Algorithm 5: McSplit-Swap: a version of McSplit that swaps the input graphs.

```
1 McSplit-Swap(G, H)
2 begin
3     M ← McSplit(H, G) ▷ Call McSplit with the graphs swapped
4     return {(w, v) | (v, w) ∈ M} ▷ Reverse the mapping
```

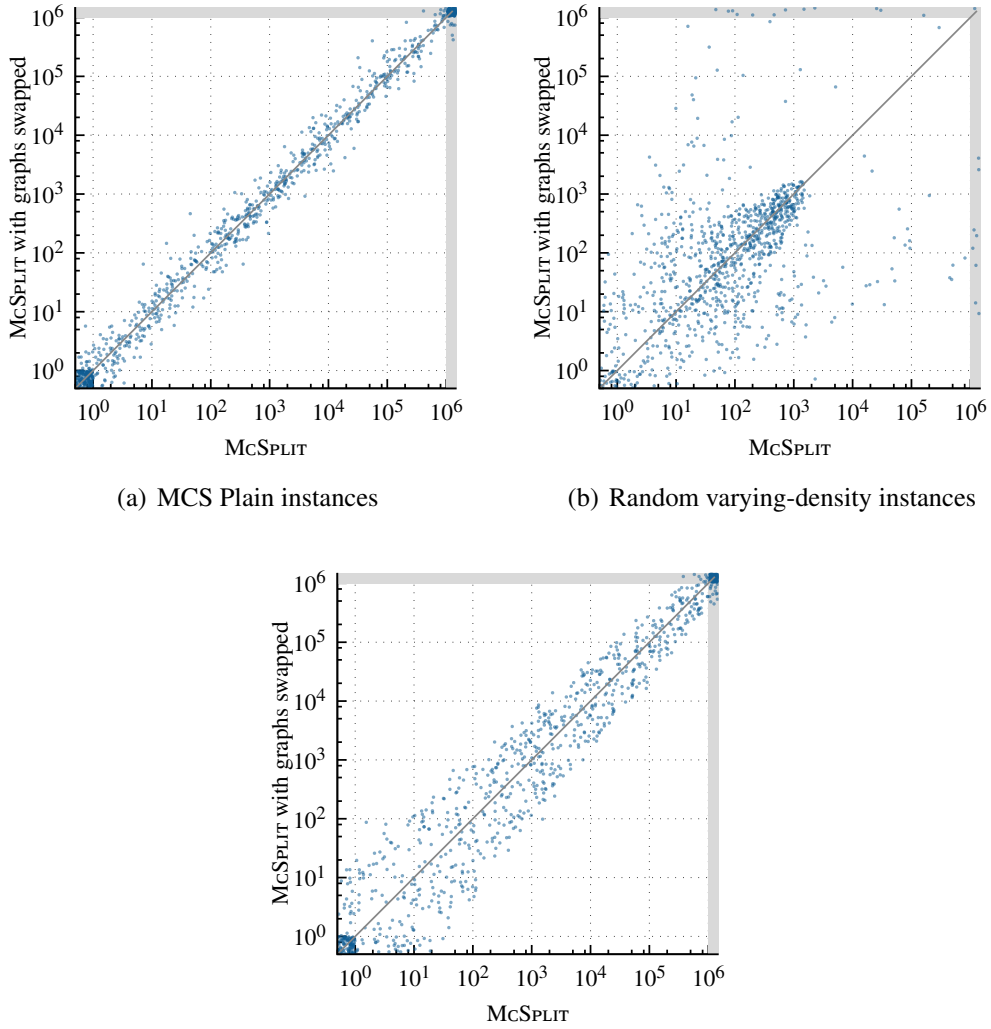
Figure 4.1(a) shows run times for unlabelled, undirected instances from the database described in Section 3.5; we will refer to these as the MCS Plain instances. As with all experiments in this chapter, the plot use a random sample of 1000 of the benchmark instances. The horizontal axis shows run time in milliseconds without swapping the graphs, and the vertical axis shows run time with swapping. Although swapping graphs can make a small difference, it does not change run time by as much as an order of magnitude for any of these instances.

The fact that swapping makes little difference to run time on the MCS Plain instances can be explained by the way these instances are generated. Within each MCS Plain instance, the two graphs are produced using the same graph generator, have the same number of vertices, and have very similar density (Figure 4.2). Given the similarity of the two graphs in each instance, it is unsurprising that swapping the graphs has little effect on run time.

To examine pairs of graphs with very *different* characteristics, we generated two sets of random instances using the Erdős-Rényi $G(n, p)$ model. The first set has 24 vertices per graph, with the p parameter of the generator chosen randomly for each graph from the interval $[0, 1]$; thus, the two graphs in an instance have the same order but can vary greatly in density. The second set of instances has the density parameter p fixed at 0.3 for all instances, with the number of vertices in each graph randomly chosen from $\{10, 11, \ldots, 50\}$. These instances will be used throughout this chapter; we call them the *varying-density instances* and the *varying-order instances* respectively.

Figure 4.1(b) shows run times with and without swapping for the varying-density instances. For some of these instances swapping graphs is very clearly beneficial, and for others, swapping greatly increases the run time. Figure 4.1(c) shows run times with and without swapping for the varying-order instances. Here again we see that swapping can be beneficial, albeit to a lesser degree.

We have seen that McSplit and McSplit-Swap can have very different run times. Are there heuristics we can use to choose the better algorithm for a given instance? We now turn to this question, beginning with the varying-density instances.



(a) MCS Plain instances

(b) Random varying-density instances

(c) Random varying-order instances

Figure 4.1: Run times in ms for McSplit (horizontal axis) and McSplit-Swap (vertical axis). Each point represents an instance. Swapping the graphs has little effect on run time for MCS Plain instances, but changes the run time by orders of magnitude for many of the random instances.

4.2.1 Random Graphs With Fixed n and Varying Density

Can we tell in advance whether we should swap the graphs of a given instance? Figure 4.3(a) shows that, in the case of our random varying-density instances, there is a very strong association between the densities of the two graphs and whether we should swap graphs. In this plot, the axes measure graph density.[1] Colour is used to show the effect of swapping graphs on run time; red dots represent those instances that are solved more quickly with the graphs swapped. The diagonal lines on the plot show the curves $y = x$ and $y = 1 - x$; these divide the plot into four triangular regions. The figure shows clearly that instances lying in the upper

[1]By *density*, we are referring to the actual density of a graph, $\frac{|E(G)|}{n_G(n_G-1)/2}$, rather than the parameter p used to generate the graph. In practice the two measures are very similar.

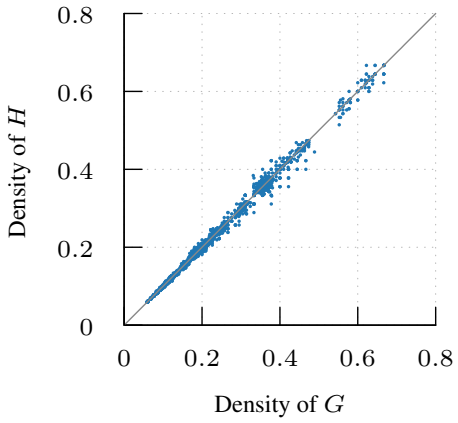
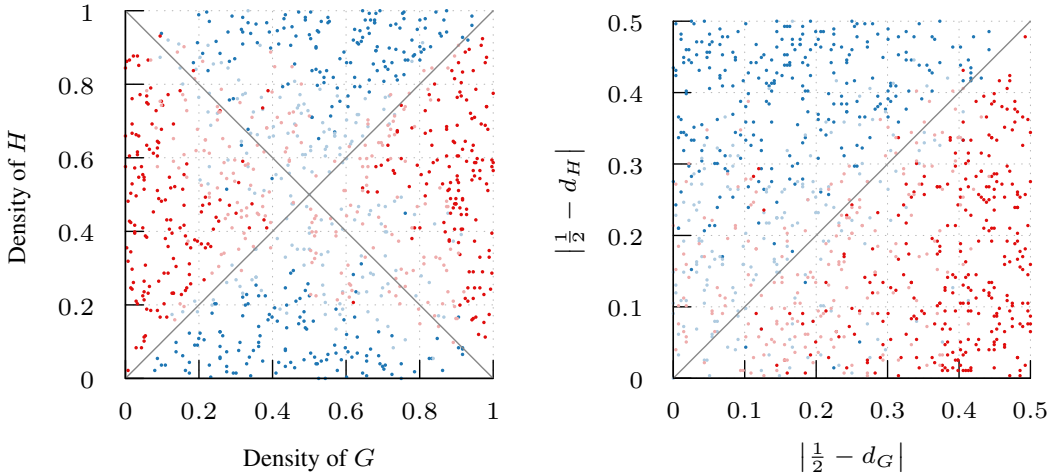


Figure 4.2: In each pair of MCS plain instances, the two graphs have very similar densities. The figure plots the densities of graphs G and H in each graph pair.

and lower triangles tend to be solved more quickly by McSplit, while those lying in the left and right triangles tend to be solved more quickly by McSplit-Swap.



(a) Density

(b) "Extremeness" of density

Figure 4.3: There is a strong association between the densities of graphs G and H and whether it is beneficial to use McSplit-Swap rather than McSplit. The first subfigure plots density of G against density of H, with one point plotted for each of the random varying-density instances. Instances for which McSplit-Swap is faster are plotted in red; instances where McSplit is faster are plotted in blue. Dark red and dark blue indicate that McSplit-Swap and McSplit, respectively, result in run times at least twice as fast as the alternative.

The union of the left and right triangles in Figure 4.3(a) has a simple characterisation. Define the measure *density extremeness* of a graph as $\left|\frac{1}{2} - d\right|$, where d is the graph's density. This measures how close a graph's density is to either 0 or 1; a clique and an independent set have the highest possible density extremeness. The union of the left and right triangles in Figure 4.3(a) contains exactly those graph pairs (G, H) such that the density extremeness of G is greater than the density extremeness of H. The second subfigure, Figure 4.3(b), replots the data in Figure 4.3(a), with density extremeness rather than density measured on the axes.

It is evident from the figure that McSplit-Swap almost always runs faster than McSplit on instances where the density extremeness of G exceeds the density extremeness of H.

Given the strong relationship between the densities of G and H and the optimal order in which to pass the graphs to McSplit, a natural next step is to devise a version of McSplit that swaps the graphs if and only if the density extremeness of G is greater than the density extremeness of H. We call this algorithm McSplit-SD (with SD standing for "swapping on density"). It is shown in Algorithm 6.

Algorithm 6: McSplit-SD: a version of McSplit that uses density to decide whether to swap the input graphs.

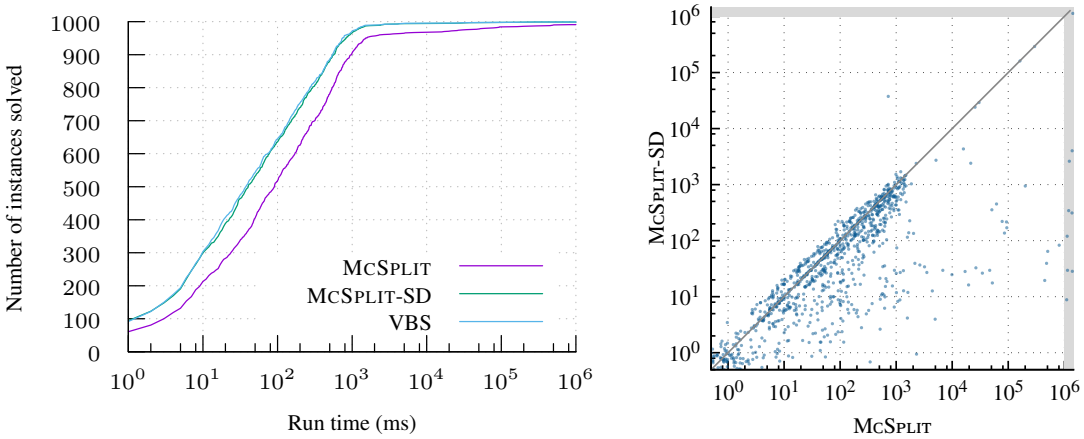
```
1 McSplit-SD(G, H)
2 begin
3     if |1/2 − d_G| > |1/2 − d_H| then
4         return McSplit-Swap(G, H)
5     return McSplit(G, H)
```

Figure 4.4 compares on our set of varying-density random instances the run times of three solvers: McSplit, McSplit-SD, and the virtual best solver (VBS) of McSplit and McSplit-Swap. The VBS time is computed by running McSplit and McSplit-Swap, then recording the lower of the two run times. It can be viewed as the run time of an algorithm that uses an oracle to determine whether to swap the two graphs. The cumulative plot shows that McSplit-Swap and the VBS have almost identical performance overall, and that both outperform McSplit. The scatter plot in Figure 4.4(b) shows that McSplit-SD is much slower than McSplit on only a single instance, and is orders of magnitude faster on several of the instances. In summary, for this family of instances, McSplit-SD clearly improves upon McSplit, and makes near-perfect decisions when selecting between McSplit and McSplit-Swap.

4.2.2 Random Graphs With Similar Density and Varying n

We now turn to our family of varying-order instances: random instances generated with $p = 0.3$ and varying values of n. Figure 4.5 shows a strong relationship between the graphs' orders and the relative run times of McSplit and McSplit-Swap: McSplit-Swap is preferable if and only if G has more vertices than H. (Instances on which both algorithms reached the time limit are not shown; this explains the blank region at the top right.)

McSplit-SO, an algorithm that swaps the graphs if the order of G is greater than the order of H, is shown in Algorithm 7. Figure 4.6 shows the run times of McSplit, McSplit-SO, and the VBS of McSplit and McSplit-Swap for our varying-order instances. The McSplit-SO algorithm performs about as well as the VBS, and is not substantially slower



(a) Cumulative plot of instances solved

(b) Run times (ms)

Figure 4.4: The run times of McSplit-SD are faster overall than those of McSplit for the random varying-density instances, and almost indistinguishable from those of the virtual best solver of McSplit and McSplit-Swap.

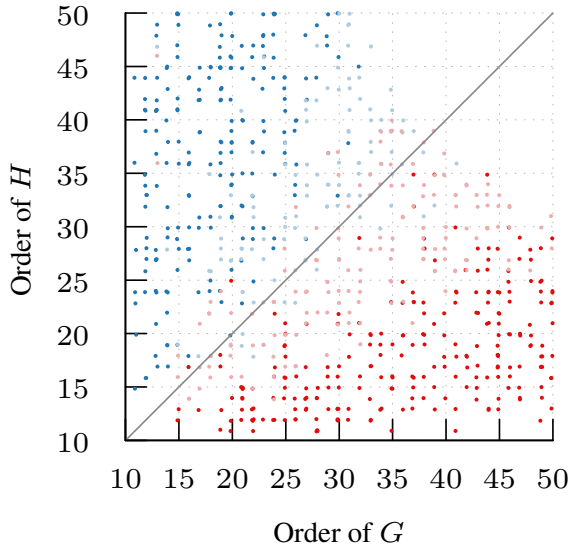
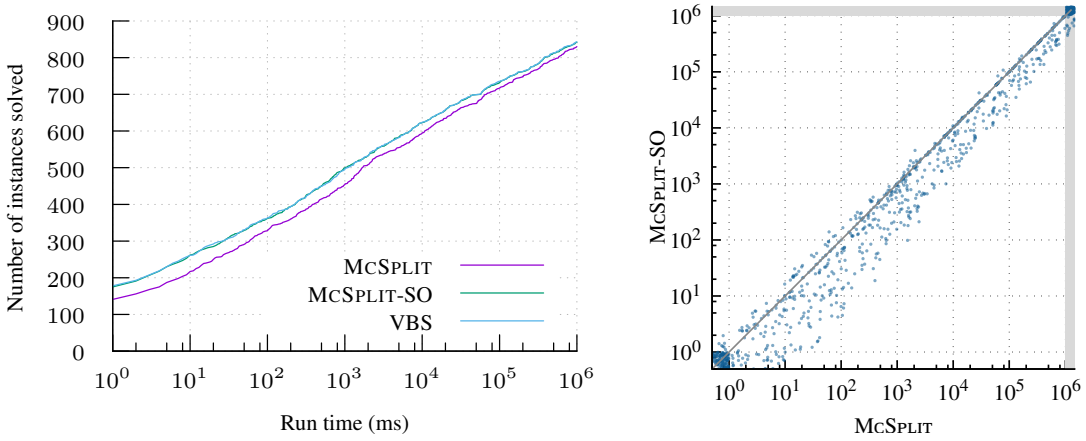


Figure 4.5: For our random varying-order instances, is preferable to use McSplit when G has fewer vertices than H, and to use McSplit-Swap when G has more vertices than H. The plot shows one point per instance. Instances for which McSplit-Swap is faster are plotted in red; instances where McSplit is faster are plotted in blue. Dark red and dark blue indicate that McSplit-Swap and McSplit, respectively, result in run times at least twice as fast as the alternative.

than McSplit on any instance. However, the improvement provided by McSplit-SO is less dramatic than that provided by McSplit-SD; McSplit-SO is seldom more than an order of magnitude faster than McSplit on non-trivial instances.

Algorithm 7: McSplit-SO: a version of McSplit that uses vertex counts to decide whether to swap the input graphs.

```
1 McSplit-SO(G, H)
2 begin
3     if n_G > n_H then
4         return McSplit-Swap(G, H)
5     return McSplit(G, H)
```



(a) Cumulative plot of instances solved

(b) Run times (ms)

Figure 4.6: The run times of McSplit-SD are faster overall than those of McSplit for the random varying-order instances, and almost indistinguishable from those of the virtual best solver of McSplit and McSplit-Swap.

4.3 Explaining the Success of McSplit-SD and McSplit-SO

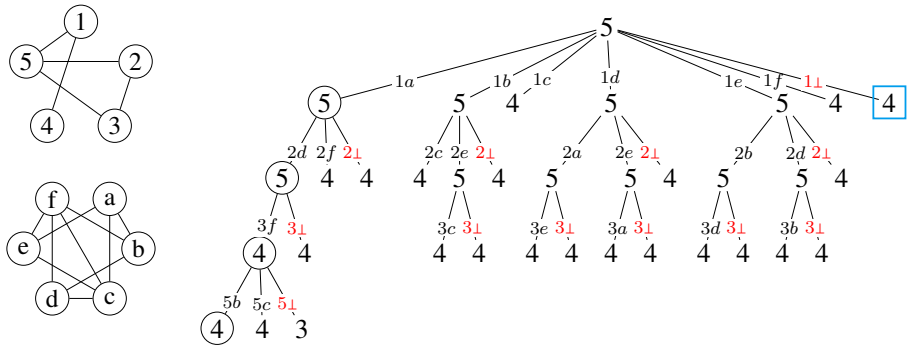
Why are McSplit-SO and McSplit-SD effective? In this section, we give some speculative reasons related to the bound calculated at each search node that go some way towards explaining the success of the swapping heuristics. In each case, we start with an example, then broaden the discussion to the general case.

First, consider McSplit-SO: why it is useful to select the graph with fewer vertices as the first argument when calling McSplit? Let G be a graph with n_G vertices and H be a graph with n_H vertices, such that $n_G < n_H$. If we call McSplit(G, H), the first call to `Search()` in Algorithm 1 makes $n_H + 1$ recursive calls to `Search()`: one for each vertex $w \in V(H)$, and a final call where vertex v of G is rejected. Thus, the root node of the search tree has $n_H + 1$ children. If we reverse the order of the graphs and call McSplit(H, G), a similar argument shows that the root node of the search tree has $n_G + 1$ children. Thus — at least at the root node and plausibly also deeper in the tree — the search tree has a smaller

branching factor if we call McSplit with the *larger* graph first.

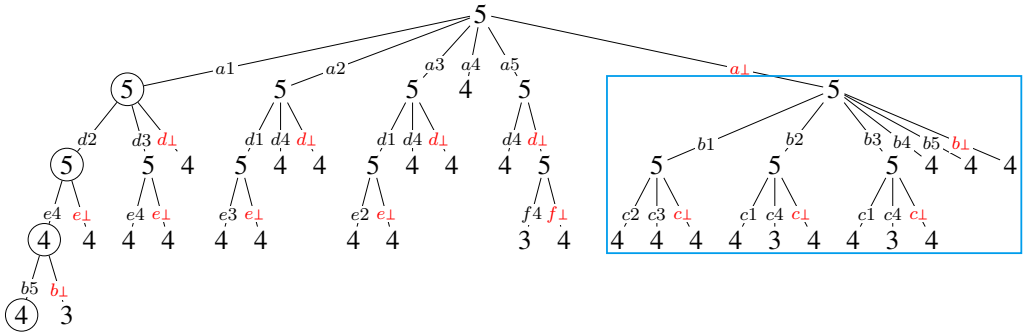
We might expect a smaller branching factor to lead to lower run times, and this suggests that we should place the larger graph first by calling McSplit(H, G). But this is exactly the opposite of the McSplit-SO strategy. Therefore, we must look elsewhere to explain why calling McSplit with the small graph first is effective. It seems likely that the bound calculated on line 4 of Algorithm 1 is part of the explanation.

To give a concrete example, consider the example graphs G and H from Figure 3.1, which are reproduced for convenience in Figure 4.7(a). Figure 4.7(b) shows the search tree if — following the strategy of McSplit-SO — the smaller graph is passed first to McSplit. The upper bound is shown at each node. Figure 4.7(c) shows the search tree if the larger graph is passed first to McSplit. These search trees have 35 and 45 nodes respectively; as we might expect, passing the smaller graph first results in a smaller search tree.



(a) G and H

(b) The search tree of McSplit(G, H)



(c) The search tree of McSplit(H, G)

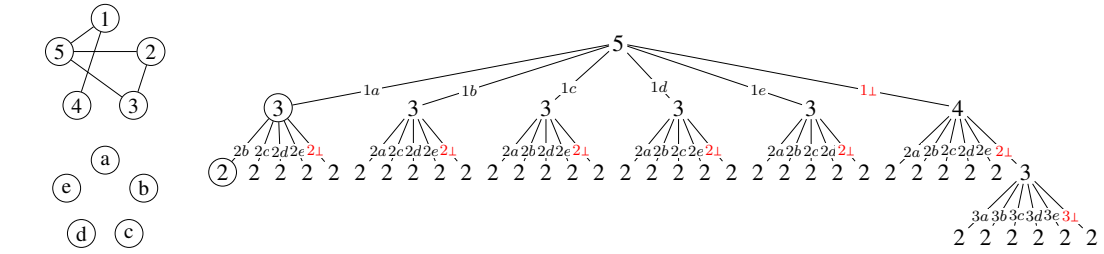
Figure 4.7: The search trees of McSplit and McSplit-Swap on example graphs G and H. The number at each search node shows the computed upper bound, and the label on each edge shows the decision made at that branch. Each blue box highlights the search tree explored after making the decision not to map a vertex in the first call to `Search()`.

The blue rectangle on each search tree highlights the subproblem after the final branching decision in the first call to `Search()` — the decision not to use the first vertex in the first graph (vertex 1 in Figure 4.7(b) and vertex a in Figure 4.7(c)). The size difference between these highlighted subtrees — 1 vertex and 16 vertices — is more than enough to account

for the overall difference in size between the two search trees. The subtree in Figure 4.7(b) is small because after rejecting a vertex of the smaller graph, we can reduce the bound to 4 which lets us prune the search tree immediately. In Figure 4.7(b), the rejected vertex belongs to the larger side of the label class, and therefore the bound remains at 5.

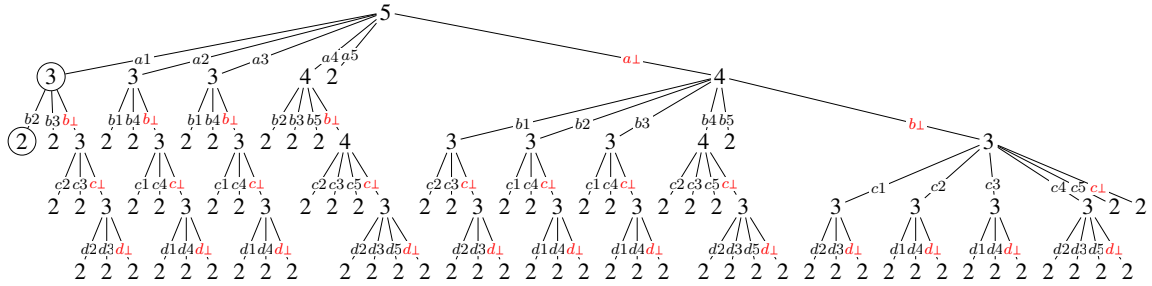
This argument generalises: for any two graphs of unequal order, the final child of the search tree's root node will have a smaller bound if we call McSplit with the smaller graph first than if we call the algorithm with the larger graph first. This provides at least some explanation for the success of McSplit-SO.

We now turn to McSplit-SD. Why might passing the graph with the lowest density extremeness first to McSplit result in a smaller search tree? As an example, we use graph G from the previous example along with I_5, the edgeless graph on five vertices. These graphs have density extremeness 0 and 0.5 respectively — the minimum and maximum possible values. Thus, using the heuristic of McSplit-SD, we would expect the search tree of McSplit(G, I_5) to be smaller than the search tree of McSplit(I_5, G). Indeed, this is the case; the search trees, which are shown in Figure 4.8, have 44 and 97 nodes respectively.



(a) G and I_5

(b) The search tree of McSplit(G, I_5)



(c) The search tree of McSplit(I_5, G)

Figure 4.8: The search trees of McSplit and McSplit-Swap on example graphs G and I_5.

Why might it be preferable to call McSplit with I_5 as the second rather than the first graph? We argue that the reason is similar to the reason we gave for placing the smaller graph first in our previous discussion. It is clear that when calling McSplit with G and I_5 (in either order), there will be exactly one label class throughout search, consisting of the vertices in both graphs that have not been explicitly rejected by the algorithm and that are not adjacent to any vertex that has already been mapped. The side of that label class containing vertices of G will tend to contain fewer vertices than the side containing vertices of I_5, since

the vertex set of I_5 is an independent set. Now consider a red branch in either search tree, in which a vertex is rejected. If that vertex is on the smaller side of the label class, the bound at the child node in the search tree will be one less than the bound at the parent node; otherwise, the bound at the child node will be equal to the bound at the parent node. It is therefore preferable, in our example, to call McSplit with G first. Indeed, in Figure 4.8 we can see that there are ten nodes in the search tree of McSplit(I_5, G) that result from the rejection of a vertex (i.e. have a red-labelled edge above) and have the same bound as their parent. There are no such nodes in the search tree of McSplit(I_5, G).

The average branching factor of the search tree for McSplit(I_5, G) is lower than the average branching factor of the search tree for McSplit(G, I_5). This suggests that with McSplit-SD, as is the case for McSplit-SO, there is a trade-off where the average branching factor suggests we should use one ordering of the graphs, but this is outweighed by the better bounds computed with the opposite ordering.

4.4 McSplit-2S: Generalising McSplit-SD and McSplit-SO

Our arguments in favour of McSplit-SD and McSplit-SO both point towards the benefit of choosing a vertex in the smaller side of a label class to branch on in the `Search()` function of McSplit. This suggests a modified version of McSplit in which branching may be carried out on vertices of either graph, with the decision of which side to choose made at each search node based on which side of the label class is larger. We show this algorithm, McSplit-2S (with the name signifying "two sided"), in Algorithm 8.

Lines 20 to 35 contain the section of code that is changed from Algorithm 1. (In addition, the function `NewFuture`, which creates the new label classes after a vertex assignment, has been extracted to avoid duplication.) Line 20 chooses which side of the label class $\langle S_G, S_H \rangle$ to branch on. If S_G is no larger than S_H, the algorithm maps a vertex from S_G to each vertex in S_H in turn (lines 25 to 27), just as in the standard McSplit algorithm. Otherwise, the algorithm maps a vertex from S_H to each vertex in S_G in turn (lines 33 to 35).

In addition to McSplit-2S, we implemented the reverse heuristic, which replaces the condition $|G| \leq |H|$ on Line 20 with $|G| > |H|$. This algorithm, which we would expect to perform poorly, is called McSplit-2S'.

Table 4.1 summarises the variants of McSplit that we compare in this section.

Algorithm 8: McSplit-2S: a branch-and-bound algorithm to find a maximum common induced subgraph of two graphs.

```
1  NewFuture(future, v, w)
2  begin
3      future' ← ∅
4      for ⟨S'_G, S'_H⟩ ∈ future do
5          S''_G ← S'_G ∩ N_G(v)
6          S''_H ← S'_H ∩ N_H(w)
7          if S''_G ≠ ∅ and S''_H ≠ ∅ then
8              future' ← future' ∪ {⟨S''_G, S''_H⟩}
9          S''_G ← (S'_G ∩ N̄_G(v))
10         S''_H ← (S'_H ∩ N̄_H(w))
11         if S''_G ≠ ∅ and S''_H ≠ ∅ then
12             future' ← future' ∪ {⟨S''_G, S''_H⟩}
13     return future'

14 Search(future, M)
15 begin
16     if |M| > |incumbent| then incumbent ← M
17     bound ← |M| + Σ_{⟨S_G,S_H⟩∈future} min(|S_G|, |S_H|)
18     if bound ≤ |incumbent| then return
19     ⟨S_G, S_H⟩ ← SelectLabelClass(future)
20     if |S_G| ≤ |S_H| then
21         ▷ Branch as in standard McSplit
22         v ← SelectVertex(S_G)
23         for w ∈ S_H do
24             Search(NewFuture(future, v, w), M ∪ {(v, w)})
25         S'_G ← S_G ∖ {v}
26         future ← future ∖ {⟨S_G, S_H⟩}
27         if S'_G ≠ ∅ then future ← future ∪ {⟨S'_G, S_H⟩}
28     else
29         ▷ Swapped version: branch on a vertex of H
30         w ← SelectVertex(S_H)
31         for v ∈ S_G do
32             Search(NewFuture(future, v, w), M ∪ {(v, w)})
33         S'_H ← S_H ∖ {w}
34         future ← future ∖ {⟨S_G, S_H⟩}
35         if S'_H ≠ ∅ then future ← future ∪ {⟨S_G, S'_H⟩}
36     Search(future, M)

37 McSplit-2S(G, H)
38 begin
39     global incumbent ← ∅
40     Search({⟨V(G), V(H)⟩}, ∅)
41     return incumbent
```

Name	Description	Pseudocode
McSplit	The base McSplit algorithm	Algorithm 1
McSplit-SD	McSplit with G and H swapped if and only if $\left\|\frac{1}{2} - d_G\right\| > \left\|\frac{1}{2} - d_H\right\|$	Algorithm 6
McSplit-SO	McSplit with G and H swapped if and only if $n_G > n_H$	Algorithm 7
McSplit-2S	McSplit with branching on both sides	Algorithm 8
McSplit-2S'	The same as McSplit-2S, but with the condition for branching on a vertex of H rather than G (line 20) negated	-

Table 4.1: Summary of McSplit variants used in this chapter's experiments

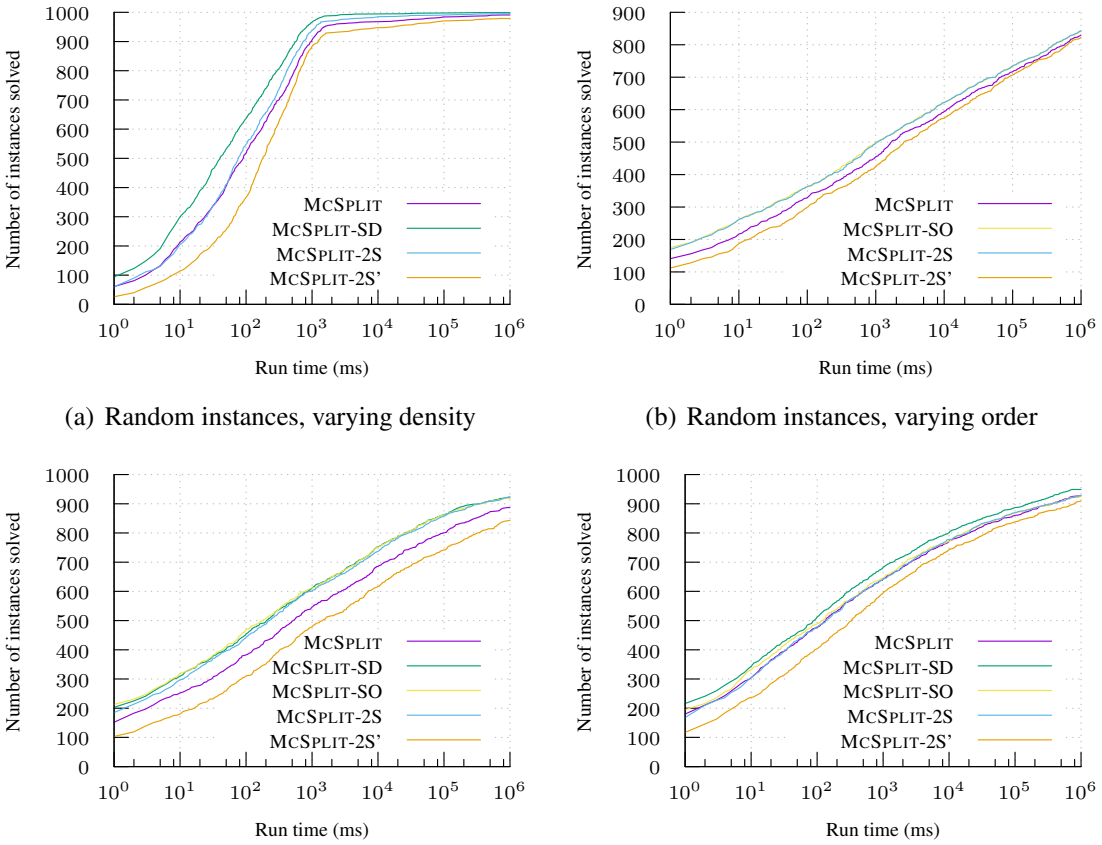
Figure 4.9(a) shows a cumulative plot comparing McSplit, McSplit-2S, McSplit-2S', and McSplit-SD on our family of 24-vertex graphs with varying density. McSplit-SD is fastest of the four algorithms, followed by McSplit-2S.

Figure 4.9(b) compares McSplit, McSplit-2S, McSplit-2S', and McSplit-SO on our family of graphs with similar density but varying vertex count. McSplit-SO and McSplit-2S have almost identical results.

Figure 4.9(c) shows results for a new family of instances, the "mix and match" MCS Plain instances. Each of these instances is generated by selecting two MCS Plain instances at random, and using graph G from the first instance and graph H from the second instance. These compensate for what may be viewed as a weakness of the MCS Plain instances: that the two graphs in each instance are produced using the same generator and the same parameter values. On these "mix and match" instances, McSplit-SD, McSplit-SO, and McSplit-2S have similar performance, and all are clearly faster overall than McSplit. A partial explanation for the similar performance of McSplit-SD and McSplit-SO on these instances is that density and degree are negatively correlated for this set of instances. As a result, the two swapping rules agree — and thus the two algorithms behave identically — on 745 of the 1000 instances.

Figure 4.9(d) introduces another new set of random instances, with both density and order varying. Each graph is a $G(n, p)$ random graph, with n selected at random from the set $\{10, 11, \ldots, 40\}$ and p chosen uniformly at random from the interval $[0, 1]$. Of the five algorithms, McSplit-SD is the clear winner. Unlike in the three previous plots, McSplit-2S does not clearly perform better overall than McSplit; indeed, these two algorithms and McSplit-SO have very similar cumulative curves. The unremarkable performance of McSplit-2S on these instances is something of a puzzle.[2]

[2]My conjecture is that McSplit-2S *would* outperform McSplit if the initial sorting by degree of graphs G and H took into account the densities of both graphs in a way that made sense for the two-sided algorithm. In a small preliminary experiment in which the sorting of graph G depended on the density of graph H and vice versa, McSplit-2S outperformed McSplit, although McSplit-SD remained the fastest solver. I leave



(a) Random instances, varying density

(b) Random instances, varying order

(c) "Mix and match" MCS Plain instances

(d) Random instances, varying density and order

Figure 4.9: Cumulative plots comparing McSplit, McSplit-2S, and the swapping versions of McSplit on four graph classes.

As expected, McSplit-2S' is the worst-performing algorithm on all four instance families shown in Figure 4.9.

4.4.1 Pairs of Graphs With Equal Density and Order

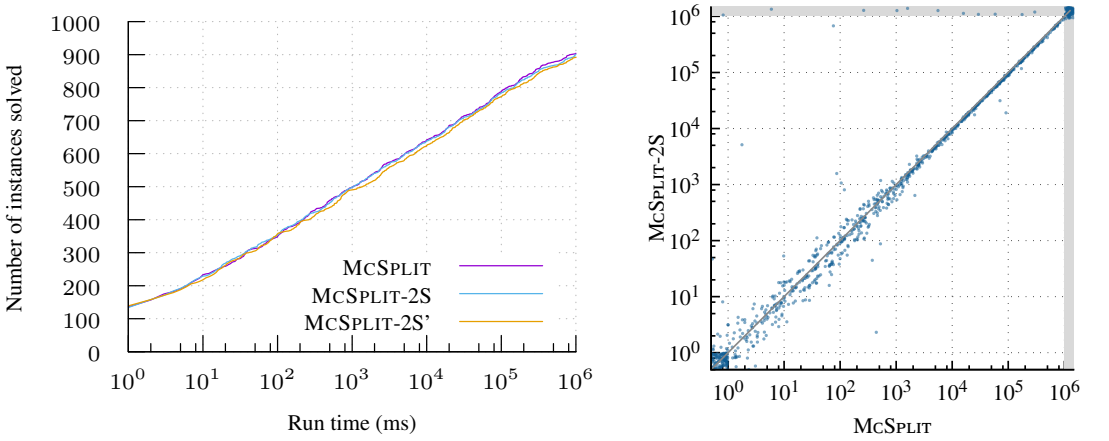
In the previous section, we saw that McSplit-2S is substantially faster overall than McSplit for families of instances in which the pattern and target graph differ in density or order. Yet the results in that section could be viewed as somewhat disappointing; McSplit-2S is outperformed by McSplit-SD on instances whose graphs have different densities, and by McSplit-SO on instances whose graphs have similar density but different numbers of vertices. Are there families of instances for which McSplit-2S clearly outperforms both McSplit-SD and McSplit-SO? To examine this question, this section considers three families of instances in which graphs G and H have the same number of vertices and equal density. For these families of instances, McSplit-SD and McSplit-SO behave identically

further investigation of strategies for sorting by degree in McSplit-2S as future work.

to McSplit. Thus, if McSplit-2S outperforms McSplit on any of these families, it also outperforms both McSplit-SD and McSplit-SO on that family.

The first of our equal-density, equal-order families of instances contains pairs of Erdős-Rényi $G(n, m)$ graphs. For each instance, n is chosen uniformly at random from $\{15, \ldots, 40\}$, then m is chosen uniformly at random from the range of possible edge counts: $\{0, \ldots, n(n-1)/2\}$. Each of the graphs G and H is generated by creating a set of n vertices, then adding a random m-element subset of the possible edges between pairs of vertices.

Figure 4.10 shows a cumulative plot and a scatter plot of run times of McSplit and McSplit-2S. The cumulative plot also shows results for McSplit-2S'. The results are disappointing. All three algorithms have very similar performance overall; worse still, McSplit-2S exceeds the time limit on some instances that are easy for McSplit.[3] A small point in favour of McSplit-2S is that the algorithm is slightly faster than McSplit for many of the hard instances, but the differences in run times are almost imperceptible on the scatter plot.



(a) Cumulative plot

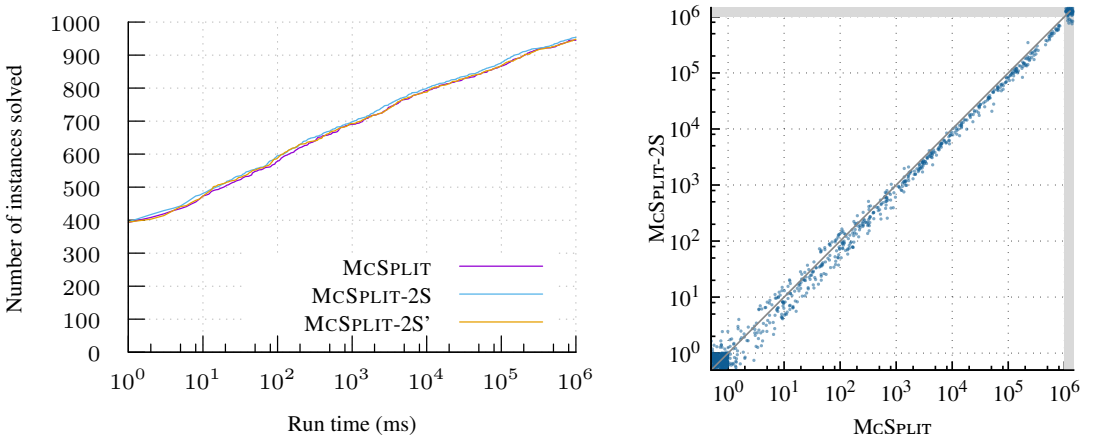
(b) Scatter plot with one point per instance

Figure 4.10: Comparison of McSplit and McSplit-2S on $G(n, m)$ graphs

Our second instance generator creates a pair of graphs using the Barabási-Albert model (Barabási and Albert, 1999), which is very widely cited in the field of network science. We chose this model because, as with the $G(n, m)$ model but unlike many other random graph models such as $G(n, p)$, we can fix the exact density of the generated graphs. The Barabási-Albert model generates graphs with a wide distribution of degrees, allowing us to test our algorithms on random graphs that are quite different in structure from those generated by an Erdős-Rényi model. The Barabási-Albert generator has two parameters, n and m, such that $1 \leq m < n$. The generator works as follows: begin with the star $K_{1,m}$. Extra

[3] All of these outliers are very sparse or very dense. The reason for the difference between the algorithms' run times on these instances remains unclear.

vertices are added one by one until the graph has n vertices. Each time a vertex v is added, m edges are added from v to earlier vertices. These edges are added at random using a "preferential attachment" mechanism: at each step, vertices with more existing edges are proportionately more likely to have a new edge added. We use the implementation of this generator from the Python package NetworkX (Hagberg et al., 2008). For each instance, we use the following ranges of parameter values in order to have wide variation in order and density between instances: n is chosen at random from $\{10, \ldots, 50\}$ and m is chosen at random from $\{2, \ldots, n-1\}$. Within each instance, the two generated graphs use the same parameter values n and m. Figure 4.11 compares McSplit with McSplit-2S on these Barabási-Albert instances. Overall, McSplit-2S has a small speed advantage over McSplit. The scatter plot shows that the two algorithms tend to have very similar run times, with McSplit-2S the slightly faster algorithm for most instances.



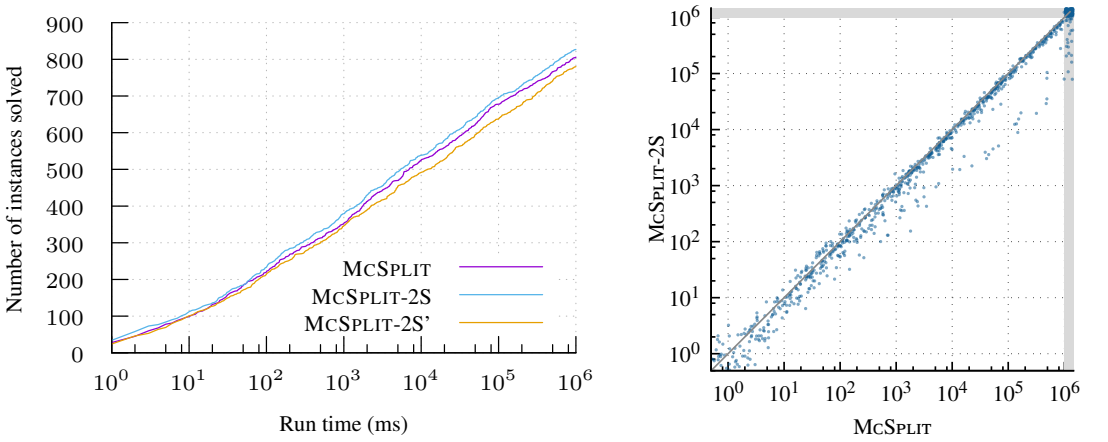
(a) Cumulative plot

(b) Scatter plot with one point per instance

Figure 4.11: Comparison of McSplit and McSplit-2S on Barabási-Albert graphs

The goal of our final instance generator in this section is to produce two graphs with very different characteristics, but with equal orders and densities. Each instance in this family consists of one Barabási-Albert graph and one $G(n, m)$ graph, with the parameters for the latter generator chosen to match the order and density of the Barabási-Albert graph. A "coin flip" is used to determine whether G is the Barabási-Albert graph or the $G(n, m)$ graph. Figure 4.12 shows the results of running our algorithms on these instances. The cumulative curve shows that McSplit-2S is, by a small margin, the fastest algorithm overall, and that McSplit-2S' is the slowest. McSplit-2S and McSplit exceeded the time limit on 105 and 127 instances respectively. The scatter plot shows that the two algorithms often have very different run times, with McSplit outperforming McSplit-2S on several instances.

This section has shown a modest win for McSplit-2S: for our two families of instances using the Barabási-Albert model, McSplit-2S slightly outperforms McSplit — and there-



(a) Cumulative plot

(b) Scatter plot with one point per instance

Figure 4.12: Comparison of McSplit and McSplit-2S on $G(n, m)$ / Barabási-Albert graphs pairs

fore also outperforms McSplit-SD and McSplit-SO.

4.5 Related Work

I have not been able to find any prior work on maximum common subgraph problems in which graphs are swapped using any criterion other than number of vertices. Nor have I found any systematic evaluation of when graphs should be swapped.

A number of papers on maximum common subgraph problems make the assumption, consistent with McSplit-SO, that $n_G \leq n_H$. An early example is McGregor's paper on the maximum common edge subgraph problem (McGregor, 1982), where no rationale for the ordering of graphs is given. Hoffmann et al. (2017) put the smaller graph first because it improves domain filtering in the $k\downarrow$ algorithm (McCreesh, 2022). Dalke and Hastings (2013) place the smallest graph first in FMCS, an algorithm for the maximum common *edge* subgraph problem.[4] A short explanation is given in a code comment (Dalke, 2012). To summarise: this strategy tends to result in fewer subgraphs of the first graph being visited. This explanation is very important in FMCS, which makes a call to a subgraph isomorphism algorithm for every visited subgraph, but is not directly applicable to McSplit.

I have not found any prior maximum common subgraph algorithm that branches on both graphs, as McSplit-2S does. However, it is common in constraint programming to combine multiple models for a problem, with "channeling" constraints to ensure consistency between these models (Cheng et al., 1999). In effect, the data structures of McSplit give us access to

[4]More precisely, the algorithm chooses a graph whose largest connected component is as small as possible, as FMCS searches only for connected subgraphs.

a second model — the "dual" model with the roles of variables and values reversed (Geelen, 1992) — without any need to explicitly store additional domains or constraints, and therefore with no overhead in memory use or run time.

4.6 Conclusion

In this chapter, we have introduced two rules to determine whether to swap graphs G and H when calling McSplit: one using the graphs' densities, and the other using their node counts. We have seen that both rules are very effective in reducing the run time of McSplit both on random graphs and on "mix and match" instances created by choosing random pairs of graphs from the MCS Plain instances. Using insights from the behaviour of these versions of McSplit, we have introduced McSplit-2S, a version of McSplit that branches on vertices of both graphs. This achieves similar performance to McSplit-SD and McSplit-SO on several graph families, and there exist families of graphs on which it outperforms both algorithms.

There are many potential avenues for future work. We could investigate whether swapping rules are useful for other maximum common subgraph algorithms, including algorithms for maximum common *edge* subgraph which are beyond the scope of this dissertation. We could also explore rules for swapping graphs that take into account both density and order, perhaps along with additional characteristics of the two graphs.

With McSplit-2S, we have only begun to explore the possibilities of branching on both graphs. We could investigate replacing our simple "branch on the larger side" heuristic with more sophisticated rules for choosing which side to branch on at each search node. We could also combine the two-sided branching of McSplit-2S with the initial swapping rules of McSplit-SD or McSplit-SO.

Towards the end of next chapter, we will return to the MCIS problem with specialised algorithm for large, sparse graphs. All of the instances in that chapter have $|V(G)| < |V(H)|$, so we will implicitly use the McSplit-SO swapping heuristic. First, we turn to our second problem: induced subgraph isomorphism.

Chapter 5

McSplit-SI: An Algorithm for the Induced Subgraph Isomorphism Problem

5.1 Introduction

In the induced subgraph isomorphism problem (ISIP), we seek an induced copy of *pattern* graph G in *target* graph H. This is closely related to the maximum common induced subgraph problem: we can view ISIP as a decision version of MCIS in which the common subgraph is required to contain all of pattern graph's vertices. The McSplit algorithm may be trivially modified to solve the induced subgraph isomorphism problem: rather than calculating an upper bound at each search node, simply backtrack when the G-set of any label class is larger than the corresponding H-set.

McSplit is well suited to the small (tens of vertices), relatively dense pattern and target graphs that are typical of maximum common subgraph instances, but it has two disadvantages for large (hundreds or thousands of vertices), sparse graphs that appear in benchmark instances for subgraph isomorphism. The first disadvantage relates to space: $b(n_G^2 + n_H^2)$ space is needed to store the adjacency matrices, where b is the memory size of a Boolean variable and n_G and n_H are the orders of the pattern and target graphs.[1] The second disadvantage relates to time: during the partitioning step, McSplit iterates over the vertices in each label class, which often requires checking close to $n_G + n_H$ adjacency-matrix elements each time the partitioning procedure is carried out.

[1] We could switch to a more space-efficient representation such as hash sets of neighbours which would permit amortised constant time adjacency tests in the algorithm's partitioning step, but this would slow down the algorithm significantly.

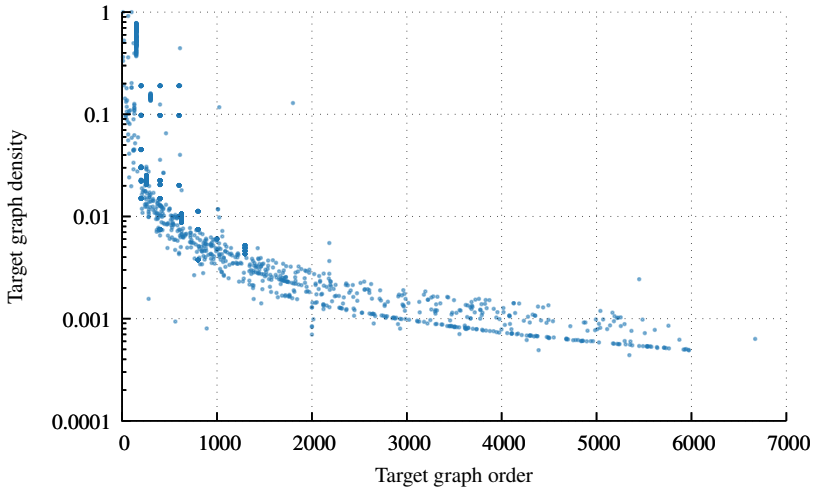


Figure 5.1: Number of vertices and density (log scale) for each target graph in the benchmark set of $14,621$ subgraph isomorphism decision instances.

As Figure 5.1 shows, the target graphs of the benchmark instances that we will use in Section 5.8.2 are predominantly sparse, and many have thousands of vertices. Furthermore, over three quarters of the *pattern* graphs in this benchmark set have density less than 0.01. Thus, it would greatly improve our McSplit algorithm for subgraph isomorphism on these and similar instances if we could reduce the time complexity of the partitioning step from $O(n_G + n_H)$ to $O(|N(v)| + |N(w)|)$, where (v, w) is the most-recently made mapping of a pattern vertex to a target vertex. In this chapter we introduce this improved algorithm, which we call McSplit-SI.

McSplit-SI explores the same search tree as the first decision problem of McSplit↓ (where the target is to find a copy of all of G in H). Its key differences from McSplit are (1) the use of adjacency lists rather than adjacency matrices to represent graphs; (2) a backtrackable, doubly linked list of label class objects, which replaces McSplit's LC-arrays; and (3) algorithms for partitioning and backtracking that are more intricate than the corresponding steps in McSplit, and are much faster for sparse graphs.

Section 5.2 introduces the data structures of McSplit-SI, Section 5.3 describes the algorithm itself, and Section 5.4 discusses variable and value ordering heuristics. Section 5.5 introduces an alternative version of McSplit-SI using linked lists of vertices, and Section 5.6 introduces another alternative version that is well-suited to dense graphs. Section 5.7 shows that McSplit-SI achieves generalised arc consistency on the all-different constraint without the need for a complex filtering algorithm. Section 5.8 presents a detailed set of experiments comparing McSplit-SI with other solvers. Section 5.9 shows how, with only a few changes, we can use a version of McSplit-SI to solve MCIS on sparse graphs. Section 5.10 concludes.

5.2 The Data Structures of McSplit-SI

In the MCIS version of McSplit described in Chapter 3, the label class objects at each level of the search tree are stored contiguously in an array (the LC-array), and each object representing a label class $\langle S_G, S_H \rangle$ requires only four indices or pointers—to the start and end of the array slices that contain S_G and S_H. To enable partitioning in $O(|N_G(v)| + |N_H(w)|)$ time, McSplit-SI requires a more elaborate label-class object, and stores these objects in a doubly linked list which is modified when partitioning domains and restored on backtracking.

Table 5.1 lists the member variables of a label class object. The first two members are $prev$ and $next$ pointers, which allow the set of label classes to be jointed together as a doubly linked list. These pointers are useful not only for iterating over the list but also for restoring deleted elements when backtracking, as we will discuss shortly.

Name	Type	Description
$prev$	Pointer to Label Class	The previous label class in the doubly linked list
$next$	Pointer to Label Class	The next label class in the doubly linked list
$start_G$	Pointer to Integer	Pointer to the first vertex of set S_G
end_G	Pointer to Integer	Pointer to one element past the last vertex of set S_G
$start_H$	Pointer to Integer	Pointer to the first vertex of set S_H
end_H	Pointer to Integer	Pointer to one element past the last vertex of set S_H
$active$	Boolean	Is this label class in the doubly linked list?
$splitting$	Boolean	Is this label class being split?

Table 5.1: The member variables of McSplit-SI label class object

The next four members of the object play the same role as the four indices used in McSplit's simpler label class object: they point to the ranges in the permutations of $V(G)$ and $V(H)$ that contain S_G and S_H.

Finally, we have two Boolean flags, $active$ and $splitting$. The first flag records whether the label class object is currently in the list of label classes. (Inactive label classes have been deleted, but are maintained in memory so that they can be restored when backtracking.) The second flag is used temporarily during the partitioning step to record whether the label class has been partitioned.

The collection of data structures used by McSplit-SI to represent the set of active label classes has three components. The first is the doubly linked list of label class objects that we have described; we call this $\mathcal{LC}$. The second component is the pair of arrays that store partitions of $V(G)$ and $V(H)$ that are pointed to by each label class object (McSplit similarly uses two arrays for this purpose). We call these arrays A_G and A_H.

The final component of our collection of data structures contains the arrays P_G and P_H. These contain an object for each vertex v of G and H respectively. Each object comprises

two pointers. The first pointer of $P_G[v]$ points to the location in A_G at which v appears, and the second points to the label class containing v (or is a null pointer if v is not currently in a label class). Similarly, the pointers of $P_H[w]$ point to the position in A_H at which w appears and the label class containing w. The arrays P_G and P_H allows us to perform constant-time manipulations to label classes given only a vertex; it is these arrays that enable MCSPLIT-SI to carry out the partitioning step without iterating over the vertices in each label class.

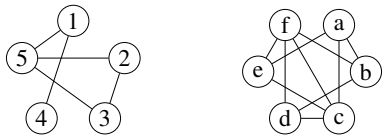


Figure 5.2: Example graphs G and H

Figure 5.2 reproduces the graphs from Chapter 3 which we will use again as the instance for this chapter's running example. Figure 5.3 shows the data structures of MCSPLIT-SI after making the assignment $(1, a)$. In the middle row of the figure we have the doubly linked list $\mathcal{LC}$, and immediately above and below this are the A_G and A_H arrays. At the top and bottom are the arrays P_G and P_H. The latter two arrays are indexed by the vertex sets of the two graphs; although sets $\{1, \ldots, 5\}$ and $\{a, \ldots, f\}$ are used in this example, our implementation requires these sets to be $\{0, \ldots, n_G - 1\}$ and $\{0, \ldots, n_H - 1\}$ so that constant-time array operations can be performed in the usual way.

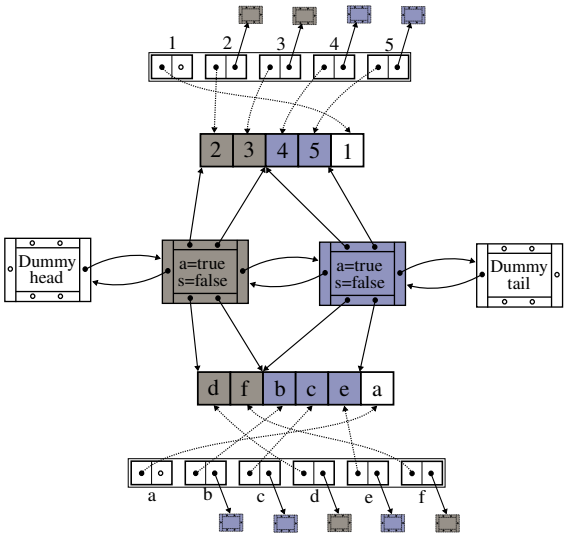


Figure 5.3: The data structures of MCSPLIT-SI after assigning vertex 1 to vertex a. Circles represent pointers; hollow circles are null pointers. The middle row shows the list of label classes. Immediately above and below this are the permutations of $V(G)$ and $V(H)$, stored as arrays A_G and A_H. The top and bottom rows show arrays P_G and P_H. Each element of these arrays corresponds to a vertex v of G or H, and points to the position of v in A_G or A_H and to the label class containing v. To reduce clutter, the label class pointers are shown pointing to a rectangle of the same colour as the label class.

5.3 The McSplit-SI Algorithm

We begin our detailed look at McSplit-SI with Algorithm 9, which presents the algorithm's overall structure. For simplicity, the algorithm as presented returns only $true$ or $false$; it is of course straightforward to also return the mapping found for feasible instances if required.

The entry point is line 17. If G has more vertices than H, the instance is trivially unsatisfiable. Otherwise, the global data structures are set up with a single label class, and the main recursive function `Search` is called with an empty mapping.

The `Search` function works as follows. Line 3 returns $true$–signifying that the induced subgraph isomorphism instance is satisfiable—if a mapping containing all vertices of the pattern graph has been found. Otherwise, the algorithm selects a label class $\langle S_G, S_H \rangle$, and a vertex from S_G on which to branch. (We will discuss variable selection heuristics in Section 5.4.) We then iterate over the vertices w in S_H, attempting to map v to w.

Algorithm 9: McSplit-SI

```
1  Search(M)
2  begin
3      if |M| = |V(G)| then return true
4      ⟨S_G, S_H⟩ ← SelectLabelClass()
5      v ← SelectVertex(S_G)
6      for w ∈ S_H do
7          Assign(v, w)  ▷ Algorithm 10
8          (splits, deletions, failed) ← Filter(v, w)  ▷ Algorithm 11
9          if failed then
10             success ← false
11         else
12             success ← Search(M ∪ {(v, w)})
13         Unfilter(splits, deletions)  ▷ Algorithm 14
14         Unassign(v, w, ⟨S_G, S_H⟩)  ▷ Algorithm 10
15         if success then return true
16     return false

17 McSplitSI(G, H)
18 begin
19     if |V(G)| > |V(H)| then return false
20     Initialise global data structure with the label class {⟨V(G), V(H)⟩}
21     return Search(∅)
```

Within this loop are two pairs of functions: `Assign` / `Unassign` and `Filter` / `Unfilter`. In each pair, the second function reverses the action of the first function on the global data structure of label classes.

The first pair of functions is shown in Algorithm 10. Both of these run in constant time. The function `Assign`(v, w) updates the label classes to reflect the assignment of v in the

Algorithm 10: The `Assign` and `Unassign` functions of MCSPLIT-SI

```
1  Assign(v, w)
2  begin
3      LC ← P_G[v].labelClass
4      ▷ delete v from LC
5      u ← the vertex in A_G whose address is one element before LC.end_G
6      Swap v with u in A_G, using the address of v in P_G[v].vertexPtr
7      Swap P_G[v].vertexPtr with P_G[u].vertexPtr
8      Decrement LC.end_G
9      P_G[v].labelClass ← null
10     ▷ delete w from LC
11     u ← the vertex in A_H whose address is one element before LC.end_H
12     Swap w with u in A_H, using the address of w in P_H[w].vertexPtr
13     Swap P_H[w].vertexPtr with P_H[u].vertexPtr
14     Decrement LC.end_H
15     P_H[w].labelClass ← null
16     if LC.start_G = LC.end_G then
17         ▷ Delete LC from the doubly linked list of label classes
18         LC.prev.next ← LC.next
19         LC.next.prev ← LC.prev

20 Unassign(v, w, LC)
21 begin
22     if LC.start_G = LC.end_G then
23         ▷ Restore LC to the doubly linked list of label classes
24         LC.prev.next ← LC
25         LC.next.prev ← LC
26     ▷ restore v and w to LC
27     P_G[v].labelClass ← LC
28     P_H[w].labelClass ← LC
29     Increment LC.end_G
30     Increment LC.end_H
```

pattern graph to w in the target graph. It does this by swapping v and w to the end of their label class in A_G and A_H, then decrementing the end pointers of the label class. If no vertices of the pattern graph remain in the label class, the label-class object is deleted from the linked list $\mathcal{LC}$.

To maintain the invariants of the P_G and P_H arrays, we set the label-class pointers of $P_G[v]$ and $P_H[w]$ to null since these vertices are no longer in a label class, and update the vertex pointers for v, w, and the vertices with which these were swapped to point to these vertices' new positions in the A_G and A_H arrays. Figure 5.4 shows the data structures after the assignment of 2 to d in our example.

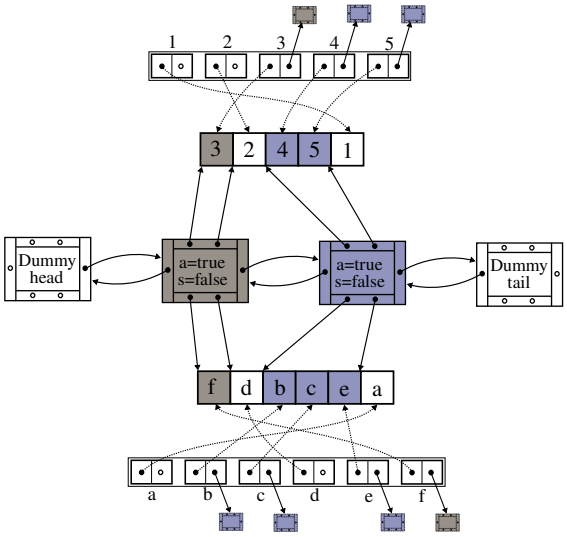


Figure 5.4: The data structures after mapping vertex 2 to vertex d.

The partitioning algorithm After assigning a vertex of G to a vertex of H, the algorithm proceeds to the partitioning step—the `Filter` function of Algorithm 11. Figure 5.5 shows, in our running example, the data structures after mapping 2 to d and performing all but the last two lines of the partitioning step.

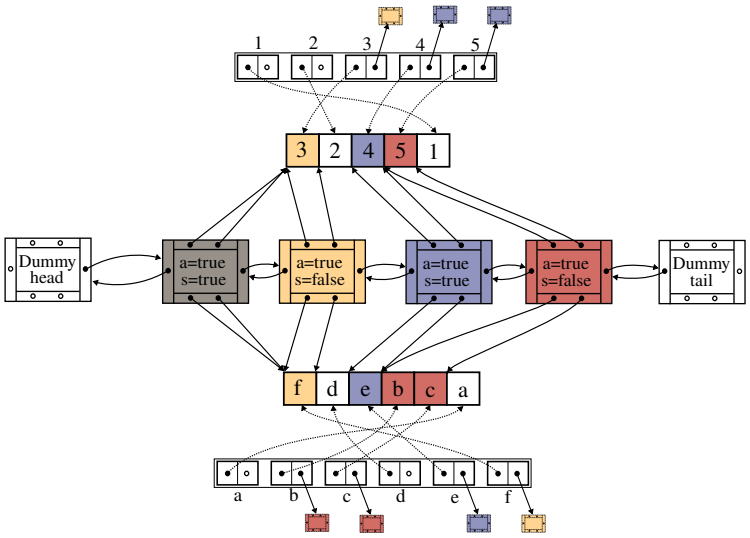


Figure 5.5: The data structures after partitioning

The list $\mathcal{LC}$ of label classes is modified in place. Each label class object that contained the sets $\langle S_G, S_H \rangle$ prior to the partitioning process contains $\langle S_G \setminus N_G(v), S_H \setminus N_H(w) \rangle$ at the end of the partitioning process. If either $S_G \cap N_G(v)$ or $S_H \cap N_H(w)$ is non-empty, the partitioning process creates a new label class object $\langle S_G \cap N_G(v), S_H \cap N_H(w) \rangle$ which is positioned in the doubly linked list immediately after the original label class. To carry out

the process, the loops beginning at line 5 and line 17 in Algorithm 11 iterate over $N_G(v)$ then $N_H(w)$, creating new label classes as required (Algorithm 12).

After partitioning, the loop beginning at line 33 of Algorithm 11 returns early and indicates failure if the G set of any label class contains more vertices than the H set. This allows the `Search` function to backtrack.

Cleanup In Figure 5.5, we see that the first label class no longer contains any elements of S_G. As a result, we must delete this label class object from the linked list $\mathcal{LC}$. We remove the item from the list in the usual way, by causing the previous and next elements of the list to point to one another. Unlike a normal linked-list deletion, however, we do not garbage-collect the removed label class. Instead, we leave the label class in memory, and append a pointer to it to a list of deleted label classes. The label classes on this list will be restored on backtracking. Algorithm 13 shows the deletion process in full.

Return values of the Filter function After calling the `DoDeletions` function, the `Filter` function returns on line 39 of Algorithm 11. Three values are returned. The lists of split label classes and deleted label classes will be used to restore the data structure when backtracking. The final returned value is a Boolean flag that indicates that the filtering step completed successfully without finding a reason to backtrack immediately.

A small implementation detail that avoids the need to allocate memory for arrays on each call to `Filter`: rather than storing the returned lists as linear lists of pointers, we store two additional pointers within each label class so that the lists can be stored as (intrusive) singly linked lists without any need to allocate additional arrays. In the case of the *splits* list, our implementation returns a list of the newly created label classes rather than the label classes from which they were split.

Reversing the effects of the Filter function After recursively calling `Search` on line 12 of Algorithm 9, the algorithm calls `Unfilter` (Algorithm 14) to reverse the effects of the `Filter` function. First, each deleted label class is restored to its original position in the doubly linked list. We use the "dancing links" method introduced by Hitotumatu and Noshita (1979) and named by Knuth (2020): since we have kept each deleted label class in memory and since its *prev* and *next* pointers already point to the position in the linked list from which it was removed, the operation of re-inserting a label class is very simple and fast.

Next, we merge partitioned label classes. Just as in McSplit, there is no need to reorder the S_G and S_H permutations when backtracking. It is necessary, however, to restore the label class pointers in P_G and P_H (line 10 and line 11 of Algorithm 14) of those vertices that were removed from the original label class.

Algorithm 11: The `Filter` function

```
1  Filter(v, w)
2  begin
3      splits ← [] ▷ Initialise array of pointers to split label classes
4      ▷ For each neighbour of v that is in a label class, move v into a new label class
5      for u ∈ N_G(v) do
6          LC ← P_G[u].labelClass
7          if LC = null then
8              continue ▷ u is already in M, and therefore not in any label class
9          if LC.splitting = false then
10             LC.splitting ← true
11             CreateLabelClassAfter(LC) ▷ Algorithm 12
12             Append to splits a pointer to LC
13         Swap u to the end of LC in A_G, updating the two relevant vertexPtr members in P_G
14         P_G[u].labelClass ← LC.next
15         Decrement LC.end_G and LC.next.start_G ▷ Move u from LC to LC.next
16     ▷ For each neighbour of w that is in a label class, move w into a new label class
17     for u ∈ N_H(w) do
18         LC ← P_H[u].labelClass
19         if LC = null then
20             continue ▷ u is already in M, and therefore not in any label class
21         if LC.active = false then
22             ▷ The label class containing u was deleted at a shallower level of the search tree
23             continue
24         if LC.splitting = false then
25             LC.splitting ← true
26             CreateLabelClassAfter(LC) ▷ Algorithm 12
27             Append to splits a pointer to LC
28         Swap u to the end of LC in A_H, updating the two relevant vertexPtr members in P_H
29         P_H[u].labelClass ← LC.next
30         Decrement LC.end_H and LC.next.start_H ▷ Move u from LC to LC.next
31     for LC ∈ splits do
32         LC.splitting ← false
33     for LC ∈ splits do
34         if LC.end_G − LC.start_G > LC.end_H − LC.start_H then
35             return (splits, [], true)
36         if LC.next.end_G − LC.next.start_G > LC.next.end_H − LC.next.start_H then
37             return (splits, [], true)
38     deletions ← DoDeletions(splits) ▷ Algorithm 13
39     return (splits, deletions, false)
```

Algorithm 12: The `CreateLabelClassAfter` function

```
1  CreateLabelClassAfter(LC)
2  begin
3      Insert a new label class object LC' after LC in the doubly linked list
4      LC'.start_G ← LC.end_G
5      LC'.end_G ← LC.end_G
6      LC'.start_H ← LC.end_H
7      LC'.end_H ← LC.end_H
8      LC'.active ← true
9      LC'.splitting ← false
```

Algorithm 13: The `DoDeletions` function

```
1  DoDeletions(splits)
2  begin
3      deletions ← [] ▷ Initialise list of pointers to deleted label classes
4      for LC ∈ splits do
5          if LC.start_G = LC.end_G then
6              DeleteLabelClass(LC)
7              Append LC to deletions
8          if LC.next.start_G = LC.next.end_G then
9              DeleteLabelClass(LC.next)
10             Append LC.next to deletions
       return deletions

11 DeleteLabelClass(LC)
12 begin
13     LC.prev.next ← LC.next
14     LC.next.prev ← LC.prev
15     LC.active ← false
```

Algorithm 14: The `Unfilter` function

```
1  Unfilter(splits, deletions)
2  begin
3      for LC ∈ deletions, in reverse order do
4          ▷ Restore LC
5          LC.prev.next ← LC
6          LC.next.prev ← LC
7          LC.active ← true
8      for LC ∈ splits, in reverse order do
9          ▷ Merge LC.next into LC
10         for u in the set S_G of LC.next do P_G[u].labelClass ← LC
11         for u in the set S_H of LC.next do P_H[u].labelClass ← LC
12         LC.end_G ← LC.next.end_G
13         LC.end_H ← LC.next.end_H
14         Remove LC.next from the doubly linked list of label classes
```

5.3.1 Finding all Solutions

The McSplit-SI algorithm that we have described so far determines whether there exists an isomorphism from the pattern graph to an induced subgraph of the target graph. We can trivially modify the program to solve the enumeration problem of counting *all* such mappings. The only change required is to increment a global counter of solutions on line 3 of Algorithm 9 rather than returning $true$.

5.3.2 Optimisations

This section describes two optimisations in our implementation of McSplit-SI.

Lazy partitioning The partitioning loops beginning on line 5 and line 17 of Algorithm 11 are critical to the performance of the algorithm; the Linux `perf` utility shows that the program typically spends more than half of its execution time on these two loops. Therefore, it is helpful to do as little work in these loops as possible.

To reduce the work carried out in these loops, our implementation does not create new label classes or swap vertices during the loops; these tasks are delayed until just before line 38. Often, the algorithm is able to determine that the subproblem is infeasible during the loop beginning on line line 33, and therefore does not need to create the new label classes or move vertices at all.

Memory allocation Our implementation aims to make as few calls to the system memory allocator as possible when creating new label class objects. We have implemented a very simple allocator, as follows. There is a *free list* of label class objects, which is a singly linked list of objects that are not currently in use. When a new label class is required, the first element of the free list is used. If the free list is empty, we allocate a contiguous pool of 100 label class objects using the system allocator (in order to improve locality of reference), and add each of these to the free list. A label class is deleted simply by adding it to the head of the free list. The pools of objects are released by the system allocator only when the algorithm terminates.

Using this approach, the partitioning step does not need to make any dynamic memory allocations, except on rare occasions when the free list is exhausted. This approach to memory allocation typically reduces run time by around 10% in comparison to use of C++'s `new` and `delete` keywords for each allocation and deallocation.

5.3.3 Variants: Vertex and Edge Labels and Directed Graphs

MCSPLIT-SI can be straightforwardly extended to support vertex labels (and loops, by treating a loop as a label modifier) using the same method as in MCSPLIT: for each vertex label l that appears in the pattern graph, we initially create a label class that contains all vertices in the pattern and target graphs that have label l. If any of these initial label classes contains more pattern vertices than target vertices, the algorithm can report failure immediately without calling `Search()`.

Our implementation supports directed graphs using a straightforward approach. For each vertex, we store lists of in-edges and out-edges separately. The call to `Filter()` in Algorithm 9 is then replaced with two calls: the first one for in-edges, the second for out-edges. Correspondingly, two calls are made to `Unfilter()` in reverse order of the calls to `Filter()`.

The current implementation of MCSPLIT-SI does not support edge labels, but it would be possible to do so with a fairly simple modification to the algorithm. Moreover, this modification does not change the $O(|N_G(v)| + |N_H(w)|)$ time complexity of the filtering step. We assume that the labels belong to some ordered set such as the integers. The key is to sort the adjacency lists of G and H according to edge label before running the MCSPLIT-SI algorithm: each adjacency list $N(v)$ is sorted such that if the label on edge $\{v, u\}$ is less than the label on $\{v, u'\}$ then u appears before u' in the adjacency lists. When partitioning is carried out, the vertices in each new label class are therefore ordered by edge label. By simultaneously traversing the S_G and S_H sets of a new label class from left to right, we can subdivide the label class according to edge labels. This is similar to how edge labels are handled in MCSPLIT; unlike MCSPLIT, however, the data structures of MCSPLIT-SI allow us to avoid sorting in the `Filter()` function.

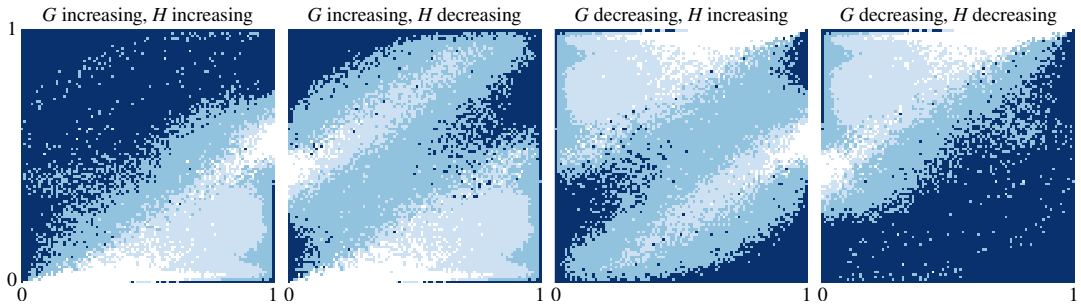
5.4 Variable and Value Ordering Heuristics

The speed of MCSPLIT-SI, like that of other subgraph isomorphism solvers (Bonnici and Giugno, 2017; McCreesh et al., 2018), is greatly affected by the order in which pattern and target vertices are chosen during search. Using terminology from constraint programming, we call the strategy used to select a pattern vertex on line 5 of Algorithm 9 the *variable ordering heuristic*, and the strategy used to decide the iteration order over target vertices on line 6 the *value ordering heuristic*.

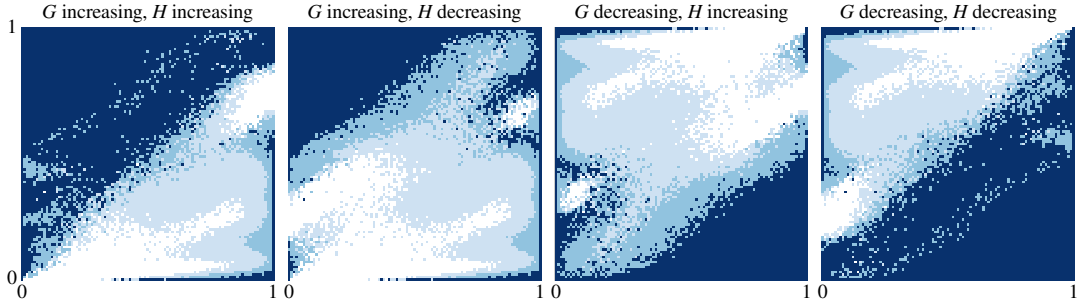
Our variable ordering heuristic is *smallest domain first*, which is also used by the Glasgow Subgraph Solver. In MCSPLIT-SI, the smallest domain first heuristic requires that we choose a vertex v from a label class $\langle S_G, S_H \rangle$ with as small a set S_H as possible. Since

we cannot have a solution to the induced subgraph isomorphism problem if $|S_G| > |S_H|$ for any label class, this heuristic is equivalent to the McSplit heuristic of minimising $\max(|S_G|, |S_H|)$ that we used in Chapter 3.

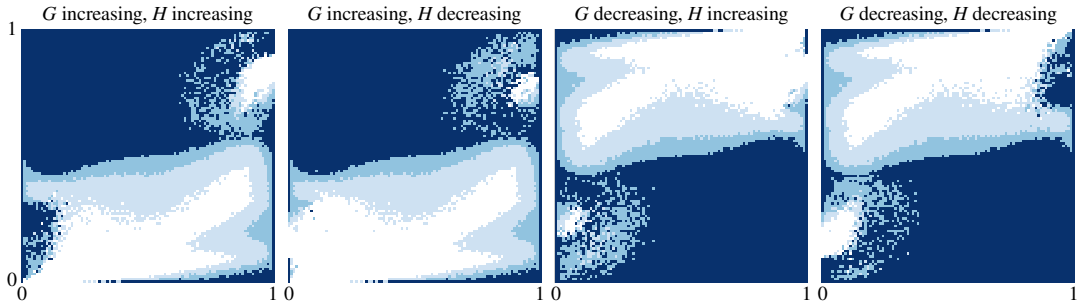
As with the McSplit algorithm for MCIS (see Section 3.4), McSplit-SI sorts the vertices of both graphs according to degree before search. We now repeat the exercise in Section 3.4—this time for induced subgraph isomorphism—of varying the pattern and target graph densities and finding out which ordering strategy works best for a given pair of densities.



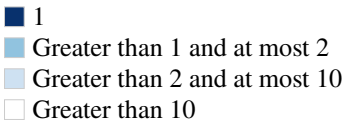
(a) Pattern order 8



(b) Pattern order 12



(c) Pattern order 16



(d) Colour key (search effort divided by search effort of best strategy)

Figure 5.6: Heatmaps of McSplit-SI search effort compared to the best strategy, varying pattern density (horizontal axis) and target density (vertical axis)

As before, we generated undirected graphs using the $G(n, p)$ model, varying pattern and target densities were varied from 0 to 1 in steps of 0.01. We ran the experiment three times, with 8-, 12- and 16-vertex pattern graphs; in each case, the target graph had 80 vertices. Search effort was measured by counting the number of calls to `Search`. Each run was repeated 32 times, and an average (mean) number of calls was computed. A node limit of 100 million was set; runs with more than 100 million recursive calls were counted as requiring 100 million calls. (Similar heatmaps for the Glasgow Subgraph Solver comparing the first and last of the strategies shown were presented by McCreesh et al. (2018). Moreover, that paper shows that there is a strong relationship between the optimal strategy and the location of the phase transition between unsatisfiable and satisfiable instances. The results of McCreesh et al. show similar patterns to those shown here for MCSPLIT-SI.)

The heatmaps in Figure 5.6 show which pairing of pattern and target graph orders was best for each pattern density / target density pair. The first subplot of Figure 5.6(a) shows that the "G decreasing, H decreasing" strategy is preferable if the density of the target graph is less than that of the pattern graph. The two middle subplots are predominantly lighter blues and white, implying that using opposite orders for the pattern and target graphs is only of benefit in small parts of the parameter space.

Figure 5.6(c) — with pattern graphs of order 16 — has a somewhat different pattern. Now, the "G increasing" order appears to be preferable in most cases where the target density exceeds 0.5. This is equivalent to preferring pattern vertices that are involved in as few constraints as possible.

As we saw for MCIS, we can see that there is a complex relationship between the densities of the graphs and the optimal strategy for ordering vertices. For the experimental evaluation in Section 5.8, I tried two strategies which broadly extrapolate from the results in Figure 5.6(a) and Figure 5.6(c). The first strategy is to use "G increasing, H increasing" if the target density is greater than the pattern density, and "G decreasing, H decreasing" otherwise. The second strategy is to use "G increasing, H increasing" if the target density is greater than 0.5, and "G decreasing, H decreasing" otherwise. The latter of these strategies—which matches the MCSPLIT strategy—performed much better, and therefore the results of that strategy are reported in Section 5.8.

5.4.1 Static Variable Ordering Heuristics

So far in this section, we have considered only *dynamic* variable ordering heuristics, in which the choice of v is made during search and depends primarily on the current sizes of label classes. The RI solver does not use the concept of domains, and therefore by necessity takes a simpler approach: it uses a static variable order that is determined before search

(Bonnici and Giugno, 2017). We will use this as an alternative variable ordering strategy for MCSPLIT-SI and—as we will see in Section 5.8—it is often a very successful strategy in practice. Therefore we will now describe the RI strategy in detail.

RI generates its variable order before search begins by adding the vertices of graph G to a list one by one, maintaining the following three sets. Set X_1 contains pattern-graph vertices already in the order. Set X_2 contains vertices not in X_1 but adjacent to at least one element of X_1. Finally, X_3, contains all remaining pattern vertices. At each step, the next element of the order is chosen by selecting a vertex with as many neighbours in X_1 as possible, tie-breaking by the number of neighbours in X_2, then finally tie-breaking by the number of neighbours in X_3.

In the common case where graphs G and H are sparse, this variable ordering strategy can be viewed intuitively as approximating the smallest domain strategy of MCSPLIT-SI with tie breaking by degree. To see this, note that the vertices prioritised by RI—those adjacent to many assigned vertices—will tend to have many values removed from their domains as a result of those adjacencies. The tie breaking (approximately) by degree follows from the fact that RI prioritises vertices with many neighbours in each set X_i.

Yet clearly there is more to the RI heuristic that simply tie breaking by degree, since the heuristic distinguishes between vertices that are neighbours of some assigned vertex (X_2) and vertices that are not (X_3). This distinction may explain the success of the RI heuristic in our experimental evaluation (Section 5.8). It would be interesting future work to explore whether the same distinction could be used to improved dynamic variable ordering heuristics.

I have implemented the RI variable ordering heuristic in a variant of MCSPLIT-SI which I will refer to as MCSPLIT-SI-static. I have modified the RI heuristic in one respect: if graph H has density greater than ½, the order is computed on the complement of G rather than on G itself.

5.5 MCSPLIT-SI-LL: A Version Using Linked Lists of Vertices

The data structure used by MCSPLIT-SI to represent the two lists of vertices within each label class enables fast partitioning, but has the disadvantage that the vertices in each list are not kept in any particular order. This has two negative consequences. First, when selecting a label class according to the smallest domain first heuristic with tie breaking on degree, we need to scan the pattern-graph vertices within each label class of minimum size to find a vertex of optimal degree. Second, we face a time-space tradeoff when iterating over the target-graph vertices in degree order on line 6 of Algorithm 9: we can choose either to

copy the list of vertices representing S_H and then sort this copy, or to scan the list S_H on each iteration for the next-best element. The first of these options — which we use in our implementation — requires $O(|S_H| \log |S_H|)$ time and $|S_H|$ words of additional space; the second requires $O(|S_H|^2)$ time and no additional space.

These disadvantages tend to be small in practice, as the label class chosen for iteration tends to be small except at the root of the search tree. Nevertheless, there exist pathological instances in which the smallest label classes are large even deep in the search tree. For example, if the pattern and target graphs have no edges then there is only one label class at every depth of recursion. Therefore, it would be useful to have a version of MCSPLIT-SI that keeps the vertices sorted in each label class without increasing the time complexity of the partitioning procedure. This section describes such a version, which we call MCSPLIT-SI-LL because it stores vertices in linked lists.

The data structures of MCSPLIT-SI-LL differ from those of MCSPLIT-SI as follows. MCSPLIT-SI-LL does not use the arrays A_G and A_H; rather, each element of P_G and P_H has forward and backward pointers that enable the vertices within a label class to be joined together as a doubly linked list. In each label class, $start_G$, end_G, $start_H$, and end_H point to the first and last nodes of these linked lists in P_G and P_H. Each label class has two additional integer members that store the lengths of the two lists of vertices.

Since the linked-list nodes are embedded in an array, we can delete a given vertex from its list in constant time. We have the usual time complexity guarantees of doubly linked lists for other operations: constant-time append and insert, and linear-time iteration.

Figure 5.7 shows the data structures of MCSPLIT-SI-LL after assigning vertex 1 to vertex a; this corresponds to Figure 5.3 which shows the same state using the data structures of MCSPLIT-SI.

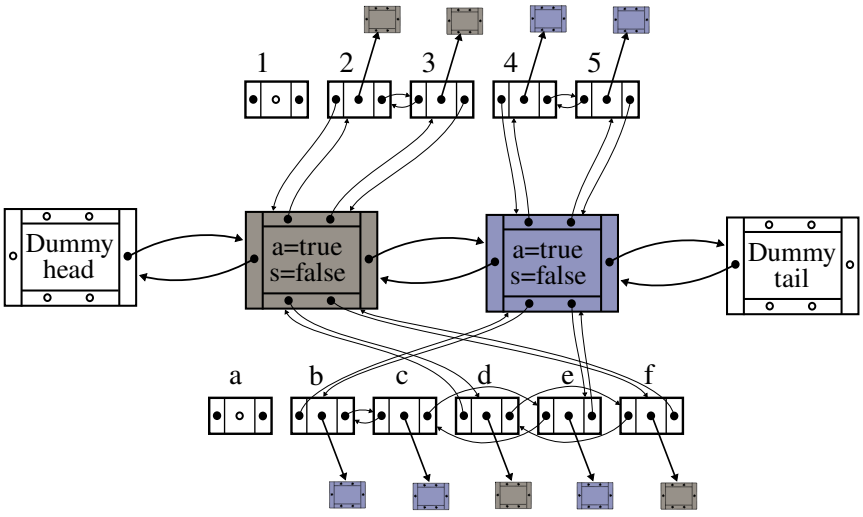


Figure 5.7: The data structures of MCSPLIT-SI-LL. Each of the two sets of vertices in a label class is represented as a doubly linked list.

Before the first call to `Search()`, the two linked lists of vertices in each label class are sorted in order of vertex degree. We also sort each adjacency list of graphs G and H in order of vertex degree.

We require the order of each linked list of vertices to be maintained by the `Filter` and `Unfilter` functions. The first of these is straightforward: when iterating over an adjacency list in the `Filter` function, we delete each vertex v from its label class and append it to the new label class. Because the adjacency lists have been sorted by degree before search, vertices are appended to each new label class in order of degree.

The `Unfilter` function requires more work in MCSPLIT-SI-LL than in MCSPLIT-SI, since when merging label classes we must ensure that each vertex is returned to the position in its former label class from which it was removed. To enable this, we perform a little extra bookkeeping during the `Filter` function: each time we remove a vertex u, we store v in an array together with a pointer to its successor node in the linked list from which u was removed. This array is no longer than $|N_G(v)|$ or $|N_H(v)|$, and the storage space for all of these arrays can be pre-allocated before search.

In the next section, we will see that MCSPLIT-SI-LL is space-efficient. The experimental evaluation in Section 5.8 shows that MCSPLIT-SI-LL and MCSPLIT-SI have broadly similar run times, but that each of the two algorithms has families of instances for which it is the faster solver.

5.5.1 Space Complexity

The following proposition demonstrates that the memory needed by MCSPLIT-SI-LL is bounded above by a constant multiple of the space used to store the adjacency lists of G and H. While this is greater than the space complexity of VF3 and RI, which require only $|V(G)|+|V(H)|$ space, it is an improvement over all previous constraint programming algorithms, which require at least $O(|V(G)||V(H)|)$ space just to represent the initial domains. The Glasgow Subgraph Solver uses $O(|V(H)|)$ space to represent adjacency matrices, and requires $O(|V(G)|^2|V(H)|)$ space to solve satisfiable instances.

Proposition 1. MCSPLIT-*SI-LL uses* $O(n_G + m_G + m_H)$ *space, where* $n_G = |V(G)|$, $m_G = |E(G)|$, *and* $m_H = |E(H)|$.

Proof. The function `Filter`(v, w) runs in $O(|N_G(v)| + |N_H(w)|)$ time and each of its operations allocates at most $O(1)$ space; therefore the function uses $O(|N_G(v)| + |N_H(w)|)$ space.

Each of the functions `SelectLabelClass`, `SelectVertex`, `Assign`, `Unassign`, and `Unfilter` uses $O(1)$ space and does not allocate any memory that is not released when

the function returns. Moreover, `Unfilter` releases all of the memory that was allocated by the corresponding call to `Filter`.

The mapping M can be no larger than n_G, so the maximum recursion depth for `Search` calls is $n_G + 1$. Now suppose we are at the start of the `Search` function at some given node at depth k of the search tree ($k \leq n_G + 1$). At earlier levels of the call stack, $k - 1$ calls to `Filter` have been made; the v parameter for these calls has taken distinct values $\{v_1, \ldots, v_{k-1}\} \subseteq V(G)$, and the w parameter has taken distinct values $\{w_1, \ldots, w_{k-1}\} \subseteq V(H)$. These `Filter` calls have used $\sum_{i=1}^{k-1} O(|N_G(v_i)| + |N_H(w_i)|) = O(m_G + m_H)$ space. In total, local variables used in the k recursive calls of `Search` use a further $O(k) = O(n_G)$ space. Thus, the total space usage is $O(n_G + m_G + m_H)$. □

5.6 McSplit-SI-AM: A Version for Dense Graphs

Our second alternative version of McSplit-SI stores graphs as adjacency matrices rather than adjacency lists. This does not use the McSplit-SI data structures described in this chapter, and is essentially a decision version of the McSplit solver for maximum common subgraph described in Chapter 3. We will see in Section 5.8 that the algorithm often outperforms McSplit-SI on instances with dense graphs. The main additional feature compared to the maximum common subgraph version of McSplit is a version of the lazy partitioning technique that was described in Section 5.3.2. Before running the partitioning algorithm, we iterate over the label classes, counting in each label class the number of vertices adjacent to v in the pattern graph and to w in the target class. This often allows us to backtrack without the need for the relatively expensive step of swapping vertices within a label class.

5.7 Generalised Arc Consistency on the All-Different Constraint

The constraint programming model for induced subgraph isomorphism in Section 2.6.2 contains an all-different constraint over all the variables; this constraint ensures that each of the pattern-graph vertices is mapped to a distinct vertex in the target graph. The strongest level of consistency that can be achieved for this constraint is *generalised arc consistency (GAC)*, in which value x may only be in the domain of variable X if there is some assignment to all the variables involved in the constraint such that X takes the value x and the constraint is satisfied. The classic algorithm for achieving GAC on an all-different constraint is Régin's (1994), which operates on a (perhaps implicit) bipartite graph with variables on the left and

values on the right, and deletes every edge that does not appear in any maximum matching. The algorithm operates by computing a maximum matching on the bipartite graph, then finding strongly connected components on an related directed graph. Many optimisations to the algorithm have been proposed since its introduction; see Gent et al. (2008) for a detailed review and empirical study.

The Glasgow Subgraph Solver (McCreesh and Prosser, 2015) introduces a new propagation algorithm, the *counting all-different propagator*. The algorithm iterates over the domains involved in the constraint, maintaining a set A containing all values seen so far. If, at any step during the algorithm, $|A|$ is smaller than the number of domains visited so far, the algorithm can backtrack. If $|A|$ equals the number of domains visited so far, then all of the members of A are added to a set H (the *Hall set*), and are deleted from the domains of subsequently visited variables.[2] The order in which variables are visited is crucial to the algorithm's effectiveness in practice; McCreesh and Prosser propose visiting variables in increasing order of domain size in order to keep the set A small.

The counting all-different propagator provides weaker filtering than Régin's propagator: it never deletes more values from domains than Régin's algorithm, and sometimes deletes fewer. Nevertheless, McCreesh and Prosser showed that it runs many times faster than Régin's algorithm and its filtering is almost as effective as that of Régin's algorithm in practice on a large set of benchmark instances.

This section has two contributions. First, we show that McSplit-SI achieves generalised arc consistency on the all-different constraint for free, without requiring an all-different propagator. Second, as an existence proof that this can provide benefits beyond those of the counting all-different propagator, we describe a family of instances that cannot be solved efficiently by Glasgow — or indeed by RI — but can be solved very quickly by McSplit-SI.

5.7.1 McSplit-SI Achieves GAC

In Proposition 2, we view the label classes as a domain store in which each label class $\langle S_G, S_H \rangle$ represents a set of $|S_G|$ variables, each with domain S_H, and show that McSplit-SI maintains GAC. The proof depends on the fact that domains in McSplit-SI are guaranteed to be either equal or disjoint, and also on the fact that McSplit-SI backtracks if $|S_G| > |S_H|$ for any label class.

Proposition 2. *At the start of every call to* `Search` *of* McSplit-*SI, the list of label classes is GAC with respect to the all-different constraint.*

[2] To simplify the presentation, I have made trivial changes from the algorithm described by McCreesh and Prosser. These changes affect neither the results nor the time complexity of the propagation algorithm.

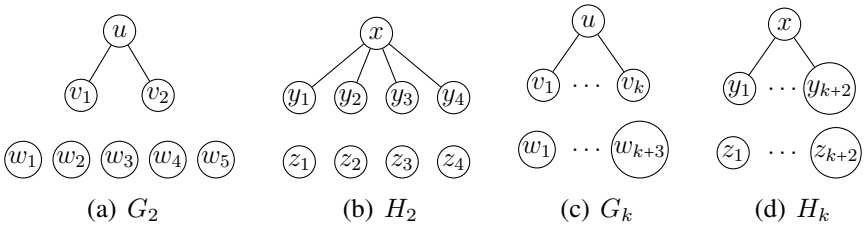
Proof. Let $\langle S_G^1, S_H^1\rangle, \ldots, \langle S_G^k, S_H^k\rangle$ be the list of label classes. We have that $|S_G^i| \leq |S_H^i|$ for $1 \leq i \leq k$, since the algorithm backtracks whenever this does not hold. Furthermore, a *disjointness property* holds: for $i \neq j$, we have $S_G^i \cap S_G^j = \varnothing$ and $S_H^i \cap S_H^j = \varnothing$ (which follows from the way label classes are partitioned, as discussed in Chapter 3).

Let $v \in S_G^i$ and $w \in S_H^i$ for some $1 \leq i \leq k$. We need to show that we can extend the mapping (v, w) to a complete assignment of the vertices in all of the S_G^j $(1 \leq j \leq k)$ such that no two vertices of G are mapped to the same vertex of H. This can be achieved by assigning, for each j $(1 \leq j \leq k)$ the vertices of $S_G^j \setminus \{v\}$ to any subset of $S_H^j \setminus \{w\}$ with $|S_G^j \setminus \{v\}|$ vertices. Such a subset exists since $|S_G^j| \leq |S_H^j|$. No two of these subsets contain the same vertex of H by the disjointness property. □

5.7.2 A Family of Instances Where McSplit-SI Outperforms Other Algorithms

In this subsection, we consider a family of graphs devised to demonstrate that the generalised arc consistency achieved by McSplit-SI can give a dramatic speed-up compared to algorithms that do not achieve GAC. The instances described here are presented as an existence proof, rather than as representative of real-world instances.

Consider the pattern graph G_2 in Figure 5.8(a) and the target graph H_2 in Figure 5.8(b). This induced subgraph isomorphism instance is unsatisfiable: u may only be mapped to x because the other vertices in H_2 have insufficient degree, after which we can deduce that each of the five isolated w_i vertices must be mapped to one of the four isolated z_j vertices.



(a) G_2 (b) H_2 (c) G_k (d) H_k

Figure 5.8: Example graphs G_2 and H_2, and their generalised versions G_k and H_k.

When solving this instance, McSplit-SI begins by mapping u to x. This leaves two label classes: $\langle\{v_1, v_2\}, \{y_1, y_2, y_3, y_4\}\rangle$ and $\langle\{w_1, w_2, w_3, w_4, w_5\}, \{z_1, z_2, z_3, z_4\}\rangle$. In the latter label class, the set S_G is larger than the set S_H, and therefore the algorithm can backtrack and terminate.

Now consider how the Glasgow algorithm behaves on this instance. After mapping u to x, the domains correspond to our label classes, as shown in the first two columns of Table 5.2. The remaining two columns of the table illustrate the behaviour of the counting all-different

propagator on these domains. (Since all domains are of the same size, the stable sort function used by the algorithm does not reorder the domains.) The third column shows set A, which is the union of domains in the current and previous rows. The fourth column shows the number of variables up to and including the current row. Since $|A| \geq n$ on each row, the propagator does not delete any values from the domains and does not conclude that we can backtrack.

Variable	Domain	A	n
v_1	$\{y_1, y_2, y_3, y_4\}$	$\{y_1, y_2, y_3, y_4\}$	1
v_2	$\{y_1, y_2, y_3, y_4\}$	$\{y_1, y_2, y_3, y_4\}$	2
w_1	$\{z_1, z_2, z_3, z_4\}$	$\{y_1, y_2, y_3, y_4, z_1, z_2, z_3, z_4\}$	3
w_2	$\{z_1, z_2, z_3, z_4\}$	$\{y_1, y_2, y_3, y_4, z_1, z_2, z_3, z_4\}$	4
w_3	$\{z_1, z_2, z_3, z_4\}$	$\{y_1, y_2, y_3, y_4, z_1, z_2, z_3, z_4\}$	5
w_4	$\{z_1, z_2, z_3, z_4\}$	$\{y_1, y_2, y_3, y_4, z_1, z_2, z_3, z_4\}$	6
w_5	$\{z_1, z_2, z_3, z_4\}$	$\{y_1, y_2, y_3, y_4, z_1, z_2, z_3, z_4\}$	7

Table 5.2: A demonstration of the counting all-different propagator on G_2 and H_2 after assigning u to x.

The final two graphs in Figure 5.8 generalise G_2 and H_2; as k is incremented, a vertex is added to each of the v, w, y and z sets. Table 5.3 shows run times for the enumeration problem using McSplit-SI, Glasgow, Glasgow with no supplemental graphs, and RI for a range of values of k.[3] McSplit-SI solves the instance k = 1 000 000 in less than a second; Glasgow and RI time out on the k = 10 and k = 6 instances respectively.

k	McSplit-SI	Glasgow	Glasgow-NS	RI
3	0	0	0	2
4	0	0	0	86
5	0	2	2	4305
6	0	21	19	*
7	0	195	191	*
8	0	2083	1966	*
9	0	24254	23443	*
10	0	*	*	*
10000	6	*	*	*
100000	66	*	*	*
1000000	676	*	*	*

Table 5.3: Run times in ms for the induced subgraph isomorphism enumeration problem on G_k and H_k. An asterisk indicates timeout at 100 seconds. Glasgow-NS is the Glasgow Subgraph Solver with supplemental graphs disabled.

[3] VF3 was excluded from the experiment because the version available when the experiment was run did not handle disconnected graphs correctly.

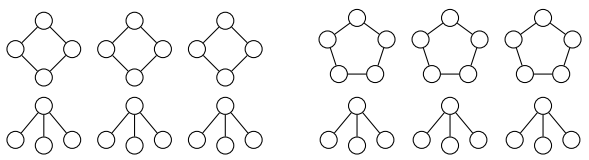
5.7.3 A Family of Instances Where McSplit-SI is Outperformed by Glasgow

In the previous subsection, we saw a family of instances on which McSplit-SI outperforms the Glasgow Subgraph Solver. We can easily construct a family of instances where the reverse is true. Consider the graphs in Figure 5.9. Graph G'_3 consists of three copies of the cycle C_4 and three copies of the star $K_{1,3}$, with no additional edges. Graph H'_3 consists of three copies of C_5 and three copies of $K_{1,3}$. We generalise these definition to G'_k and H'_k, which have k copies the cycle and star rather than 3. For $k \geq 1$, no instance (G'_k, H'_k) is satisfiable, since H'_k does not contain an induced 4-cycle.

The McSplit-SI algorithm prefers to map high-degree vertices first, and therefore its first decisions on these instances involves the stars rather than the cycles. Only at deeper levels of the search tree, when all of the stars have been mapped, does the algorithm begin to work on the cycles, at which point it can backtrack. Unfortunately, there are $6^k k!$ ways to map the stars of G'_k to the stars of H'_k, since any star in the pattern graph can be mapped to any star in the target graph and there are six possible ways to map the leaf nodes of any star. Therefore McSplit-SI's search tree is very large even for small values of k.

With supplemental graphs disabled, the Glasgow algorithm suffers from the same problem. However, supplemental graphs allow the Glasgow Subgraph Solver to solve these instances without search. In particular, consider the supplemental graph in which v and w are adjacent if and only there are at least two 2-paths between v and w in the original graph. This supplemental graph contains edges between vertices of the 4-cycles in the pattern graph, but has no edges for the target graph.

Table 5.4 shows that supplemental graphs make a huge difference to Glasgow's run times in practice on this family of instances. The table shows times for McSplit-SI, Glasgow, Glasgow without supplemental graphs, and RI. McSplit-SI and RI have similar run times, and cannot solve instances with k greater than 5 within the 100 second time limit. Glasgow can solve the instance with k = 1000 (a 7000-vertex pattern graph and an 8000-vertex target graph) in less than 4 seconds.



(a) G'_3 (b) H'_3

Figure 5.9: Example graphs G'_3 and H'_3.

As we have seen in this section, neither McSplit-SI nor Glasgow dominates the other

k	McSplit-SI	Glasgow	Glasgow-NS	RI
1	0	0	0	0
2	0	0	1	0
3	10	0	47	10
4	329	0	1793	324
5	12522	0	80880	11987
6	*	0	*	*
7	*	0	*	*
8	*	0	*	*
9	*	0	*	*
10	*	0	*	*
100	*	27	*	*
1000	*	3825	*	*

Table 5.4: Run times in ms for the induced subgraph isomorphism enumeration problem on G'_k and H'_k. An asterisk indicates timeout at 100 seconds. Glasgow-NS is the Glasgow Subgraph Solver with supplemental graphs disabled.

solver. In the next section, we perform an experimental evaluation to compare solvers on benchmark instances.

5.8 Experimental Evaluation

We now compare the speed of McSplit-SI with existing state-of-the-art algorithms on three sets of benchmark instances. Section 5.8.2 uses an existing large, heterogeneous set of pairs of unlabelled graphs. In Section 5.8.3, we consider a new set of decision-problem instances recently proposed by Donald Knuth based on a generalisation of the knight's tour problem. Finally, in Section 5.8.4, we use pairs of random directed graphs based on another existing benchmark set; for these instances, we solve the enumeration problem of counting all induced subgraph isomorphisms from the pattern to the target.

5.8.1 Solvers Used

Our experiments compare McSplit-SI with four state-of-the-art subgraph isomorphism solvers: VF3, RI, RI-DS, and Glasgow. In addition, we use PathLAD in Section 5.8.3; this solver has been shown in prior work to be outperformed by Glasgow and VF3 respectively on the other two sets of instances (Carletti et al., 2018; Solnon, 2019). These solvers are described in Section 2.7. In terms of effort per search node, McSplit-SI may be viewed intuitively as sitting between Glasgow and PathLAD on one hand and VF3, RI, and RI-DS on the other.

We use the 14 March 2022 version of the Glasgow solver, published online.[4] In addition,

[4] `https://github.com/ciaranm/glasgow-subgraph-solver`

we report results of the Glasgow solver with supplemental graphs switched off. The most recent versions of all other solvers are used.

The MCSPLIT-SI variants used are shown in Table 5.5.

Name	Description
MCSPLIT-SI	The base MCSPLIT-SI algorithm
MCSPLIT-SI-LL	MCSPLIT-SI with lists of vertices stored as sorted doubly linked lists (Section 5.5)
MCSPLIT-SI-AM	An adaptation of MCSPLIT decision-problem solver (using adjacency matrix representation of graphs) to the induced subgraph isomorphism problem. This does not use MCSPLIT-SI's partitioning algorithm optimised for sparse graphs. (Section 5.6)
MCSPLIT-SI-static	MCSPLIT-SI with the dynamic variable-ordering heuristic replaced by the static heuristic of RI (Bonnici and Giugno, 2017) (Section 5.4.1)

Table 5.5: Summary of MCSPLIT-SI variants used in this chapter's experiments

5.8.2 Decision Instances

Our first set of benchmark instances is a collection of $14,621$ pairs of undirected, unlabelled graphs. This collection was used by Archibald et al. (2019), extending the work of Kotthoff et al. (2016); it assembles instances from several smaller benchmarks.

Families of instances The benchmark set contains graphs in the following families.

- **Scalefree:** Randomly generated scale-free graphs (Barabási, 2003): 80 satisfiable instances and 20 unsatisfiable instances. In each instance, $|V(G)| = 0.9 \times |V(H)|$. Source: Zampelli et al. (2010)
- **LV:** Pairs of graphs derived from the Stanford GraphBase (Knuth, 1993). Source: Larrosa and Valiente (2002).
- **BVG:** Regular and irregular bounded valence target graphs, randomly generated. In the regular instances, the vertices of the target graph all have degree 3, 6, or 9 depending on the instance. The irregular instances have variable degree but a fixed total number of edges. This and the following two families are from the database of instances described in Santo et al. (2003). For each instance in these three families, the pattern graphs were generated by taking a random connected induced subgraph of the target graph. All instances are therefore satisfiable.
- **M4D:** Regular and irregular 4-dimensional meshes. The irregular instances have extra edges added to the mesh graph.
- **Rand:** Erdős-Rényi $G(n, p)$ random graphs.

- **Phase:** Pairs of Erdős-Rényi $G(n, p)$ random graphs with parameters chosen near the satisfiable/unsatisfiable phase transition in order to make the instances very challenging despite having only 30 vertices per pattern graph and 150 vertices per target graph. (McCreesh et al., 2018)

- **PR**, **Meshes** and **Images:** Instances derived from a pattern recognition problem, where the graphs are generated from the adjacencies of regions of an image (PR, Images) or 3D object model (Meshes). (Solnon et al., 2015; Damiand et al., 2011)

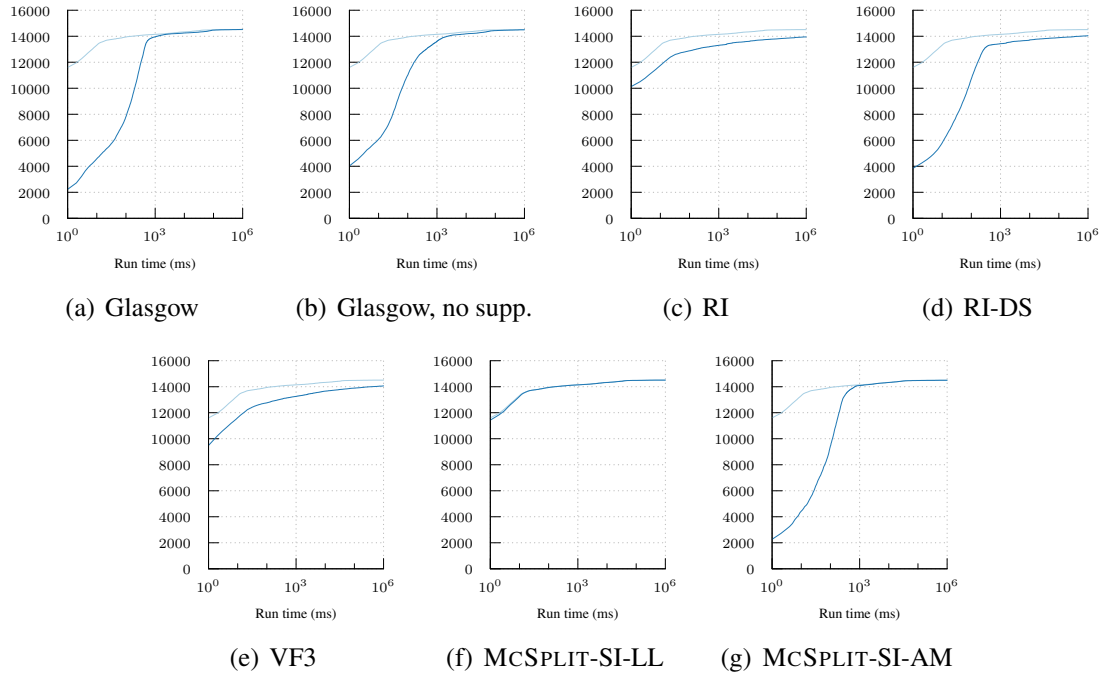
Table 5.6 shows a summary of the instances in each family. Most of the graphs are sparse; the LV and Phase families are unusual in that they contain some very dense target graphs.

Family	Count	Pattern graph				Target graph			
		$n_{\min}$	$n_{\max}$	$d_{\min}$	$d_{\max}$	$n_{\min}$	$n_{\max}$	$d_{\min}$	$d_{\max}$
Scalefree	100	180	900	0.0060	0.1646	200	1000	0.0060	0.1594
LV	3831	10	128	0.0198	1.0000	10	6671	0.0006	1.0000
BVG	540	40	480	0.0055	0.2013	200	800	0.0037	0.0452
M4D	360	51	777	0.0045	0.0839	256	1296	0.0043	0.0253
Rand	270	40	360	0.0201	0.2090	200	600	0.0201	0.1915
Phase	200	30	30	0.2943	0.8897	150	150	0.3698	0.7821
PR	24	4	170	0.0168	0.6667	4838	4838	0.0006	0.0006
Meshes	3018	40	199	0.0219	0.1462	201	5873	0.0004	0.0224
Images	6278	15	151	0.0186	0.1905	1072	5972	0.0005	0.0027

Table 5.6: Summary of instance families. The values $n_{\min}$ and $n_{\max}$ are the smallest and largest vertex counts; the values $d_{\min}$ and $d_{\max}$ are the smallest and largest densities.

Results Figure 5.10 shows cumulative plots of instances solved by McSplit-SI and the other algorithms. To avoid a tangle of overlapping curves, each solver's curve is shown as a dark blue line on its own plot, with the curve for McSplit-SI shown in light blue for comparison on each plot. The solvers can be split into three broad categories. Constraint programming solvers tend to run slowly on the easiest instances but perform well for hard instances. The two versions of Glasgow are in this category; McSplit-SI-AM has similar results. Pattern recognition solvers—RI and VF3 in particular—perform well on easy instances but solve fewer of the hard instances than the constraint programming solvers; Solnon (2019) makes a similar observation for these two solver categories. Finally, McSplit-SI and McSplit-SI-LL perform well on easy instances—indeed, outperforming the pattern recognition solvers—while solving almost as many of the hard instances as Glasgow.

Figure 5.11 shows a scatter plot of run times for each solver, with the run time of McSplit-SI on the horizontal axis in each plot. McSplit-SI compares favourably to each of the non-McSplit solvers in the first five plots, although each of these has some instances



(a) Glasgow (b) Glasgow, no supp. (c) RI (d) RI-DS

(e) VF3 (f) McSplit-SI-LL (g) McSplit-SI-AM

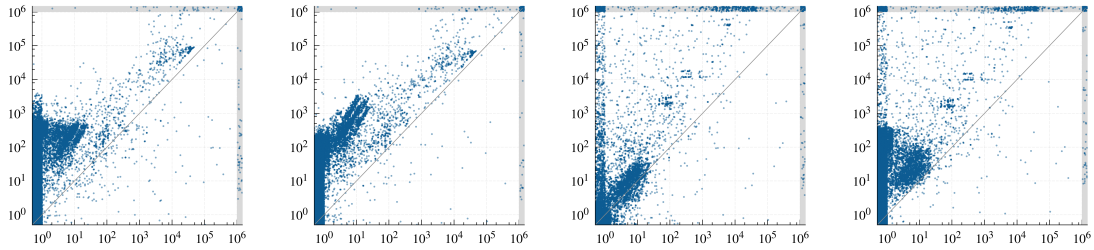
Figure 5.10: Cumulative plots of run times for decision instances. In each subfigure, the algorithm named in the caption is in dark blue and McSplit-SI is in light blue.

that it can solve in less than a second while McSplit-SI exceeds the time limit; this is likely to be due to a combination of different variable and value ordering heuristics and, in the case of Glasgow, stronger filtering.

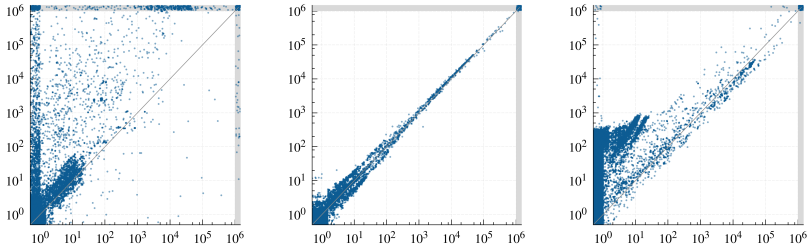
To give a finer-grained look at the performance of the two best solvers overall—McSplit-SI and Glasgow—Figure 5.12 shows those solvers' run times for each instance, with a separate sub-figure for each family of instances. On the PR, Phase, Meshes, and Images families, McSplit-SI is the clear winner. On each of the remaining five families, McSplit-SI is the faster solver on most instances, but Glasgow is much faster than McSplit-SI on a small proportion of instances.

On an instance-by-instance basis, does McSplit-SI typically run faster than the best of the other solvers? To answer this question, we calculate the run time of the virtual best other solver (VBOS) for each instance by taking the lowest run time of the five non-McSplit solvers: Glasgow, Glasgow without supplemental graphs, RI, RI-DS, and VF3. This VBOS is essentially equivalent to running the constituent solvers in parallel and stopping when the first solver terminates; we can also view it as an idealised portfolio solver with an oracle that selects the best non-McSplit solver for a given instance.

Figure 5.13 shows the run times of McSplit-SI and VBOS (vertical axis) for each instance. On six families of instances (BVG, Rand, LV, Phase, Images, Meshes), the number of instances where McSplit-SI is faster than VBOS is greater than the number of instances where the reverse is true. On the other hand, McSplit-SI certainly does not dominate the



(a) Glasgow (b) Glasgow, no supp. (c) RI (d) RI-DS



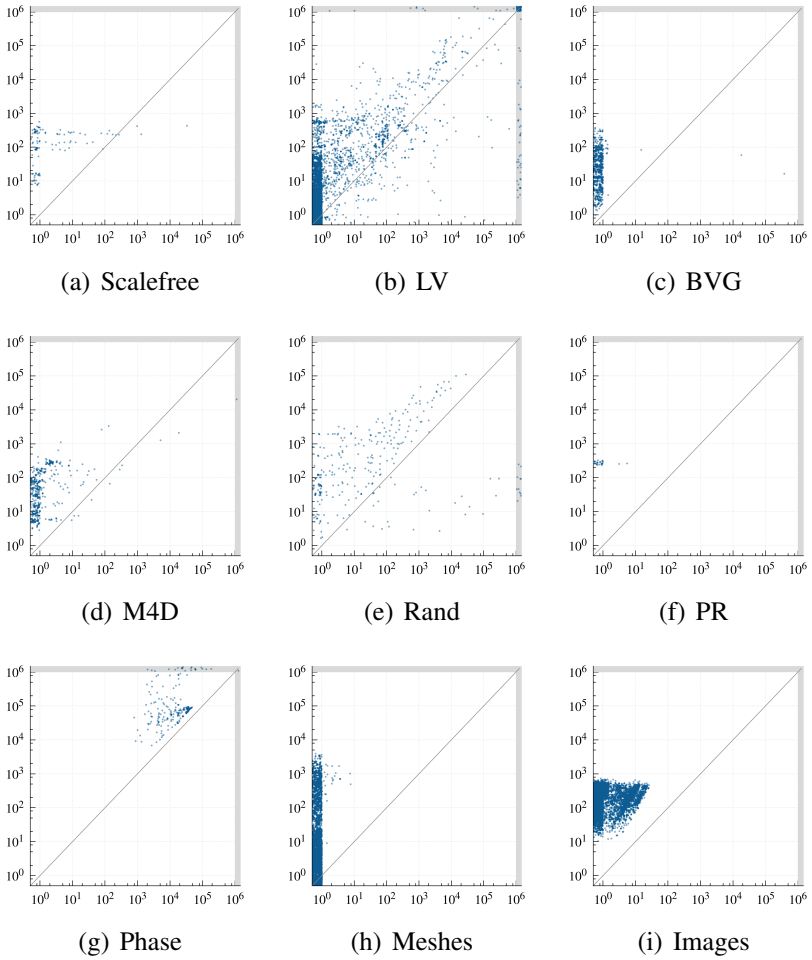
(e) VF3 (f) McSplit-SI-LL (g) McSplit-SI-AM

Figure 5.11: Scatter plots of run times in ms for decision instances. The horizontal and vertical axes show the run times of McSplit-SI and the solver named in the subfigure caption, respectively.

other solvers; indeed, there are families such as Scalefree where McSplit-SI appears to be worse overall than VBOS.

Presolve Solnon (2019) considers a solver that runs VF3 for 100 ms, then switches to Glasgow if a solution is not found. This outperforms Glasgow on easy instances while retaining almost all of the benefit of Glasgow on hard instances. Are there pairs of solvers in our experiment that work particularly well using such a 100 ms presolve? Table 5.7 shows the number of instances solved within 1000 seconds by each pair of solvers, excluding McSplit-SI-AM and RI-DS since these are outperformed by McSplit-SI and RI respectively. Seven pairs of solvers—each of which used a version of McSplit-SI as the presolver, the main solver, or both—solved more instances than Glasgow, which was the single best individual solver by this measure with $14,524$ instances solved. The best solver pair solved $14,546$ instances; this uses McSplit-SI-static as the main solver and McSplit-SI as the presolver. (A version using McSplit-SI-static as the main solver and Glasgow without supplemental graphs as the presolver also performed well, solving just one instance fewer.)

Figure 5.14 compares the run times of McSplit-SI-static + McSplit-SI presolve (x axis) with those of seven other algorithms (y axes) on an instance-by-instance basis. Unlike in Figure 5.11, which did not use presolve, we see few hard instances where Glasgow is much faster than the presolve version of McSplit-SI, and almost no hard instances where a pattern recognition solver (RI, RI-DS or VF3) is faster than the presolve version of McSplit-SI.



(a) Scalefree (b) LV (c) BVG

(d) M4D (e) Rand (f) PR

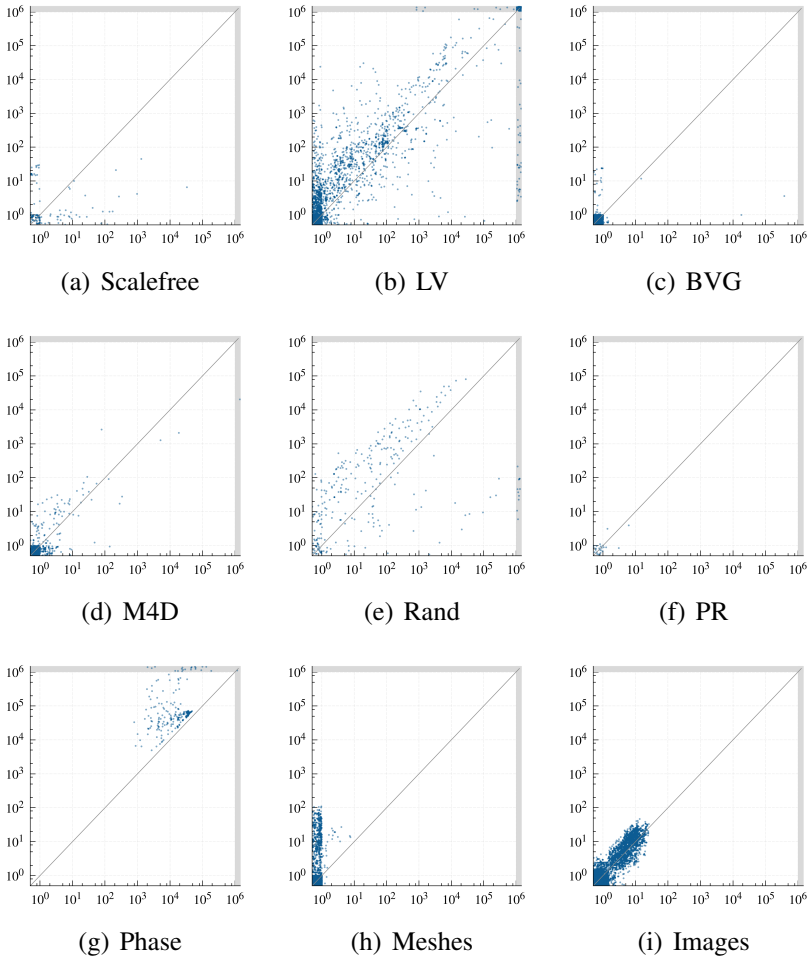
(g) Phase (h) Meshes (i) Images

Figure 5.12: Run times in ms of MCSPLIT-SI (horizontal axis) and Glasgow (vertical axis), by family.

Number of instances solved Table 5.8 shows, for each family and each solver, the number of instances that were solved within the 1000-second time limit. The final column, "McS pre.", refers to a solver that runs MCSPLIT-SI for 100 ms as a presolver, then runs MCSPLIT-SI-static for the remaining time. For every family, this solver is able to solve at least as many instances as the best other solver.

"Phase" is a family on which all MCSPLIT-SI variants perform particularly well in Table 5.8. This family is unusual in that the graphs in many of the instances have density greater than $1/2$. Much of the success of MCSPLIT-SI on these instances is due to the strategy of sorting graphs in increasing order of degree if the target graph is dense (Section 5.4). I re-ran the experiment with MCSPLIT-SI using the decreasing-degree sort order for all graphs (which is consistent with Glasgow's strategy); only 179 rather than 199 Phase instances were solved within the time limit.

Number of instances for which each solver was the fastest Table 5.9 shows, for each family and each solver, the number of instances for which the solver was the fastest



(a) Scalefree (b) LV (c) BVG

(d) M4D (e) Rand (f) PR

(g) Phase (h) Meshes (i) Images

Figure 5.13: Run times in ms of McSplit-SI (horizontal axis) and the virtual best other solver (vertical axis), by family.

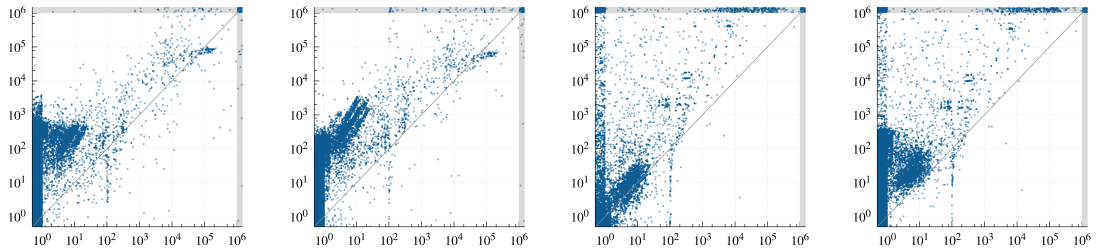
of all solvers shown. For seven of the nine families, McSplit-SI had the highest count of "wins". Glasgow performs poorly by this measure; for five families, there were no instances for which it was the fastest solver.

The effect of target graph density When is it better to use McSplit-SI than McSplit-SI-AM? Figure 5.15(a) has a point per instance, with target graph density on the x axis and the ratio of the two solvers' run times on the y axis; a ratio below 1 indicates that McSplit-SI is the faster solver. Trivial instances and instances where at least one solver timed out are excluded. There is a clear positive relationship between density and ratio of run times. For most instances of density below 0.002, McSplit-SI is more than 10 times faster than McSplit-SI-AM. However, McSplit-SI-AM soon begins to catch up as target graph density increases, and becomes the faster solver at a target graph density of around 0.1.[5]

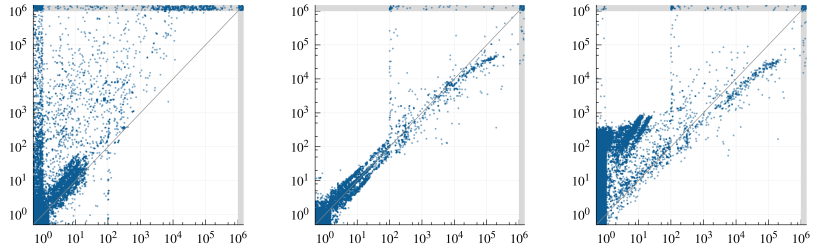
[5]This analysis was carried out using target graph density because we would expect the target graph to have a larger effect on run times than the (smaller) pattern graph. We also re-created the plot with the x axis showing

	McS-SI	McS-SI-s	Gla	Gla, no supp.	RI	VF3
McS-SI		14 546	14 526	14 498	14 278	14 254
McS-SI-s	14 536		14 526	14 512	14 067	14 142
Gla	14 537	14 515		14 510	14 051	14 115
Gla, no supp.	14 523	14 545	14 524		14 251	14 210
RI	14 528	14 512	14 524	14 505		14 070
VF3	14 511	14 511	14 524	14 496	13 974	

Table 5.7: Number of instances solved within 1000 seconds using a 100 ms presolve. Each number refers to a solver that runs the solver named in the row for 100 ms, then switches to the solver named in the column. Underlined values are greater than the number of instances solved by Glasgow without presolve (14, 524). McS-SI-s=MCSPLIT-SI-static.



(a) Glasgow (b) Glasgow, no supp. (c) RI (d) RI-DS



(e) VF3 (f) MCSPLIT-SI-LL (g) MCSPLIT-SI-AM

Figure 5.14: Scatter plots of run times in ms for decision instances. The horizontal axis shows the run time of MCSPLIT-SI-static with a 100 ms MCSPLIT-SI presolve; the vertical axis shows the run time of the solver named in the subfigure caption.

Figure 5.15(b) is a similar plot comparing the run time of MCSPLIT-SI to that of Glasgow. (Note the larger range of values of the y axis.) For very sparse target graphs with density below around 0.002, MCSPLIT-SI consistently runs faster than Glasgow—often by more than an order of magnitude. For most of the instances with denser target graphs, MCSPLIT-SI is again the faster algorithm, and this holds true even for very dense graphs which are not well suited to MCSPLIT-SI's data structures. However, there is a long tail of instances for which a combination of Glasgow's restarts and stronger filtering methods results in an orders-of-magnitude speedup over MCSPLIT-SI.

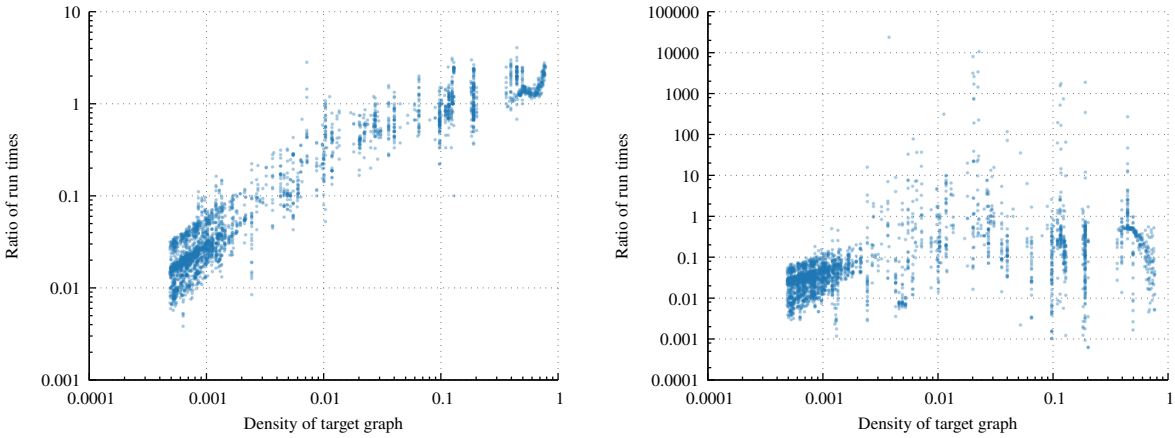
the proportion of all possible edges across pattern *and* target graphs: $(m_G + m_H)/(n_G(n_G - 1) + n_H(n_H - 1))$, where m_G and n_G denote the edge and vertex counts of G. The resulting plot was almost identical to Figure 5.15(a).

Family	Count	McS-SI	McS-SI-LL	Gla	Gla, no supp.	RI	VF3	McS pre.
Scalefree	100	100	100	100	100	90	98	100
LV	3 831	3 732	3 733	3 756	3 736	3 612	3 589	3 757
BVG	540	540	540	540	540	537	539	540
M4D	360	359	359	360	360	360	356	360
Rand	270	259	259	270	260	263	263	270
Phase	200	199	199	178	179	5	4	199
PR	24	24	24	24	24	24	24	24
Meshes	3 018	3 018	3 018	3 018	3 018	2 781	2 914	3 018
Images	6 278	6 278	6 278	6 278	6 278	6 278	6 278	6 278
TOTAL	14 621	14 509	14 510	14 524	14 495	13 950	14 065	14 546

Table 5.8: The number of instances in each family that were solved by each solver within the 1000 second time limit. The "Count" column shows the total number of instances per family. An underlined value indicates that no other solver solved a greater number of instances in this family.

Family	Count	McS-SI	McS-SI-LL	Gla	Gla, no supp.	RI	VF3	McS pre.
Scalefree	100	61	44	0	0	59	42	61
LV	3 831	3 229	3 009	1 023	1 450	2 067	1 659	3 189
BVG	540	524	509	0	16	491	437	524
M4D	360	248	248	6	0	269	137	248
Rand	270	152	77	22	1	50	21	151
Phase	200	194	0	0	0	0	0	5
PR	24	20	21	0	0	20	18	20
Meshes	3 018	3 004	2 904	626	1 160	2 464	1 753	3 004
Images	6 278	5 134	4 474	0	1	4 425	3 622	5 134
TOTAL	14 621	12 566	11 286	1 677	2 628	9 845	7 689	12 336

Table 5.9: For each family of instances, the number of instances for which each solver's run time equalled the best run time among all solvers shown. The second column shows the total number of instances per family. Underlined values are the best (or joint-best) for that family.

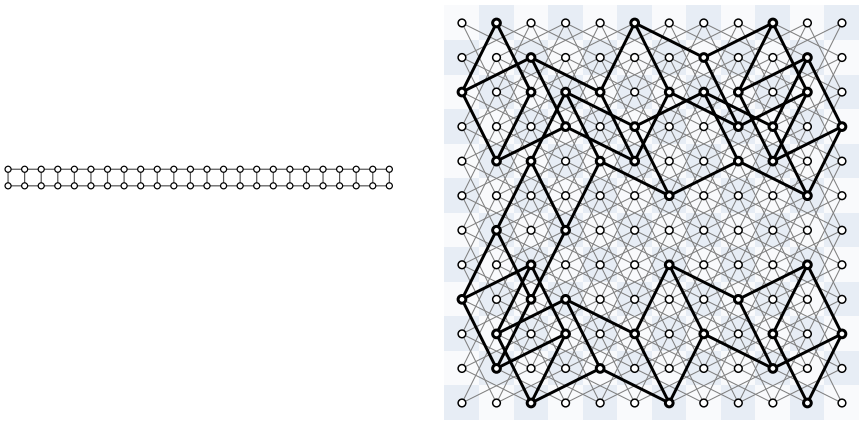


(a) McSplit-SI-AM (b) Glasgow

Figure 5.15: On the horizontal axis, the target graph density; on the vertical axis, the ratio of McSplit-SI run time to the run time of the algorithm named in the subfigure caption. Instances where either solver took less than 1 ms or exceeded the time limit are excluded.

5.8.3 Knight's Grid Instances

Our second set of decision-problem instances were proposed by Knuth in a recent prefascicle of The Art of Computer Programming (Knuth, 2022). Each pattern graph is a rectangular grid $P_m \,\square\, P_n$, and each target graph is a knight graph N_t which has one vertex for each square of a $t \times t$ chessboard and an edge between any two squares that are a knight's move (two squares horizontally and one square vertically or vice versa) apart. Figure 5.16 shows the pattern and target graph for one such instance, with a solution shown on the target graph.



(a) The grid $P_2 \,\square\, P_{24}$ (b) The knight graph N_{12}

Figure 5.16: A satisfiable knight's grid instance. On the left is the pattern graph; on the right is the target graph with an induced subgraph isomorphic to the pattern shown by heavy nodes and edges.

If $m = 1$ and $n = t^2$, we have the classic knight's tour problem in which a single knight must visit every square of the chessboard without revisiting any square. The instances proposed by Knuth have $m \in \{2, 3, 4\}$. Clearly, the embedding of each row and of each column of the grid graph must itself be a miniature knight's tour. Thus, these instances may be interpreted as the problem of moving m knights — themselves organised in the formation of a knight's tour of length m — around the board without changing their positions relative to each other, and without revisiting any square. In the induced case (which is the only case we consider here), there is the additional restriction that no knight attacks any square except the square that it has just visited, the square it will visit next, and the squares of its neighbours in the miniature tour of length m.

Instances For $m \in \{2, 3, 4\}$, we considered instances of the form "is $P_m \,\square\, P_n$ isomorphic to an induced subgraph of N_k?", with $k \geq 4$ and n equal to either the largest value for which the instance is satisfiable or the smallest value for which the instance is unsatisfiable. The vertices of each graph were randomly permuted to ensure that the vertex order does not give

clues to the solvers that might make the satisfiable instances artificially easy. The same pair of permuted graphs was given to each solver.

Results for satisfiable instances Tables 5.10 to 5.12 show run times for satisfiable instances of the form $P_m \square P_n$ for $m \in \{2, 3, 4\}$. MCSPLIT-SI-static, Glasgow without supplemental graphs, and RI are excluded from the tables for brevity, because MCSPLIT-SI, Glasgow, and RI-DS respectively perform similarly or better. MCSPLIT-SI and MCSPLIT-SI-LL both outperformed MCSPLIT-SI-AM, as we would expect given the sparsity of the instances. MCSPLIT-SI-LL was consistently slightly faster than MCSPLIT-SI.

For these satisfiable instances, there was no clear winning solver. Each of MCSPLIT-SI-LL, Glasgow, and RI-DS each was the fastest solver for some instances. We observe some surprising behaviour: larger instances are often solved much more quickly than smaller ones by a given solver. In Table 5.10, for example, MCSPLIT-SI-LL solves the instance with target N_{16} faster than the instance with target N_{15}. This is probably nothing more than an artifact of the way vertices were randomly permuted in the instance files. In the author's experience, different random permutations give very different run times for these instances. (We might therefore expect good results from a solver that tries solving different permutations of the pattern and target graphs in parallel. This is left as future work.)

G	H	McS-SI	McS-SI-LL	McS-SI-AM	Glasgow	RI-DS	VF3	pathLAD
$P_2 \square P_{15}$	K_{10}	5	4	11	36	<u>3</u>	8	26
$P_2 \square P_{19}$	K_{11}	10	8	14	<u>5</u>	18	60	132
$P_2 \square P_{24}$	K_{12}	838	<u>664</u>	1 130	969	1 376	3 657	12 583
$P_2 \square P_{28}$	K_{13}	3 253	<u>2 745</u>	4 362	10 776	4 511	10 541	48 231
$P_2 \square P_{32}$	K_{14}	25 910	<u>22 048</u>	37 363	40 292	99 807	199 784	531 638
$P_2 \square P_{36}$	K_{15}	5 548 230	4 826 963	7 801 939	*	<u>182 898</u>	540 235	2 027 954
$P_2 \square P_{40}$	K_{16}	290 309	245 771	460 528	<u>145 627</u>	1 240 156	2 373 784	6 063 535
$P_2 \square P_{46}$	K_{17}	379 949	<u>329 911</u>	585 019	*	*	4 260 239	*
$P_2 \square P_{52}$	K_{18}	*	*	*	*	*	*	*

Table 5.10: Runtimes in ms for satisfiable knight's grid instances with pattern graphs of the form $P_2 \square P_n$. An asterisk indicates timeout at 10 000 seconds; the best run time for each instance is underlined. Trivial instances that all solvers could solve in less than 10 ms are not shown.

Results for unsatisfiable instances Tables 5.13 to 5.15 show run times for unsatisfiable instances. For each of these families of instances, MCSPLIT-SI-LL is the clear overall winner. Among the non-MCSPLIT solvers, Glasgow is the fastest constraint programming solver and RI-DS is the fastest other solver.

Knuth's specialised program The goal of this experiment was to compare a set of general-purpose subgraph isomorphism solvers on an interesting set of benchmark instances,

G	H	McS-SI	McS-SI-LL	McS-SI-AM	Glasgow	RI-DS	VF3	pathLAD
$P_3 \square P_{10}$	K_{10}	15	22	33	6	12	917	173
$P_3 \square P_{11}$	K_{11}	182	123	277	25	22	5 079	239
$P_3 \square P_{12}$	K_{12}	25	22	42	44	39	10 904	441
$P_3 \square P_{14}$	K_{13}	341	275	523	242	179	348 164	1 196
$P_3 \square P_{16}$	K_{14}	3	5	7	286	18	685 783	206
$P_3 \square P_{20}$	K_{15}	181	165	311	133	1 354	*	8 432
$P_3 \square P_{21}$	K_{16}	120	108	243	80	386	*	3 449
$P_3 \square P_{25}$	K_{17}	89	77	230	498	212	*	1 644
$P_3 \square P_{28}$	K_{18}	2 323	1 963	3 945	1 813	2 120	*	127 279
$P_3 \square P_{32}$	K_{19}	26 928	23 823	55 799	8 254	3 821	*	47 641
$P_3 \square P_{34}$	K_{20}	19 686	17 720	39 567	36 385	69 904	*	954 517
$P_3 \square P_{41}$	K_{21}	539 151	488 436	1 149 339	584 330	2 017 613	*	*
$P_3 \square P_{44}$	K_{22}	3 605 979	3 196 204	7 709 573	280 619	*	*	*
$P_3 \square P_{49}$	K_{23}	*	*	*	*	*	*	*

Table 5.11: Runtimes in ms for satisfiable knight's grid instances with pattern graphs of the form $P_3 \square P_n$.

but we should note that the solvers compared here are far from the fastest way to solve the specific problem of embedding a large grid in an knight graph. Knuth (2022) has written six specialised programs for finding such maximal embeddings. To give an example of the programs' run times, the program `back-knightgrid2-strict` takes less than three seconds to find the largest k such that $P_2 \square P_k$ is isomorphic to an induced subgraph of N_{14}. The fastest program in our experiment needs more than 600 seconds to solve the same problem. Among several other interesting techniques, Knuth's programs use symmetry breaking; it would certainly be possible to add symmetry breaking to each of the solvers considered here in order to decrease their run times on these instances (Zampelli et al., 2007).

G	H	McS-SI	McS-SI-LL	McS-SI-AM	Glasgow	RI-DS	VF3	pathLAD
$P_4 \square P_8$	K_{10}	2	1	2	15	2	3	20
$P_4 \square P_8$	K_{11}	0	0	0	5	4	0	16
$P_4 \square P_{10}$	K_{12}	5	3	6	4	16	71	149
$P_4 \square P_{12}$	K_{13}	31	27	37	6	81	925	531
$P_4 \square P_{12}$	K_{14}	5	4	11	5	79	797	324
$P_4 \square P_{14}$	K_{15}	3	3	4	9	63	829	200
$P_4 \square P_{15}$	K_{16}	46	41	53	177	73	6 620	805
$P_4 \square P_{17}$	K_{17}	128	80	232	205	738	137 528	6 441
$P_4 \square P_{18}$	K_{18}	2 164	2 024	4 631	861	34 949	7 272 239	171 695
$P_4 \square P_{20}$	K_{19}	853	813	1 196	1 317	12 148	*	61 331
$P_4 \square P_{20}$	K_{20}	53	49	117	394	16 241	5 612 600	44 175
$P_4 \square P_{22}$	K_{21}	644	616	1 604	909	47 454	*	129 532
$P_4 \square P_{24}$	K_{22}	15 511	13 632	37 208	83 256	2 199 540	*	2 276 484
$P_4 \square P_{25}$	K_{23}	4 091	3 659	12 321	5 010	1 105 560	*	73 827
$P_4 \square P_{26}$	K_{24}	6 686	6 772	18 261	971	1 761 964	*	660 051
$P_4 \square P_{28}$	K_{25}	4 669	4 342	15 783	938	*	*	270 330
$P_4 \square P_{29}$	K_{26}	15 680	14 270	47 237	380 938	*	*	5 051 750
$P_4 \square P_{31}$	K_{27}	248 356	224 468	801 511	1 784 939	*	*	*
$P_4 \square P_{32}$	K_{28}	27 628	25 833	108 558	1 616 244	*	*	2 321 130
$P_4 \square P_{34}$	K_{29}	141 654	127 916	574 230	334 914	*	*	*
$P_4 \square P_{35}$	K_{30}	147 499	131 701	583 768	19 584	*	*	*
$P_4 \square P_{37}$	K_{31}	*	*	*	122 853	*	*	*
$P_4 \square P_{38}$	K_{32}	*	*	*	6 734 566	*	*	*
$P_4 \square P_{40}$	K_{33}	*	*	*	*	*	*	*

Table 5.12: Runtimes in ms for satisfiable knight's grid instances with pattern graphs of the form $P_4 \square P_n$.

G	H	McS-SI	McS-SI-LL	McS-SI-AM	Glasgow	RI-DS	VF3	pathLAD
$P_2 \square P_9$	K_6	1	1	2	3	2	4	14
$P_2 \square P_9$	K_7	4	3	9	15	11	25	66
$P_2 \square P_{11}$	K_8	13	11	17	52	38	84	159
$P_2 \square P_{13}$	K_9	50	41	66	126	147	337	680
$P_2 \square P_{16}$	K_{10}	330	281	455	1 049	738	1 670	5 491
$P_2 \square P_{20}$	K_{11}	1 673	1 376	2 146	5 979	3 641	8 267	30 007
$P_2 \square P_{25}$	K_{12}	5 692	4 839	8 250	24 484	21 866	40 070	100 831
$P_2 \square P_{29}$	K_{13}	31 851	27 071	49 951	147 194	138 465	290 711	630 856
$P_2 \square P_{33}$	K_{14}	528 697	446 389	748 694	2 870 431	1 564 256	3 208 128	7 911 484
$P_2 \square P_{37}$	K_{15}	8 437 193	7 151 381	*	*	*	*	*
$P_2 \square P_{41}$	K_{16}	*	*	*	*	*	*	*

Table 5.13: Runtimes in ms for unsatisfiable knight's grid instances with pattern graphs of the form $P_2 \square P_n$.

G	H	McS-SI	McS-SI-LL	McS-SI-AM	Glasgow	RI-DS	VF3	pathLAD
$P_3 \square P_6$	K_7	2	1	3	3	6	11	22
$P_3 \square P_7$	K_8	8	6	20	16	26	152	149
$P_3 \square P_8$	K_9	21	31	34	37	53	475	313
$P_3 \square P_{11}$	K_{10}	111	57	105	165	149	35 515	1 715
$P_3 \square P_{12}$	K_{11}	207	145	331	418	325	117 455	4 047
$P_3 \square P_{13}$	K_{12}	334	298	549	755	752	787 874	7 312
$P_3 \square P_{15}$	K_{13}	569	521	1 032	1 622	2 046	*	19 247
$P_3 \square P_{17}$	K_{14}	1 195	1 114	2 186	3 913	4 807	*	45 530
$P_3 \square P_{21}$	K_{15}	2 243	1 946	4 811	10 312	10 606	*	135 926
$P_3 \square P_{22}$	K_{16}	5 924	5 196	11 880	26 097	36 063	*	308 735
$P_3 \square P_{26}$	K_{17}	51 315	44 537	93 507	357 274	112 509	*	4 808 239
$P_3 \square P_{29}$	K_{18}	144 216	126 328	277 555	1 046 587	371 839	*	*
$P_3 \square P_{33}$	K_{19}	372 099	325 976	770 761	2 485 598	1 524 389	*	*
$P_3 \square P_{35}$	K_{20}	3 752 402	3 279 381	7 163 297	*	8 555 073	*	*
$P_3 \square P_{42}$	K_{21}	*	*	*	*	*	*	*

Table 5.14: Runtimes in ms for unsatisfiable knight's grid instances with pattern graphs of the form $P_3 \square P_n$.

G	H	McS-SI	McS-SI-LL	McS-SI-AM	Glasgow	RI-DS	VF3	pathLAD
$P_4 \square P_5$	K_7	1	1	1	4	6	12	33
$P_4 \square P_6$	K_8	4	3	6	15	18	31	75
$P_4 \square P_7$	K_9	9	8	16	24	47	143	223
$P_4 \square P_9$	K_{10}	32	27	54	140	156	510	921
$P_4 \square P_9$	K_{11}	84	72	182	244	375	1 402	2 128
$P_4 \square P_{11}$	K_{12}	243	209	395	572	803	6 507	6 338
$P_4 \square P_{13}$	K_{13}	417	395	750	1 284	1 903	24 227	11 657
$P_4 \square P_{13}$	K_{14}	829	682	1 560	2 778	4 894	70 695	25 875
$P_4 \square P_{15}$	K_{15}	1 834	1 524	3 451	7 277	13 036	557 027	102 684
$P_4 \square P_{16}$	K_{16}	3 001	2 620	6 402	13 878	33 982	2 016 052	143 635
$P_4 \square P_{18}$	K_{17}	6 700	5 700	13 302	34 386	92 317	*	492 191
$P_4 \square P_{19}$	K_{18}	17 405	16 800	36 632	107 312	309 409	*	1 792 279
$P_4 \square P_{21}$	K_{19}	36 909	33 449	84 628	251 562	1 081 308	*	3 937 353
$P_4 \square P_{21}$	K_{20}	87 626	78 268	200 288	576 556	2 996 995	*	9 786 249
$P_4 \square P_{23}$	K_{21}	219 394	196 354	522 896	1 609 817	*	*	*
$P_4 \square P_{25}$	K_{22}	145 464	133 384	430 028	922 425	*	*	5 036 360
$P_4 \square P_{26}$	K_{23}	1 111 894	988 298	2 944 771	7 615 315	*	*	*
$P_4 \square P_{27}$	K_{24}	1 717 197	1 555 701	5 220 754	*	*	*	*
$P_4 \square P_{29}$	K_{25}	7 288 774	6 578 751	*	*	*	*	*
$P_4 \square P_{30}$	K_{26}	*	*	*	*	*	*	*

Table 5.15: Runtimes in ms for unsatisfiable knight's grid instances with pattern graphs of the form $P_4 \square P_n$.

5.8.4 Enumeration Instances

Our final set of benchmark instances is based on the MIVIA LDGraphs dataset (Carletti et al., 2018). In each LDGraphs instance, the pattern and target graphs are random directed graphs without edge labels. Some of the instances have vertex labels. Although the results of benchmarking graph algorithms on such instances cannot always be extrapolated to real-world instances (McCreesh et al., 2017a), we include these instances because they are the an established benchmark set, and in particular they are the main set of instances used by Carletti et al. (2018) to demonstrate the performance of VF3.

Rather than using the instance files provided by the authors of the MIVIA LDGraphs dataset, we chose to generate our own random graphs from the same model and with the same parameters. We made this choice for two reasons: first, because of the large size of the LDGraphs files (90 GBytes compressed), and second, so that we could augment the set of instances with pairs of sparser graphs.

In each instance, both the pattern and target graph are generated using a directed-graph version of the Erdős-Rényi $G(n, p)$ model. In this model, a graph on n vertices is generated, and each of the $n(n-1)$ possible directed edges is added with independent probability p. Carletti et al. use the values $\{0.2, 0.3, 0.4\}$ for p. In our experiment we additionally use the values 0.05 and 0.1.

Following Carletti et al., we generate two families of directed graphs: an unlabelled family and a family with no edge labels in which the vertex labels are chosen uniformly at random from the set $\{1, \ldots, 8\}$. (Carletti et al. generated a third family, in which labels are chosen from a non-uniform distribution. Their experimental results show very similar outcomes for the uniform and non-uniform families, and therefore we have omitted the non-uniform family from our experiment.)

For each value of p, we use the values of n for the target graph that were used by Carletti et al.; these are shown in Table 5.16. In each instance, a randomly selected subset of 20% of the target vertices is selected, and the subgraph induced by this subset is used as the pattern graph. Thus, each instance is satisfiable (and a typical instance has exactly one solution).

Results for unlabelled instances Figure 5.17 shows run times for the unlabelled instances. Since the instances have a natural ordering by the parameter n, we follow the example of Carletti et al. (2018) and plot average run time as a function of n rather than using cumulative plots. Each point shows, for a given solver and given values of n and p, the arithmetic mean of run time in milliseconds for the 10 instances. The time limit was set to 10000 seconds because many of the larger instances in this experiment are particularly challenging. Runs that exceeded the time limit were counted as taking 10000 seconds; since none of the

p	Target graph n (unlabelled)	Target graph n (labelled)
0.05, 0.1, and 0.2	300, 500, 750, 1000, 1250, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000	300, 500, 750, 1000, 1250, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 9000, 10000
0.3	300, 500, 750, 1000, 1250, 1500, 2000, 2500, 3000	300, 500, 750, 1000, 1250, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 9000
0.4	300, 500, 750, 1000, 1250, 1500	300, 500, 750, 1000, 1250, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000

Table 5.16: Values of n used in enumeration instances

McSplit-SI runs timed out, this does not affect the McSplit-SI results but biases the results slightly in favour of the other algorithms. In cases where all ten runs timed out, the data point is omitted from the plot (as, for example, is the case for RI with $p = 0.4$ and $n = 7000$).

For all five values of p, either McSplit-SI or McSplit-SI-AM is the fastest algorithm. For the smallest two values of p, McSplit-SI is the winner; for $p = 0.2$ and above, McSplit-SI-AM is slightly faster. This is as we would expect, since McSplit-SI is designed to work well on instances with sparse pattern and target graphs. McSplit-SI-LL typically takes around two to three times as long as McSplit-SI to find all solutions—presumably because of the additional work required to maintain sorted linked lists of vertices.

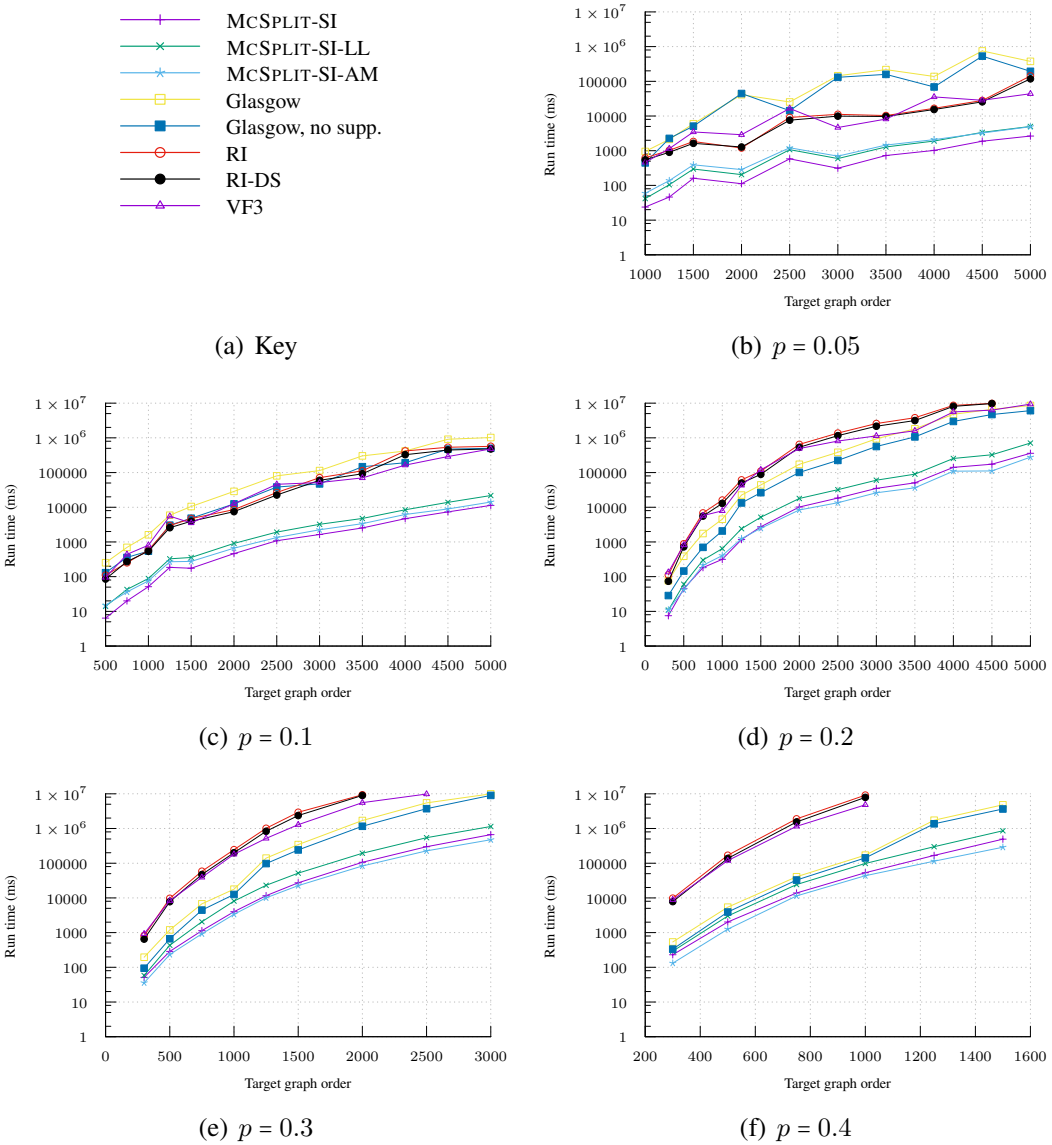
For each value of p, the best variant of McSplit-SI is approximately an order of magnitude faster than the best non-McSplit-SI algorithm. On sparser instances, VF3 and RI-DS have the closest run times to McSplit-SI. Around $p = 0.1$, the Glasgow solver with supplemental graphs disabled becomes competitive with these algorithm, and at $p = 0.4$ it is at least an order of magnitude faster than either VF3 or RI-DS.

When using the Glasgow solver, it is worthwhile to disable supplemental graphs on all but 11 of the instances. In fact, supplemental graphs have no effect on the size of the search tree on the majority of instances, since most pairs of vertices in this set of instances are joined by many paths of length 2; thus, on this set of instances supplemental graphs have a double overhead — in their construction and in their use during search — with no corresponding benefit. (This suggests a potential improvement to the Glasgow solver: if a pair of supplemental graphs are both complete graphs, they should be discarded before search begins because they will provide no domain filtering.)

Results for vertex-labelled instances Figure 5.18 shows run times for instances with labelled vertices. As was the case for the unlabelled instances, McSplit-SI is the fastest solver for sparse instances and McSplit-SI-AM is the fastest solver for dense instances

(with the exception of the smallest dense instances, where McSplit-SI is fastest). VF3 is the fastest non-CP solver for every value of p, and indeed is almost as fast as McSplit-SI for $p = 0.05$. Around $p = 0.3$, the Glasgow solver with supplemental graphs disabled overtakes VF3 as the fastest non-McSplit-SI solver.

The difference between Glasgow with and without supplemental graphs is even larger on these labelled instances than on the labelled instances, exceeding one order of magnitude in many cases.



(a) Key

(b) $p = 0.05$

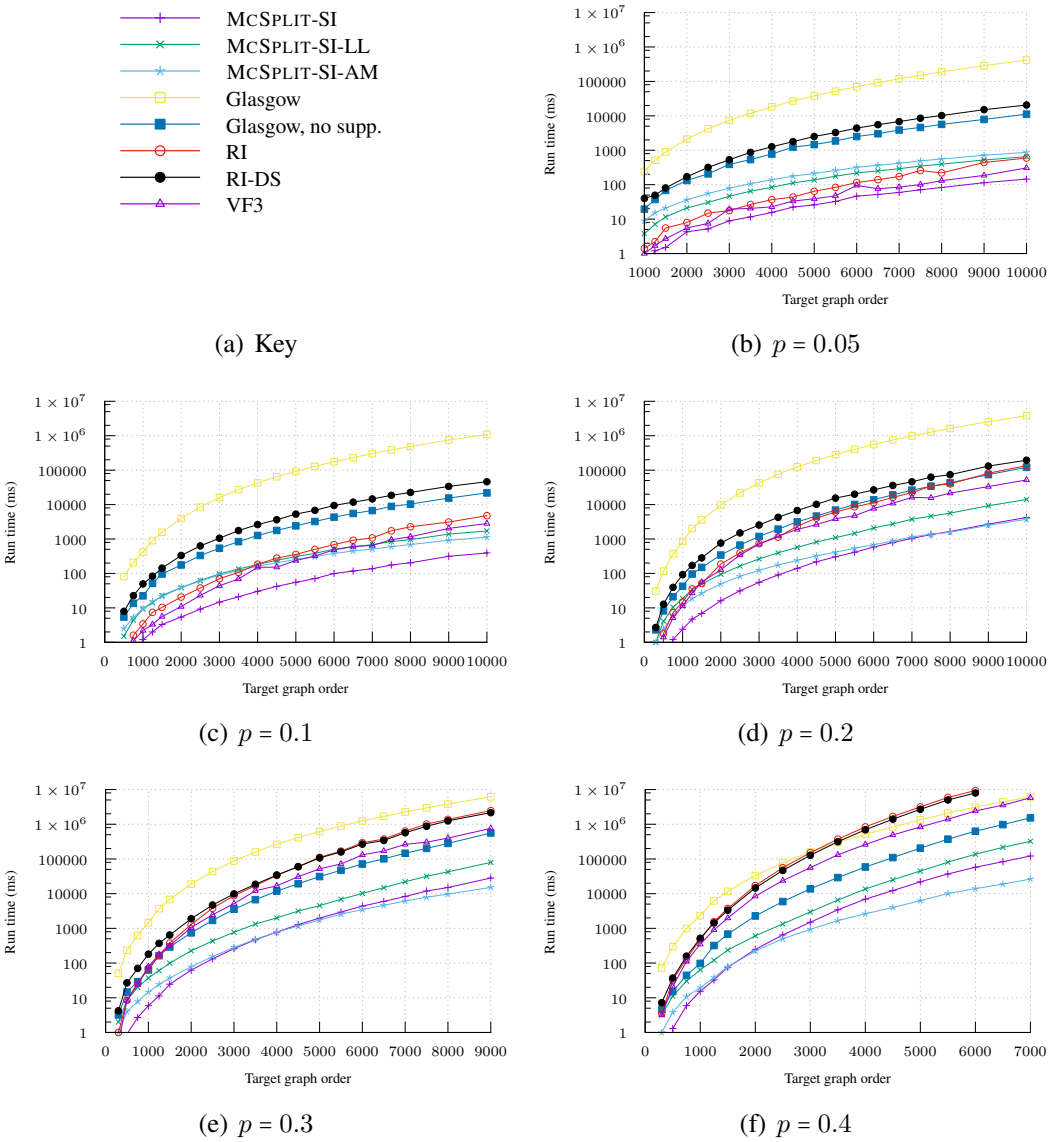
(c) $p = 0.1$

(d) $p = 0.2$

(e) $p = 0.3$

(f) $p = 0.4$

Figure 5.17: Run times on unlabelled enumeration instances with random directed pattern and target graphs



(a) Key

(b) $p = 0.05$

(c) $p = 0.1$

(d) $p = 0.2$

(e) $p = 0.3$

(f) $p = 0.4$

Figure 5.18: Run times on vertex-labelled enumeration instances with random directed pattern and target graphs

5.8.5 Verification

For each of the decision instances, we verified that all of the programs that did not time out agreed on whether the instance was satisfiable or unsatisfiable. For enumeration instances, we verified that the solvers agreed on the number of solutions.

In addition, we checked the number of search nodes visited by McSplit-SI, McSplit-SI-LL, and McSplit-SI-AM on instances where neither solver timed out. In each case, the counts of search nodes were equal, indicating that — as expected — the three solvers explored exactly the same search tree.

I verified that the pseudocode in this chapter accurately describes the McSplit-SI algorithm by re-implementing the algorithm in Python from the pseudocode and checking that it gives the same result and search node count as the original C++ implementation.

I used my own generator to produce the grid and knight graphs. To verify the correctness of the generated graphs, I used the built-in graph generators of SageMath (The Sage Developers, 2020) to generate grid and knight graphs with the same parameters, and used SageMath again to ensure that my graphs were isomorphic to the graphs generated by that program.

5.9 Using the McSplit-SI Data Structures for MCIS

We have so far seen sparse and dense variants of McSplit for induced subgraph isomorphism (McSplit-SI and McSplit-SI-AM), and a dense variant for maximum common induced subgraph (the McSplitalgorithm of Chapter 3). The algorithm that is missing from this list—a sparse variant for MCIS—can be straightforwardly created by modifying McSplit-SI. We call this new algorithm McSplit-sparse.

An implementation of McSplit-sparse requires only a few small changes to the code of McSplit-SI. We therefore give a brief account of the required changes rather than a full description of the algorithm. The McSplit-sparse algorithm uses the same sequence-of-decision-problems strategy as McSplit↓. Thus the call to `Search` on line 21 of Algorithm 9 is replaced by a sequence of calls, one for each goal value of the vertex count of the common subgraph. The `Search` function takes an extra parameter: the current upper bound. The `Filter` function updates this bound each time a label class is partitioned, and returns early if the bound falls below the goal.

5.9.1 Experimental Evaluation of McSplit-Sparse

Following the lead of Archibald et al. (2019), we use the 14,621 subgraph isomorphism instances described in Section 5.8.2 as our benchmark set. This provides a challenging set of instances; in particular, many of the Meshes and Images instances that can be solved in less than a second by McSplit-SI are very difficult for all of the maximum common induced subgraph solvers.

To summarise the results of Archibald et al. (2019) for these instances, McSplit↓ (that is, the earlier, dense version of the algorithm) is slightly faster overall than k↓, but a version of k↓ using *solution-biased search* (which we will call k↓+SBS) is faster than McSplit↓. Solution-biased search uses a randomised value-ordering heuristic, and periodically restarts search to quickly explore many parts of the search tree. New constraints are added on each restart to ensure that parts of the search tree are not revisited.

McSplit-sparse, like McSplit-SI, is a new algorithm that has not appeared in published work. To compare McSplit-sparse with k↓ and k↓+SBS, we used the three algorithms to solve the maximum common subgraph problem for each of the 14,621 instances. The cumulative plot in Figure 5.19 shows that McSplit-sparse is the clear overall winner: it solves 278 more instances than the next best algorithm within the 1000 second time limit, and solves more instances than either variant of k↓ for any per-instance time limit below 1000 seconds.

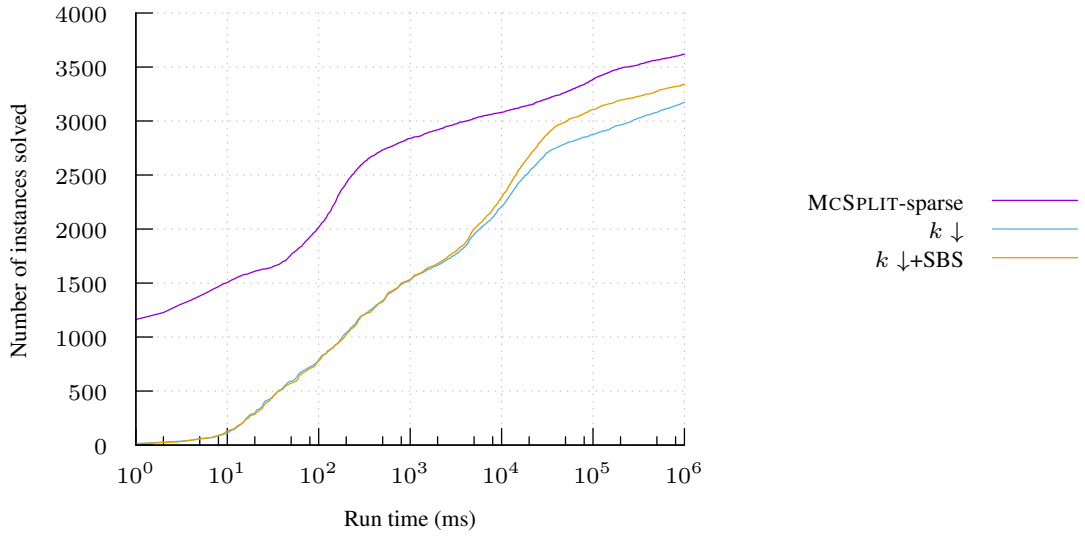
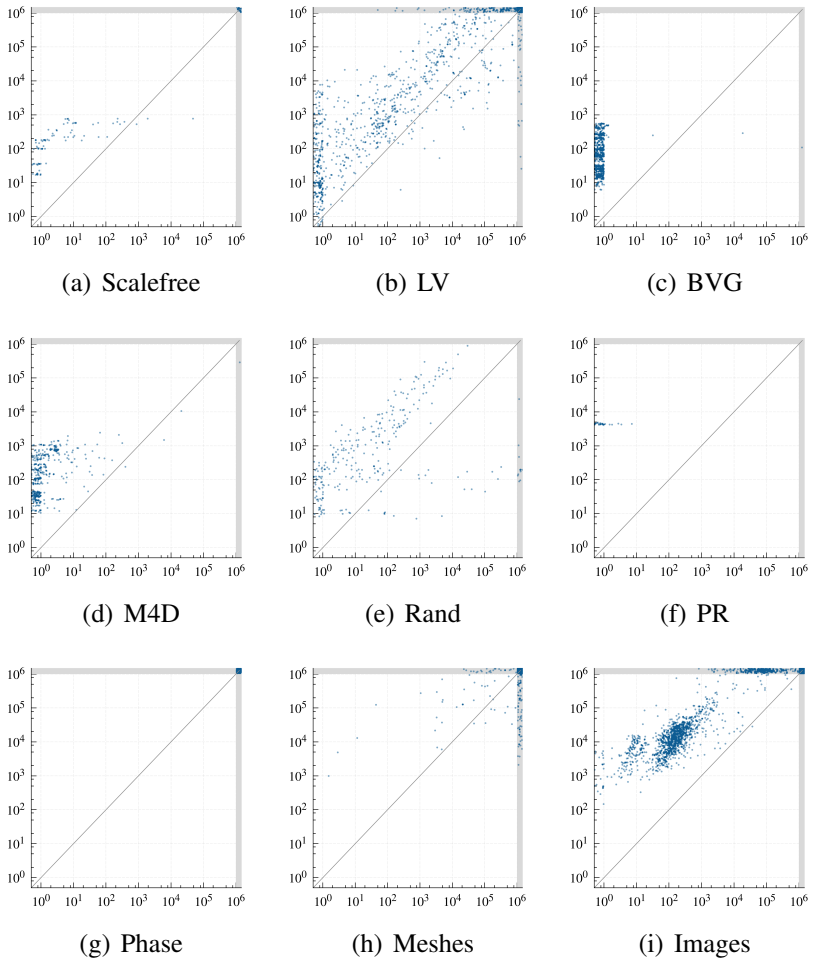


Figure 5.19: Cumulative plot of run times for solving the maximum common induced subgraph problem on the 14,621 instances described in Section 5.8.2

Figure 5.20 shows scatter plots of run times by family, comparing the best two algorithms. Points above the diagonal line represent instances for which McSplit-sparse is faster than k↓+SBS; on all families except Phase and Meshes, many more points lie above

the line than below it. (The Phase instances, all of which are unsatisfiable for the subgraph isomorphism problem, are very challenging for maximum common subgraph solvers. No solver could solve any of these instances within the time limit.)

Looking at the number of instances that were solved within the 1000 second time limit, McSplit-sparse is the clear winner on the Images family, solving 242 more Images instances within the time limit than $k\downarrow$+SBS. McSplit-sparse also performs very well on the LV instances, solving 92 more instances than $k\downarrow$+SBS. On the other hand hand, $k\downarrow$+SBS solves more instances than McSplit-sparse in the BVG, M4D, Meshes and Rand families: a total of 56 more instances across these four families.



(a) Scalefree (b) LV (c) BVG

(d) M4D (e) Rand (f) PR

(g) Phase (h) Meshes (i) Images

Figure 5.20: Run times in ms of McSplit-sparse (horizontal axis) and $k\downarrow$+SBS (vertical axis), by family.

The families where McSplit-sparse performs particularly well—Images and LV—are the largest two families, together accounting for 69% of instances in the benchmark. Is McSplit-sparse's overall success just an artifact of the benchmark's lopsidedness? As a starting point to answer this question, Figure 5.21 repeats the cumulative curve of Figure 5.19, but scales the results as if each family contained 100 instances. Thus each of the 100 Scalefree

instances that is solved contributes, as usual, 1 unit to the y value of a curve, whereas each of the 6278 Images instances that is solved contributes only $100/6278$ units. MCSPLIT-sparse remains the overall winner in this rescaled cumulative plot, but there is only a tiny difference between the rescaled number of instances solved by the two leading solvers within the 1000 second time limit. We could easily construct alternative scalings, such as one that emphasises the Meshes family, that favour $k\downarrow$+SBS overall. While it is clear that MCSPLIT-sparse outperforms the state of the art for many instances—and several families of instances—this rescaling exercise serves as a cautionary reminder that the apparent winner on a cumulative plot can be highly dependent on the composition of the set of benchmark instances used.

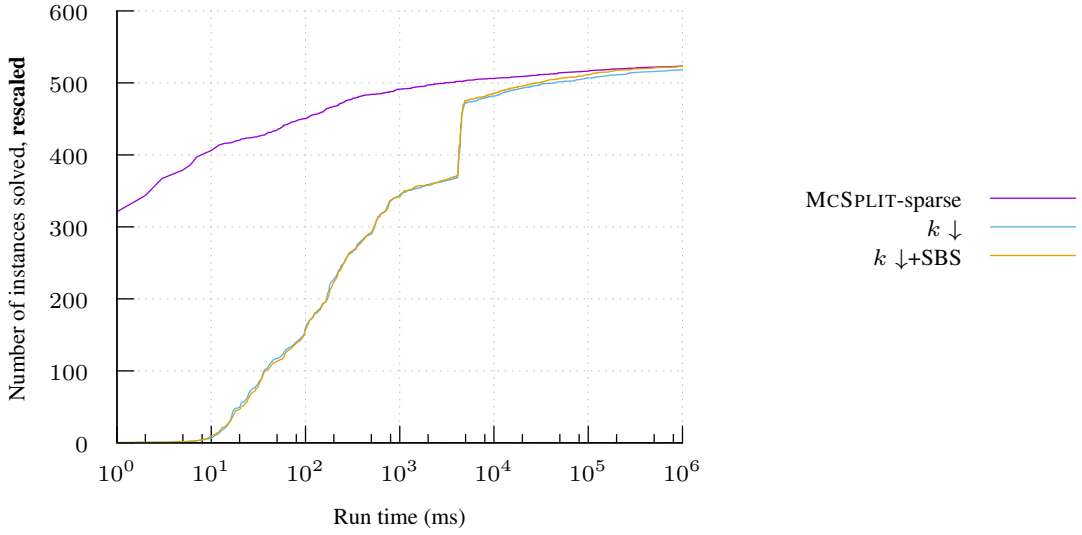


Figure 5.21: **Rescaled** cumulative plot of run times for solving the maximum common induced subgraph problem on the 14,621 instances described in Section 5.8.2

5.10 Conclusion

We have introduced a version of MCSPLIT for the induced subgraph isomorphism problem that is time- and memory-efficient for large, sparse graphs. We have shown experimentally that this algorithm outperforms state-of-the-art algorithms on many instances. Compared to the pattern-recognition solvers RI and VF3, MCSPLIT-SI appears to be a clear improvement on the instances considered, with the exception of some satisfiable knights' grid instances. The situation with respect to the Glasgow Subgraph Solver is more complex, yet MCSPLIT-SI clearly outperforms Glasgow on two of the three experiments, and is a strong competitor even on the hard decision instances where Glasgow performs best.

We have modified the MCSPLIT-SI algorithm to create the MCSPLIT-sparse algorithm for MCIS, and have shown that this outperforms the existing state of the art algorithm $k\downarrow$+SBS on large, sparse graphs.

As a minor contribution of this chapter, our experimental evaluation has brought to light two potential improvements to the Glasgow solver. First, the algorithm could swap the degree-based heuristic (or alternatively, work on complement graphs) if the density of the target graph exceeds 0.5. Second, the algorithm should discard supplemental graphs that can be determined in advance to provide no domain filtering.

Chapter 6

Induced Universal Graphs

6.1 Introduction

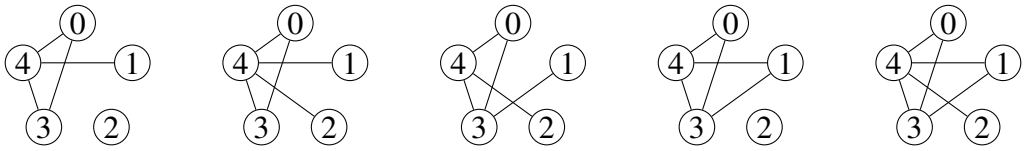
Given a collection $\mathcal{F}$ of graphs, graph G is *induced universal* for $\mathcal{F}$ if and only if each element of $\mathcal{F}$ is isomorphic to an induced subgraph of G. Graph G is a *minimal* induced universal graph for $\mathcal{F}$ if it has as few vertices as possible.[1]

Although much theoretical work has studied the size of minimum induced universal graphs for vertex counts tending to infinity, there has been little effort to develop algorithms for the problem. This chapter introduces exact and heuristic algorithms for finding minimal induced universal graphs, using McSplit-SI-AM as a subroutine. We will use these algorithms to find new terms of integer sequences for families of small graphs, complementing the known asymptotic results.

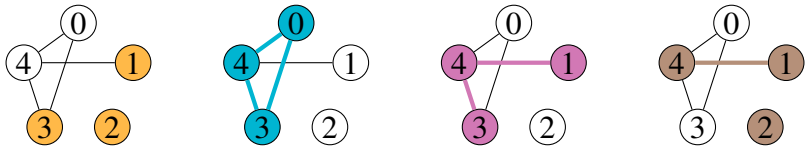
We write $\mathcal{F}(k)$ to denote the family of all graphs on k vertices, and we write $f(k)$ to denote the order (that is, the number of vertices) of a minimal induced universal graph for $\mathcal{F}(k)$. We write $F(k)$ to denote the number of non-isomorphic graphs of order $f(k)$ that are induced universal for $\mathcal{F}(k)$. To give an example, we have $f(3) = 5$ and $F(3) = 5$; all five of the minimal induced universal graphs for $\mathcal{F}(3)$ are shown in Figure 6.1(a). Figure 6.1(b) shows the four graphs in $\mathcal{F}(3)$ as induced subgraphs of a single induced universal graph.

If family $\mathcal{F}$ contains only two graphs, the problem of finding a minimal induced universal graph for $\mathcal{F}$ is the *minimum common supergraph* problem. Bunke et al. (2000) showed that we can find the minimum common supergraph of G and H by finding a maximum common induced subgraph of G and H, then adding to this subgraph each vertex and edge of G and H that does not appear in the subgraph. Thus, the order of a minimum common supergraph of

[1] We use the word *minimal* rather than *minimum* for consistency with the literature. Note that we do not use the word in the weaker sense of an induced universal graph such that removing any vertex would break the property of induced universality.



(a) The five induced universal graphs of order 5 for $\mathcal{F}(3)$ (that is, for the family of all 3-vertex graphs)



(b) A demonstration that the first graph in Figure 6.1(a) is induced universal for $\mathcal{F}(3)$. For each graph G in $\mathcal{F}(3)$ (I_3, K_3, P_3, and a graph with a single edge), an induced subgraph isomorphic to G is shown in a single colour.

Figure 6.1: Minimal induced universal graphs for the family of all graphs on 3 vertices

G and H equals the sum of the orders of G and H minus the order of a maximum common subgraph of G and H. It follows that the minimum common supergraph problem and the more general problem of finding a minimal induced universal graph are NP-hard.

Moon (1965) showed that $f(k) \geq 2^{(k-1)/2}$, and Alon (2017) showed that $f(k) = (1 + o(1))2^{(k-1)/2}$. There is an extensive literature on bounds on the order of minimal induced universal subgraphs for many families of graphs; see the references in Alon (2017). However, to my knowledge the only existing systematic attempt to find *exact* values for families of small graphs is an answer by James Preen on the Mathematics Stack Exchange website that presents results for families of small connected graphs obtained by brute-force search using the Maple programming language (Preen, 2020).

This chapter uses a brute-force approach similar to that of Preen to find minimal induced universal graphs for families of small graphs. Our contributions are (1) an ordering strategy to reduce the number of subgraph isomorphism calls required by the brute force algorithm; (2) exact values for $f(k)$ $(5 \leq k \leq 6)$ and $F(k)$ $(1 \leq k \leq 5)$, contributed to OEIS sequence A097911 (Schwarz, 2004); (3) an upper bound of 18 for $f(7)$; (4) a hill-climbing search algorithm to find small (but not necessarily optimal) induced universal graphs (5) exact values of the order of a minimal induced universal graph for the families of all trees on k vertices, for $1 \leq k \leq 8$, published as OEIS sequence A0348638 (Trimble, 2021).

The sequel is as follows. Section 6.2 describes the brute force method. Section 6.3 compares four methods for sorting the graphs in $\mathcal{F}$, and compares their effect on the number of subgraph isomorphism calls made by the brute-force algorithm. Sections 6.4 presents exact values of $f(k)$ and $F(k)$ for $k \leq 5$. Section 6.5 gives the value of $f(6)$ and Section 6.6 gives bounds on $f(7)$; in each of these sections, the lower bound is proven and a graph obtained by heuristic search demonstrating the upper bound is given. Finally, Section 6.7

describes a specialised algorithm for families of trees, and gives results for these families.

6.2 Generating all Induced Universal Graphs

Given an arbitrary family of graphs $\mathcal{F}$ and a non-negative integer n, we use the brute-force approach shown in Algorithm 15—which is essentially the same as the method described by Preen (2020)—to find the set of n-vertex graphs that are induced universal for the family $\mathcal{F}$. The entry point is the function `AllInducedUniversalGraphs()`. Line 14 tests in turn each candidate graph G from the family of n-vertex graphs, and adds to the collection $\mathcal{G}$ those that are induced universal for $\mathcal{F}$. The function `IsInducedUniversal()` tests whether a graph G is induced universal for $\mathcal{F}$ by checking for each $H \in \mathcal{F}$ that H is isomorphic to an induced subgraph of G.

Algorithm 15: A brute-force algorithm for finding all order-n induced universal subgraphs of a family $\mathcal{F}$ of graphs. The entry point is `AllInducedUniversalGraphs`$(\mathcal{F}, n)$.

```
1  IsInducedUniversal(F, G)
2  Data: A family of graphs F and a graph G.
3  Result: A Boolean value indicating whether G is induced universal for F.
4  begin
5      for H ∈ F do
6          if ¬InducedSubIso(H, G) then return false
7      return true

8  AllInducedUniversalGraphs(F, n)
9  Data: A family of graphs F and a natural number n.
10 Result: The set of all graphs of order n that are induced universal for F.
11 begin
12     G ← ∅
13     for G ∈ F(n) do
14         if IsInducedUniversal(F, G) then G ← G ∪ {G}
15     return G
```

For the collection $\mathcal{F}(n)$ of candidate graphs on line 14, our implementation uses graphs generated using Brendan McKay's `geng` program (McKay, 1998).[2] These are read in `graph6` format[3] from a text file as the program proceeds, and therefore only one candidate graph at a time needs to be stored in memory.

[2]The program `geng` is distributed with Nauty; we used version 27.r3.`https://pallini.di.uniroma1.it/`

[3]`https://users.cecs.anu.edu.au/~bdm/data/formats.html`

Our implementation is written purely in Python and run using the CPython interpreter. On line 6, the function `InducedSubIso()` calls an algorithm for the induced subgraph isomorphism decision problem; for this, our program uses an implementation by the author of the McSplit-SI-AM algorithm. To solve the optimisation problem, we call the function `AllInducedUniversalGraphs`$(\mathcal{F}, n)$ with $n = 0$, $n = 1$, and so on until the returned set of graphs is non-empty.

The full set of experiments described in this chapter, run sequentially, took less than a day to complete on a laptop with an Intel Core i5-6200U CPU and 8 GBytes RAM.[4]

6.3 Iteration Order for $\mathcal{F}$

When determining on line 5 of Algorithm 15 whether a graph G contains every element of $\mathcal{F}$ as an induced subgraph, are some iteration orders of $\mathcal{F}$ better than others? For a graph G that contains a copy of each element of $\mathcal{F}$, the iteration order is of no importance; every element of $\mathcal{F}$ must be checked before $true$ is returned. For a graph G that does *not* contain a copy of each element of $\mathcal{F}$, however, we would like to iterate over $\mathcal{F}$ in an order such that graphs that are likely to fail the subgraph isomorphism test are checked early; that way, we can quickly return $false$ and avoid unnecessary calls to the subgraph isomorphism algorithm.

We begin by considering the case where our family of graphs $\mathcal{F}$ is $\mathcal{F}(k)$ for some k; that is, $\mathcal{F}$ is the set of all graphs of order k. As Chatterjee and Diaconis (2021) note, the complete graph K_n is the 'hardest' graph of order n for a random graph to contain. This suggests that we should use an order that starts with K_k and its complement I_k; we will consider two such orders. The first of these is to sort $\mathcal{F}(k)$ in descending order of automorphism count, so that graphs with large automorphism groups appear first in the list. (Intuitively, if a graph has few automorphisms then we can generate many different labelled graphs by permuting the vertex labels. This gives more 'opportunities' for the graph to be an induced subgraph of G.) The second approach considered is to place graphs with unusually high or low edge counts, as measured by the absolute value of $2|E(G)| - \binom{|V(G)|}{2}$, first.

Figure 6.2 examines whether graphs that appear early in the orderings of $\mathcal{F}$ generated by each of these two strategies are contained in few graphs with a given number of vertices, as we would hope to be the case. For this experiment, we let $\mathcal{F} = \mathcal{F}(5)$, and we consider the problem of finding induced universal graphs of order 8. Each point on the plots represents one of the 34 graphs of order 5 (fewer than 34 points are visible due to exactly overlapping points, as we would expect). On the vertical axis, we have the number of graphs in $\mathcal{F}(8)$ containing G; we would like graphs for which this value is small to appear early in our order.

[4]The code and results from this chapter, including lists of minimal induced universal graphs in graph6 format, are available from `https://github.com/jamestrimble/small-universal-graphs`

On the horizontal axes we have each of our two measures: size of automorphism group in the first plot and 'extremeness' of edge count in the second plot. As we can see from the first plot, the automorphism measure has a strong correlation with the number of graphs containing G as an induced subgraph, suggesting that sorting by this measure could be a good strategy for reducing the number of subgraph isomorphism calls in Algorithm 15.

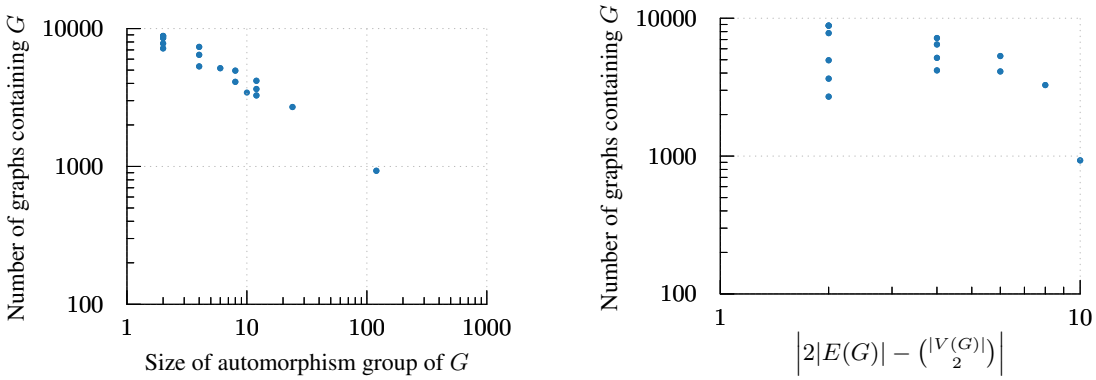


Figure 6.2: For each graph $G \in \mathcal{F}(5)$, we plot on the vertical axis the number of graphs in $\mathcal{F}(8)$ that contain G as an induced subgraph. This value has a strong negative correlation with the size of the automorphism group of G (left plot), but a weaker correlation with a measure of how extreme the edge count of G is (right plot). All axis scales are logarithmic.

6.3.1 Testing the Strategies: $\mathcal{F} = \mathcal{F}(5)$

We now test whether each of our ordering strategies results, as hoped, in a reduced number of calls to the subgraph isomorphism function in Algorithm 15. As our first test case, we consider `AllInducedUniversalGraphs`$(\mathcal{F}(5), 9)$ — finding all induced universal graphs of order 9 for the family of all graphs of order 5. We consider four strategies.

The first two of these are the automorphism-count and 'edge-count extremeness' measures described above. The fourth strategy is a random order. Finally, we used a strategy that we will call "almost random". Under this strategy, I_5 and K_5 are placed at the start of the list, and the remaining 32 graphs are in random order. This strategy was not intended to be useful in practice, but to give insight into whether the success of the first two strategies was purely due to having I_5 and K_5 at the start of the list of graphs. Table 6.1 summarises these ordering strategies.

For each of these strategies, we ran the program 50 times. The results are shown in Figure 6.3. Over 50 runs, the Automorphisms strategy required a mean of 304,746 calls to the subgraph isomorphism solver, while the Edges strategy required marginally fewer calls: a mean of 304,701. Surprisingly, the Almost-Random strategy had the lowest average number of calls of all the strategies, at 304,387. The Random strategy fared poorly, requiring over

Name	Description of strategy
Automorphisms	Sort in descending order of automorphism group size
Edges	Sort in descending order of $\left\lvert 2\lvert E(G)\rvert - \binom{\lvert V(G)\rvert}{2}\right\rvert$ (thus, give priority to graphs that are very sparse or very dense)
Almost-Random	Shuffle the list of graphs in $\mathcal{F}$, then move K_k and I_k to the start if they are present
Random	Shuffle the list of graphs in $\mathcal{F}$

Table 6.1: Ordering strategies tested for the brute-force algorithm

800,000 calls on average. The variance in number of calls was small for all but the Random strategy, with between 304,000 and 305,000 calls being required for each run.

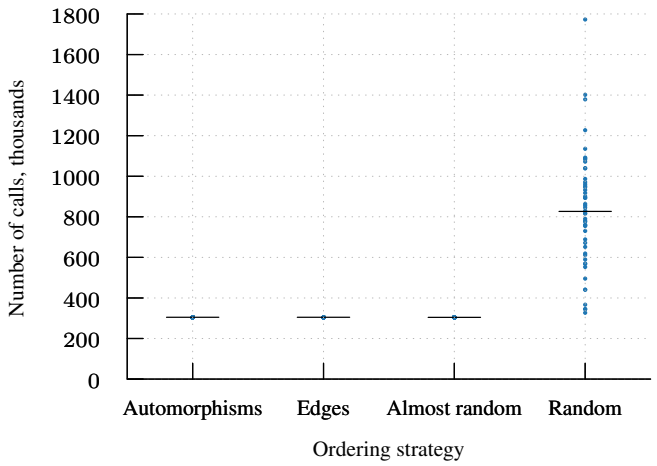


Figure 6.3: Using each of our four order strategies, we ran `AllInducedUniversalGraphs`$(\mathcal{F}(5), 9)$ 50 times. The plot shows the number of calls to the subgraph isomorphism algorithm, in thousands. Horizontal lines show average (mean) number of calls.

We note, first, that the Edges strategy appears to be at least as effective as the Automorphisms strategy for this instance, despite the lower correlation observed in Figure 6.2. Perhaps most surprising is that the Almost-Random strategy performed better than either of these two strategies; our expectation had been that this simple strategy would give some but not all of the benefit of the first two strategies. We conjecture that the Almost-Random strategy is effective because it quickly checks a diverse range of graphs. Consider, for example, Table 6.2, which shows the graphs of order 5 listed in descending order of automorphism count. This is the order in which they are checked under the Automorphisms strategy (with the order within each row chosen randomly). Observe that the two graphs in the second row are similar to the graphs in the first row; the first graphs in each of the first two rows, for example, have a maximum common induced subgraph of size 4. It seems plausible that if a graph contains both graphs in the first row of the table as induced subgraphs, then it is likely also to contain the graphs in the second row. The Almost-Random strategy is likely to choose graphs that are quite different from a clique and an independent set after the first two

graphs, perhaps increasing the probability that one of these graphs will not be contained in the larger graph.

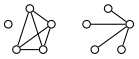
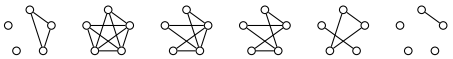
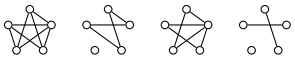
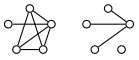
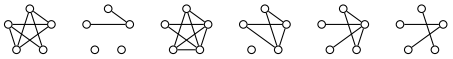
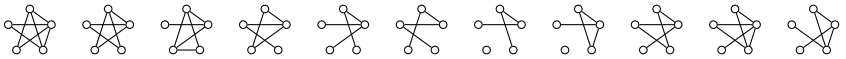
Automorphisms	Graphs
120	
24	
12	
10	
8	
6	
4	
2	

Table 6.2: The 34 graphs of order 5, classified by automorphism group size

6.3.2 Testing the Strategies: $\mathcal{F} \subset \mathcal{F}(5)$

While the Almost-Random strategy was very effective when searching for induced universal graphs of *all* graphs of a given order, it is plausible that it will be less effective if $\mathcal{F}$ is a strict subset of $\mathcal{F}(k)$, and in particular if it includes neither I_k nor K_k. Figure 6.4 examines this claim. We generated 50 subsets $\mathcal{F}$ of $\mathcal{F}(5)$ by random selection, with each subset containing exactly half of the elements of $\mathcal{F}(5)$. For each of these families, we ran `AllInducedUniversalGraphs`$(\mathcal{F}, 9)$ using each of the four strategies. The Automorphisms strategy was the best overall in this case, requiring on average 8%, 27%, and 52% fewer calls to the subgraph isomorphism function than the Edges, Almost-Random, and Random strategies respectively.

To help us understand why the Almost-Random strategy performs poorly for these instances, we connect the four solver runs on each instance together on the plot with blue lines. The Almost-Random strategy, on the instances where it performs poorly, requires exactly the same number of subgraph isomorphism calls as the Random strategy. These are the instances in which $\mathcal{F}$ contains neither K_5 nor I_5; thus, the Almost-Random and Random strategies are equivalent. Notably, for these instances it is also the case that the Automorphisms strategy tends to outperform the Edges strategy.

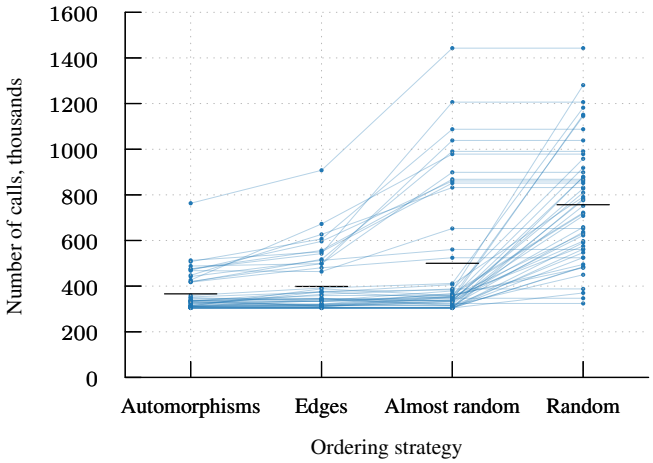


Figure 6.4: Using each of our four order strategies, we ran `AllInducedUniversalGraphs`$(\mathcal{F}, 9)$ 50 times, where $\mathcal{F}$ is a random subset of $\mathcal{F}(5)$ containing 17 graphs. This parallel coordinates plot shows the number of calls to the subgraph isomorphism algorithm, in thousands. Horizontal black lines show average (mean) number of calls. The four runs for each instance are connected by blue lines.

6.3.3 Testing the Strategies: A Family of Trees

As a final test case, we let $\mathcal{F}$ be the family of all trees on 6 vertices, and searched for all graphs of order 8 that are induced universal for $\mathcal{F}$. Figure 6.5 shows the results. The Edges, Almost-Random, and Random strategies had identical results, since all members of $\mathcal{F}$ have five edges and the family contains neither K_6 nor I_6. The Automorphisms strategy requires fewer subgraph isomorphism calls on average than the other three strategies, although on seven of the 50 instances it does require marginally more calls than the other strategies.

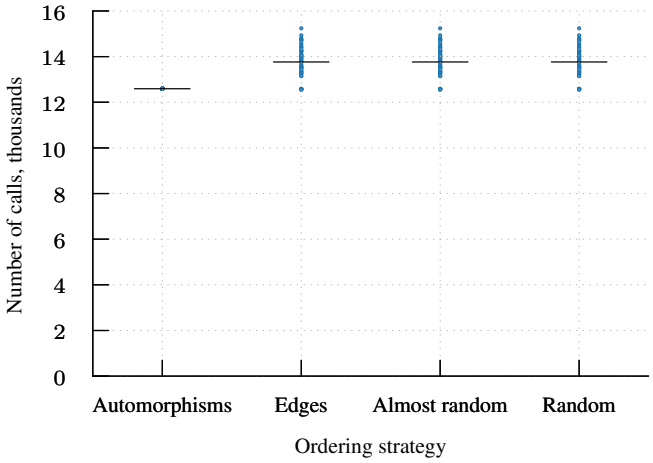


Figure 6.5: Using each of our four order strategies, we ran `AllInducedUniversalGraphs`$(\mathcal{F}, 8)$ 50 times, where $\mathcal{F}$ is the set of all trees of order 6. The plot shows the number of calls to the subgraph isomorphism algorithm, in thousands. Horizontal black lines show average (mean) number of calls.

Since the Automorphisms strategy performs consistently well across our three test cases,

we will use it for the remainder of the chapter.

6.4 Results for $k \leq 5$

Recall that $\mathcal{F}(k)$ is the family of all graphs of order k, $f(k)$ is the order of a minimal induced universal graph for $\mathcal{F}(k)$, and $F(k)$ is the number of non-isomorphic graphs of that order that are induced universal for $\mathcal{F}(k)$. This section presents values of $f(k)$ and $F(k)$ for $0 \leq k \leq 5$, computed using Algorithm 15. The results are shown in Table 6.3. To the author's knowledge, the result $f(5) = 10$ and the values of $F(k)$ have not been published previously.

k	$f(k)$	$F(k)$
0	0	1
1	1	1
2	3	2
3	5	5
4	8	438
5	10	22

Table 6.3: For each k, $f(k)$ is the minimum order of a graph containing all k-vertex graphs as induced subgraphs, and $F(k)$ is the number of distinct $f(k)$-vertex graphs that contain all k-vertex graphs as induced subgraphs.

The values $f(1)$, $f(2)$ and $f(3)$ are equal to the simple lower bound $2k - 1$ given in a question by the user "Chain Markov" on Mathematics Stack Exchange (Chain Markov, 2019). The values $f(4)$ and $f(5)$ are equal to a lower bound given in a comment by a user named "bof" on the same question: $f(k) \geq 2k$ if $k \geq 4$. (For $k < 10$, this bound improves upon Moon's lower bound $f(k) \geq 2^{(k-1)/2}$.) To briefly summarise the proof, if $f(k) \leq 2k$ then G must be a split graph (that is, a graph whose vertices can be partitioned into a clique and an independent set); therefore, G cannot contain the cycle C_4 as an induced subgraph. An example of an 8-vertex induced universal graph for the family of 4-vertex graphs was given by "Chain Markov".

As Table 6.3 shows, there are 438 minimal induced universal graphs for $\mathcal{F}(4)$, and 22 minimal induced universal graphs for $\mathcal{F}(5)$. Table 6.4 (which extends over the next three pages) and Table 6.5 show the edge counts and degree distributions of these graphs, along with an example of a graph with each degree distribution.

Edges	Degrees	Count	Example	Edges	Degrees	Count	Example
11	0 1 2 3 4 4 4 4	1		11	0 1 3 3 3 4 4 4	1	
11	0 2 2 2 3 3 5 5	1		11	0 2 2 2 3 4 4 5	1	
11	0 2 2 3 3 3 4 5	1		12	0 1 3 3 4 4 4 5	1	
12	0 1 3 4 4 4 4 4	2		12	0 2 2 3 3 4 5 5	1	
12	0 2 2 3 4 4 4 5	4		12	0 2 3 3 3 3 5 5	2	
12	0 2 3 3 3 4 4 5	2		12	0 2 3 3 4 4 4 4	1	
12	1 1 2 3 4 4 4 5	6		12	1 1 2 4 4 4 4 4	1	
12	1 1 3 3 3 4 4 5	2		12	1 1 3 3 4 4 4 4	2	
12	1 2 2 2 3 3 5 6	1		12	1 2 2 2 3 4 4 6	1	
12	1 2 2 2 3 4 5 5	3		12	1 2 2 2 4 4 4 5	3	
12	1 2 2 3 3 3 4 6	1		12	1 2 2 3 3 3 5 5	2	
12	1 2 2 3 3 4 4 5	3		12	1 2 2 3 4 4 4 4	1	
12	1 2 3 3 3 3 4 5	1		12	2 2 2 2 3 3 4 6	1	
12	2 2 2 2 3 4 4 5	1		13	0 1 4 4 4 4 4 5	1	
13	0 2 3 3 4 4 4 6	2		13	0 2 3 3 4 4 5 5	3	
13	0 2 3 4 4 4 4 5	2		13	0 3 3 3 3 3 5 6	1	
13	0 3 3 3 3 4 5 5	1		13	1 1 2 4 4 4 5 5	1	
13	1 1 3 3 4 4 4 6	2		13	1 1 3 3 4 4 5 5	5	
13	1 1 3 4 4 4 4 5	5		13	1 1 4 4 4 4 4 4	1	
13	1 2 2 3 3 4 5 6	1		13	1 2 2 3 3 5 5 5	1	
13	1 2 2 3 4 4 4 6	4		13	1 2 2 3 4 4 5 5	14	
13	1 2 2 4 4 4 4 5	6		13	1 2 3 3 3 3 5 6	3	
13	1 2 3 3 3 4 4 6	4		13	1 2 3 3 3 4 5 5	12	
13	1 2 3 3 4 4 4 5	8		13	1 2 3 4 4 4 4 4	1	
13	1 3 3 3 3 3 5 5	1		13	1 3 3 3 3 4 4 5	1	
13	2 2 2 2 3 3 6 6	1		13	2 2 2 2 3 4 5 6	2	

13	2 2 2 2 3 5 5 5	1	
13	2 2 2 2 4 4 5 5	2	
13	2 2 2 3 3 4 4 6	5	
13	2 2 2 3 4 4 4 5	2	
13	2 2 3 3 3 3 5 5	1	
14	0 2 4 4 4 4 4 6	1	
14	1 1 3 4 4 5 5 5	1	
14	1 1 4 4 4 4 5 5	2	
14	1 2 3 3 4 4 4 7	2	
14	1 2 3 3 4 5 5 5	9	
14	1 2 3 4 4 4 5 5	14	
14	1 3 3 3 3 3 5 7	1	
14	1 3 3 3 3 4 5 6	4	
14	1 3 3 3 4 4 5 5	5	
14	2 2 2 3 3 4 6 6	1	
14	2 2 2 3 4 4 5 6	9	
14	2 2 2 4 4 4 4 6	2	
14	2 2 3 3 3 3 6 6	2	
14	2 2 3 3 3 5 5 5	5	
14	2 2 3 3 4 4 5 5	8	
14	2 3 3 3 3 3 5 6	1	
14	2 3 3 3 3 4 5 5	2	
15	1 2 3 4 4 5 5 6	1	
15	1 2 4 4 4 4 4 7	1	
15	1 2 4 4 4 5 5 5	3	
15	1 3 3 3 4 4 6 6	2	
15	1 3 3 3 5 5 5 5	1	
15	1 3 3 4 4 5 5 5	5	
13	2 2 2 2 4 4 4 6	1	
13	2 2 2 3 3 3 5 6	3	
13	2 2 2 3 3 4 5 5	5	
13	2 2 3 3 3 3 4 6	1	
13	2 2 3 3 3 4 4 5	2	
14	0 3 3 3 4 4 5 6	2	
14	1 1 4 4 4 4 4 6	1	
14	1 2 2 4 4 5 5 5	1	
14	1 2 3 3 4 4 5 6	16	
14	1 2 3 4 4 4 4 6	4	
14	1 2 4 4 4 4 4 5	1	
14	1 3 3 3 3 3 6 6	1	
14	1 3 3 3 3 5 5 5	2	
14	1 3 3 4 4 4 4 5	1	
14	2 2 2 3 3 5 5 6	1	
14	2 2 2 3 4 5 5 5	6	
14	2 2 2 4 4 4 5 5	5	
14	2 2 3 3 3 4 5 6	14	
14	2 2 3 3 4 4 4 6	5	
14	2 2 3 4 4 4 4 5	2	
14	2 3 3 3 3 4 4 6	1	
15	1 1 4 4 5 5 5 5	1	
15	1 2 3 4 5 5 5 5	2	
15	1 2 4 4 4 4 5 6	3	
15	1 3 3 3 4 4 5 7	2	
15	1 3 3 3 4 5 5 6	4	
15	1 3 3 4 4 4 5 6	4	
15	1 3 4 4 4 4 5 5	1	

15	2 2 2 4 4 5 5 6	1	
15	2 2 3 3 3 5 6 6	1	
15	2 2 3 3 4 4 6 6	5	
15	2 2 3 3 5 5 5 5	2	
15	2 2 3 4 4 4 5 6	12	
15	2 2 4 4 4 4 4 6	1	
15	2 3 3 3 3 3 6 7	1	
15	2 3 3 3 3 4 6 6	5	
15	2 3 3 3 4 4 5 6	8	
15	2 3 3 4 4 4 4 6	1	
15	3 3 3 3 3 3 6 6	1	
16	1 2 4 4 5 5 5 6	1	
16	1 3 4 4 4 5 5 6	1	
16	2 2 3 4 4 5 5 7	1	
16	2 2 4 4 4 4 5 7	2	
16	2 3 3 3 4 4 6 7	1	
16	2 3 3 3 4 5 6 6	6	
16	2 3 3 4 4 4 5 7	2	
16	2 3 3 4 4 5 5 6	3	
16	2 3 4 4 4 4 5 6	1	
16	3 3 3 3 3 5 6 6	1	
16	3 3 3 3 4 4 6 6	2	
17	2 2 4 4 5 5 5 7	1	
17	2 3 4 4 4 5 5 7	1	
17	3 3 3 4 4 4 6 7	1	

15	2 2 2 4 5 5 5 5	1	
15	2 2 3 3 4 4 5 7	3	
15	2 2 3 3 4 5 5 6	14	
15	2 2 3 4 4 4 4 7	1	
15	2 2 3 4 4 5 5 5	5	
15	2 2 4 4 4 4 5 5	1	
15	2 3 3 3 3 4 5 7	2	
15	2 3 3 3 3 5 5 6	6	
15	2 3 3 3 4 5 5 5	2	
15	2 3 3 4 4 4 5 5	2	
15	3 3 3 3 3 4 5 6	1	
16	1 3 3 4 5 5 5 6	1	
16	1 3 4 4 5 5 5 5	1	
16	2 2 3 4 5 5 5 6	3	
16	2 2 4 4 4 5 5 6	2	
16	2 3 3 3 4 5 5 7	4	
16	2 3 3 3 5 5 5 6	3	
16	2 3 3 4 4 4 6 6	2	
16	2 3 3 4 5 5 5 5	1	
16	3 3 3 3 3 4 6 7	2	
16	3 3 3 3 4 4 5 7	1	
16	3 3 3 3 4 5 5 6	1	
17	2 3 3 4 5 5 5 7	1	
17	3 3 3 3 4 5 6 7	1	

Table 6.4: There are 438 minimal induced universal graphs for $\mathcal{F}(4)$, and this set of 438 graphs has 157 distinct degree distributions. For each degree distribution, this table shows the number of edges, the degree distribution, the number of graphs with that distribution, and an example graph.

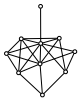
Edges	Degrees	Count	Example
21	0 2 4 4 4 4 5 5 7 7	1	
21	0 3 3 4 4 4 5 5 7 7	1	
22	1 2 3 4 5 5 5 5 7 7	1	
22	1 2 3 4 5 5 5 6 6 7	2	
22	1 2 4 4 4 4 5 5 7 8	1	
22	1 2 4 4 4 4 5 6 7 7	1	
22	1 3 3 4 4 4 5 5 7 8	2	
22	1 3 3 4 4 4 5 6 7 7	2	
23	1 2 4 4 5 5 5 5 7 8	1	
23	1 2 4 4 5 5 5 6 6 8	2	
23	2 2 3 4 5 5 5 5 7 8	1	
23	2 2 3 4 5 5 5 6 6 8	2	
23	2 2 4 4 4 4 5 6 7 8	1	
23	2 3 3 4 4 4 5 6 7 8	2	
24	2 2 4 4 5 5 5 5 7 9	1	
24	2 2 4 4 5 5 5 6 6 9	1	

Table 6.5: There are 22 minimal induced universal graphs for $\mathcal{F}(5)$, and this set of 22 graphs has 16 distinct degree distributions. For each degree distribution, this table shows the number of edges, the degree distribution, the number of graphs with that distribution, and an example graph.

6.5 $f(6) = 14$

This section shows that $f(6) = 14$. We begin with the lower bound. For $k \geq 6$, we can increase by 2 the lower bound — $f(k) \geq 2k$ for $k \geq 4$ — cited in the previous section. This gives the following proposition, which implies that $f(6) \geq 14$.

Proposition 3. *$f(k) \geq 2k + 2$ for all $k \geq 6$.*

Proof. Suppose that G is an induced universal graph for the family of all graphs on k vertices, and that G has no more than $2k + 1$ vertices. Graphs K_k and I_k are both members of $\mathcal{F}(k)$, and therefore must be induced subgraphs of G. Clearly, this clique and independent set may overlap by no more than one vertex, so their union must contain at least $2k - 1$ vertices. Therefore it is possible to partition the vertices of G into three sets: a clique S_1, an independent set S_2, and a third set S_3 containing at most 2 vertices.

We will show that G cannot contain as induced subgraphs both of the graphs in Figure 6.6. These graphs are complements of each other, and we refer to them as H and H' respectively.

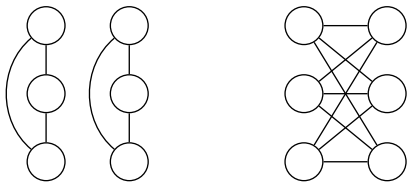


Figure 6.6: If $k > 5$ and G is a graph on $2k + 1$ or fewer vertices, then G cannot have as induced subgraphs all four of the following graphs: the clique K_k, the independent set I_k, and the two graphs shown (which we call H and H'). See the proof of Proposition 3.

Let G_1 be an induced subgraph of G that is isomorphic to H. Since there are no edges between the two three-vertex cliques in G_1, it must be the case that the vertex set of at least one of these cliques does not intersect S_1. Since S_2 is an independent set in G, this clique must have exactly one vertex in S_2 and two vertices in S_3. We can deduce, then, that S_3 contains exactly two vertices, and that these vertices are adjacent in G.

Now consider graph H'. Since H' is an induced subgraph of G, it follows by taking complements of H' and G that H is an induced subgraph of G' (the complement of G). We can repeat the argument of the previous paragraph with the roles of S_1 and S_2 reversed to show that the two vertices in S_3 must be adjacent in the complement of G, and therefore must not be adjacent in G. Since we previously showed that these vertices are adjacent in G, we have a contradiction. □

To give an upper bound on $f(6)$, we use the local search algorithm in Algorithm 16, which finds an induced universal graph for $\mathcal{F}(6)$. The entry point to the algorithm is the function `FindInducedUniversalGraph()`, which takes as its parameter the maximum number of attempted improvement steps to take before restarting from scratch. The main search function, `FindInducedUniversalGraph'()` is called repeatedly until a given time limit is reached.

The function `FindInducedUniversalGraph'()` begins by generating a random graph G on 14 vertices as follows (see Figure 6.7). Numbering the vertices from zero, we

Algorithm 16: A hill-climbing algorithm to find an induced universal graph for family $\mathcal{F}(6)$

```
1  FlipEdge(G, v, w)
2  begin
3      if G has edge {v, w} then
4          return (V(G), E(G) ∖ {v, w})
5      else
6          return (V(G), E(G) ∪ {v, w})

7  Score(G)
8  begin
9      return |{H ∈ F(6) | H is an induced subgraph of G}|

10 FindInducedUniversalGraph'(maxIter)
11 begin
12     G ← ({0, ..., 13}, ∅)  ▷ A graph on 14 vertices with no edges
13     Add edges to make vertices {0, ..., 5} of G a clique
14     Add to G each possible edge marked · in Figure 6.7 with probability 1/2
15     for i ∈ {1, ... maxIter} do
16         v, w ← vertices corresponding to a randomly selected position marked · in
             Figure 6.7
17         G' ← FlipEdge(G, v, w)
18         if Score(G') = |F(6)| then return G'  ▷ Success!
19         if Score(G') ≥ Score(G) then G ← G'
20     return null

21 FindInducedUniversalGraph(maxIter)
22 begin
23     while time limit is not exceeded do
24         G ← FindInducedUniversalGraph'(maxIter)
25         if G ≠ null then return G
26     return null
```

make the k vertices numbered 0 to $k-1$ a clique, and the k vertices numbered $k-1$ to $2k-2$ an independent set.[5]

Each possible edge that is not involved in either the clique or the independent set — shown in the figure with a dot — is added with probability $1/2$. The loop in lines 15 to 19 then carries out up to $maxIter$ iterations of a step that randomly modifies graph G, and accepts the modification if it causes the number of graphs in $\mathcal{F}(6)$ that are induced subgraphs of G to increase. The attempted-improvement step chooses a random pair of vertices $\{v, w\}$, both

[5]The clique and the independent set thus have one vertex in common. The proof of Proposition 3 can be modified straightforwardly to show that there is no 14-vertex induced universal graph for this family of graphs that contains a 6-vertex clique and a 6-vertex independent set as induced subgraphs with disjoint vertex sets.

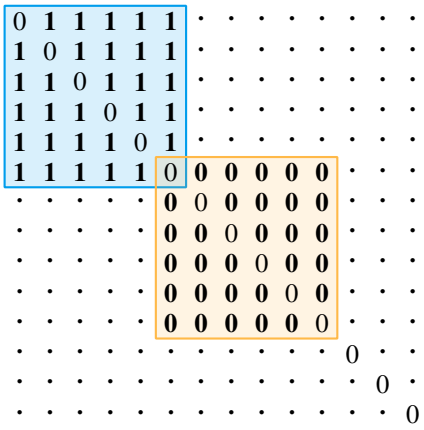


Figure 6.7: The adjacency matrix used in our heuristic algorithm when searching for an induced universal graph of order 14 for $\mathcal{F}(6)$. The blue region (a 6-vertex clique) and the yellow region (a 6-vertex independent set) remain unchanged as the algorithm runs. The remaining possible edges, marked ·, are initially set to 0 or 1 at random, and are flipped between 0 and 1 as the algorithm progresses.

of which are not in the same coloured square in Figure 6.7, and "flips" the status of the edge between these two vertices: the edge is added if it is not present, and removed otherwise. After each flip, we count the number of graphs on 6 vertices (from a total of 156 graphs) that are isomorphic to an induced subgraph of our 14-vertex graph. If the most recent flip increased this number or left it unchanged, we accept the modification to the graph. After $maxIter$ flips, we restart the algorithm with a new randomly generated graph to avoid getting stuck at a bad local optimum.

Our implementation has two optimisations that are not shown in Algorithm 16. First, scores are cached in a dictionary data structure, each key of which is the adjacency matrix of a graph flattened to a one-dimensional string of bits. This allows us to quickly look up the score of a graph that has already been visited. This data structure is cleared on return from `FindInducedUniversalGraph'`, in order to bound the program's memory use. (An alternative approach, which we have not implemented but which may be useful in order to explore more of the space of possible graphs, would be to use a tabu list (Glover and Laguna, 1997) to avoid revisiting graphs that have been visited recently.)

The second optimisation often avoids the need to visit every graph in $\mathcal{F}(6)$ during calls to `Score()`. Rather than beginning our count at 0 and incrementing it for every graph in $\mathcal{F}(6)$ that is an induced subgraph of G, we begin our count at $|\mathcal{F}(6)|$ and decrement it for every graph that is *not* an induced subgraph of G. As soon as the counter falls below the score that was obtained prior to the most recent edge-flip, we can return early from the function with a score of -1. This does not affect the behaviour of the hill-climbing search, since any score below the previous graph's score causes the most recent change to be rejected.

By running this search algorithm for less than one minute, we found the graph of order

14 whose adjacency matrix is shown in Figure 6.8. This is induced universal for $\mathcal{F}(6)$; therefore, $f(6) \leq 14$. Combining our two bounds, we have $f(6) = 14$.

```
0 1 1 1 1 1 0 1 1 0 0 0 1 0
1 0 1 1 1 1 1 1 1 1 1 0 0 1
1 1 0 1 1 1 0 1 1 0 1 0 1 0
1 1 1 0 1 1 1 0 0 0 1 1 0 0
1 1 1 1 0 1 0 1 0 0 0 0 0 0
1 1 1 1 1 0 0 0 0 0 0 1 0 1
0 1 0 1 0 0 0 0 0 0 0 1 1 1
1 1 1 0 1 0 0 0 0 0 0 1 1 0
1 1 1 0 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0 1 0 1
0 1 1 1 0 0 0 0 0 0 0 1 1 0
0 0 0 1 0 1 1 1 0 1 1 0 0 1
1 0 1 0 0 0 1 1 1 0 1 0 0 0
0 1 0 0 0 1 1 0 0 1 0 1 0 0
```

Figure 6.8: The adjacency matrix of a 14-vertex induced universal graph for the family of all six-vertex graphs

6.6 Bounds on $f(7)$

By Proposition 3, we have $f(7) \geq 16$. Figure 6.9 shows the adjacency matrix of an 18-vertex induced universal graph for the family of all seven-vertex graphs. This was generated with the heuristic described in Section 6.5, with two modifications. First, 10,000 rather than 1000 edge-flips were permitted before each restart, as this was found to be more effective in a preliminary run of the experiment. Second, the overlapping six-vertex clique and independent set were replaced with a clique on seven vertices and an independent set on seven vertices. Again, these had one vertex in common. (We also tried making the clique and independent set vertex-disjoint, but did not find an 18-vertex solution in four hours with this approach.) The algorithm found a solution in slightly over 21 minutes.

Thus we have $16 \leq f(7) \leq 18$.

```
0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0
1 0 1 1 1 1 1 0 1 1 1 0 1 1 0 1 0 0
1 1 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0 1
1 1 1 0 1 1 1 0 1 0 0 1 0 0 0 0 0 1
1 1 1 1 0 1 1 1 1 0 0 0 0 0 1 0 0 0
1 1 1 1 1 0 1 0 1 0 0 0 0 1 1 1 1 1
1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1
1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 1 0
1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1
1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0
1 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
1 1 0 0 0 1 1 0 1 0 1 0 0 0 0 0 0 1
1 0 0 0 1 1 0 1 1 1 0 1 0 0 0 0 0 1
1 1 0 0 0 1 0 1 1 1 0 1 1 0 0 0 1 1
0 0 0 0 0 1 0 1 0 1 1 1 1 0 0 1 0 1
0 0 1 1 0 1 1 0 1 1 0 0 1 1 1 1 1 0
```

Figure 6.9: The adjacency matrix of an 18-vertex induced universal graph for the family of all seven-vertex graphs

We also ran the heuristic with target graph orders of 16 and 17, in an attempt to find a smaller induced universal graph for $\mathcal{F}(7)$, but the algorithm did not find a solution within 12 hours for either target.

6.7 Trees

In this section, we find minimal induced universal graphs for families of all trees on k vertices, for small k. Alstrup et al. (2017) give a very strong related asymptotic result, proving by construction that there exists a graph of order $O(k)$ that is induced universal for the family of *forests* on k vertices. Unfortunately, no implementation of this construction exists. This section will show that for $n \leq 10$, there exists an induced universal graph with no more than $2n - 2$ vertices for the family of n-vertex trees.

For given k, we tried two approaches to generate all minimal induced universal graphs for the family of all trees on k vertices. The first approach was to run Algorithm 15, with the family $\mathcal{F}$ of order-k trees loaded from a list published by Brendan McKay.[6] The second approach, described in the following subsection, generates only candidate graphs that contain a star — one of the elements of $\mathcal{F}$ —as an induced subgraph. The final part of this section gives results generated using these two methods, along with additional upper bounds generated using a specialised version of our hill-climbing algorithm.

6.7.1 The Completion Method

In our second approach to finding induced universal graphs for the family of all trees on k vertices, we embed the star $K_{1,k-1}$ — which must be present since it is a tree on k vertices — in the top left position of the adjacency matrix, then systematically try all possible ways to complete the adjacency matrix. For each completed adjacency matrix, we test whether the graph contains as an induced subgraph each tree in our family. If it does, and if the graph is not isomorphic to any induced universal graph that has already been found, we add it to our collection of graphs. We will refer to this as the "naive completion" method.

Because the naive completion method has to test a huge number of graphs, many pairs of which are isomorphic, we created an improved version of the algorithm, which we will refer to as the "symmetry breaking completion" method. Our description refers to the adjacency matrix in Figure 6.10. This adds two symmetry-breaking constraints on the adjacency matrix. The first of these constraints concerns the grey shaded region at the bottom-right of the adjacency matrix, which has $n - k$ rows and columns. Rather than trying all $2^{\binom{n-k}{2}}$ possible

[6] https://users.cecs.anu.edu.au/~bdm/data/trees.html

ways to fill this region of the adjacency matrix, we use a list containing a canonical adjacency matrix for each possible graph of order $n - k$, and we require that the shaded region be equal to one of these adjacency matrices.

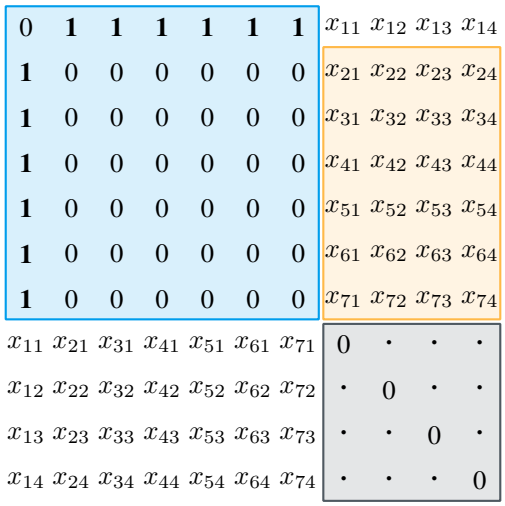


Figure 6.10: The adjacency-matrix regions used in the "symmetry breaking completion" method. For this example, $n = 11$ and $k = 7$.

The second symmetry-breaking constraint concerns the yellow shaded region near the top right of Figure 6.10. Each row of this region has $n - k$ entries, named $x_{i1}, \ldots, x_{i,n-k}$. For our symmetry-breaking constraint, we view each row as the binary number $x_{i1}, \ldots, x_{i,n-k}$ (with $x_{i,n-k}$ as the least significant digit), and we insist that these numbers are in non-decreasing order; thus $x_{21}, \ldots, x_{2,n-k} \leq \cdots \leq x_{k1}, \ldots, x_{k,n-k}$. In other words, we impose a lexicographic ordering constraint on the rows of this region.

Algorithm 17 shows the algorithm that generates and tests each adjacency matrix that satisfies these conditions. The call to `MakeGraph()` on line 14 constructs a graph by building its adjacency matrix as shown in Figure 6.10. The first k vertices are connected in the form of a star; for each i such that $1 \leq i \leq j$, the digits of the binary representation of x_i are assigned to $x_{i1}, \ldots, x_{i,n-k}$, and these values are placed in the adjacency matrix as shown in the figure. Finally, the small adjacency matrix M is copied to the bottom-right of the adjacency matrix of the constructed graph.

For the collection of graphs $\mathcal{G}$ defined on line 5, isomorphic graphs are considered identical. Thus, we only add a graph to $\mathcal{G}$ if it is not isomorphic to any existing member of the collection.[7]

We claim that although the symmetry breaking completion method does not visit every

[7] An efficient method to ensure that isomorphic graphs are not stored in $\mathcal{G}$ would be label the vertices of a graph G canonically before adding it to the collection (McKay and Piperno, 2014), and to store the collection as a hash set. Because this part of the program is not a performance bottleneck, and to minimise software dependencies, our implementation uses the simpler approach of testing whether G is isomorphic to each existing member of $\mathcal{G}$, in turn.

Algorithm 17: The symmetry-breaking completion algorithm for finding all graphs of order n that are induced universal for the family of all order-k trees

```
Search(n, k, (x_1, ..., x_j))
Data: Natural numbers n and k, and a sequence (x_1, ..., x_j)) of nonnegative
      integers such that the binary representation of each x_i represents entries
      x_{i1}, ..., x_{i,n-k} of an adjacency matrix, as shown in Figure 6.10
Result: The set of all order-n graphs that are induced universal for the family of
        trees of order k, and have an adjacency matrix given by
        MakeGraph(n, k, X, M) for some completion X of (x_1, ..., x_j) and some
        M ∈ M(n − k)
begin
    G ← ∅
    if j ≤ 1 then
        for i ∈ {0, ..., 2^{n−k} − 1} do
            G ← G ∪ Search(n, p, (x_1, ..., x_j, i))
    else if j ≤ k then
        for i ∈ {0, ..., x_j} do
            G ← G ∪ Search(n, p, (x_1, ..., x_j, i))
    else
        for M ∈ M(n − k) do
            G ← MakeGraph(n, k, (x_1, ..., x_j), M)
            if G contains every tree of order k as an induced subgraph then
                G ← G ∪ {G}
    return G

FindAllInducedUniversalGraphsForTrees(n, k)
Data: Natural numbers n and k such that k < n
Result: The set of all graphs of order n that are induced universal for the family of
        trees of order k.
begin
    return Search(n, k, ())
```

adjacency matrix visited by the naive completion method, it finds, up to isomorphism, all induced universal graphs for the family of trees on k vertices. This is shown by the following proposition.

Proposition 4. *Let n, k be given, and let G be a graph on n vertices that is induced universal for the family of trees of order k. Let $\mathcal{M}$ be a set of canonical adjacency matrices for $\mathcal{F}(n-k)$. There is a graph on n vertices numbered $1, \ldots, n$ that is isomorphic to G and satisfies the two symmetry-breaking constraints described above.*

Proof. Since G is induced universal for the family of trees of order k, it contains the star $K_{1,k-1}$ as an induced subgraph. Arbitrarily choose one such star in G and give the label 1

to its vertex of degree $k-1$. Its remaining vertices will be given the labels $\{2,\ldots,k\}$, in an order that will be specified in the next paragraph. The remaining $n-k$ vertices of G induce a subgraph isomorphic to one of the adjacency matrices in $\mathcal{M}$; call this matrix M. By choosing an appropriate relabelling of the $n-k$ vertices of G, the region of the adjacency matrix corresponding to these vertices can be made equal to M. Thus, the first symmetry-breaking constraint is satisfied.

The second symmetry-breaking constraint can now be satisfied by assigning labels $\{2,\ldots,k\}$ to the degree-1 vertices of the star $K_{1,k-1}$ in such an order that their adjacencies to the vertices outside the star, viewed as binary numbers, are in lexicographic order. □

6.7.2 Results for Families of Trees

The symmetry-breaking completion method is faster than Algorithm 15. For $n = 9$ and $k = 6$, for example, the completion method takes 9 seconds whereas the brute-force algorithm takes 46 seconds. The difference in run times becomes even greater for larger values of n and k. Using the symmetry-breaking completion method, we were able to find the values of $t(k)$ and $T(k)$ for $k \leq 7$. We were also able to show that $t(8) > 12$. We confirmed the results for $k \leq 6$ using Algorithm 15.

Table 6.6 gives the order $t(k)$ of a minimal induced universal graph for the family of k-vertex trees, and the number $T(k)$ of such graphs. Figure 6.11 shows one of the 687 minimal induced universal graphs for the family of 7-vertex trees.

k	$t(k)$	$T(k)$
1	1	1
2	2	1
3	3	1
4	5	2
5	7	18
6	9	66
7	11	687

Table 6.6: For each k, $t(k)$ is the minimum order of a graph containing all k-vertex trees as induced subgraphs, and $T(k)$ is the number of distinct $t(k)$-vertex graphs that contain all k-vertex trees as induced subgraphs.

To obtain upper bounds on $t(k)$ for larger values of k, we use a version of Algorithm 16 with minor modifications. Instead of initially embedding a clique and an independent set as in Figure 6.7, we embed a single tree — the k-vertex star $K_{1,k-1}$ — as shown in Figure 6.12. Each possible edge marked with a dot in the figure is added with probability 0.1.

Using this heuristic algorithm, we were able to find induced universal graphs to demonstrate that $t(8) \leq 13$, $t(9) \leq 15$ and $t(10) \leq 18$. Since we have matching lower and upper

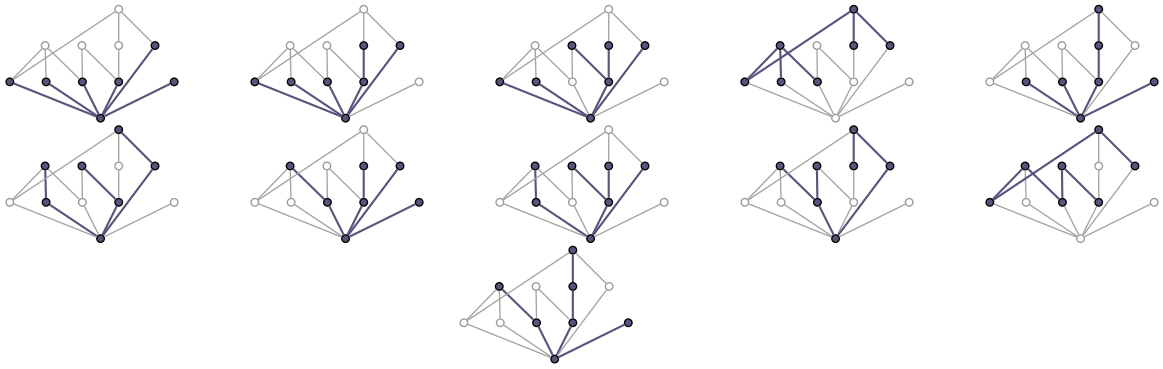


Figure 6.11: An induced universal graph for the family of all trees on 7 vertices. In each copy of the graph, one of the 11 trees of order 7 is highlighted.

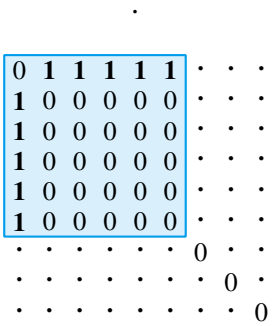


Figure 6.12: The adjacency matrix used in our heuristic algorithm when searching for an induced universal graph of order 9 for the set of all trees of order 6. The blue region, a star with 6 vertices, remains unchanged as the algorithm runs. The remaining possible edges, marked ·, are flipped between 0 and 1 as the algorithm progresses.

bounds for $t(8)$, we can state that $t(8) = 13$.

6.8 Verification

To verify correctness of the results in Table 6.3 and Table 6.6, we repeated every call to McSplit-SI-AM using the LAD algorithm (Solnon, 2010) as implemented in the `igraph` library (Csardi and Nepusz, 2006).

We verified that the graphs shown as adjacency matrices in Figure 6.8 and Figure 6.9 are induced universal graphs for their corresponding families using a shell script that calls the Glasgow Subgraph Solver (McCreesh and Prosser, 2015; McCreesh et al., 2020) to check for the inclusion of every graph on k vertices. This script shares no code with the Python program that generated the graphs.

Finally, we verified all but the final row in each of Table 6.3 and Table 6.6 using a second shell script that calls the Glasgow Subgraph Solver. Again, this shares no code with the program used to generate the results in the tables.

6.9 Conclusion

In this chapter, we have introduced exact and heuristic methods for finding induced universal graphs for a given family of graphs. Using these algorithms, we have found two new terms for the sequence $f(k)$ and the first eight terms of the sequence $t(k)$.

Future work could apply the techniques from this chapter to find the minimum size of a induced universal graph for families such as forests or graphs or bounded degree, complementing the known asymptotic results for these families (Alstrup et al., 2017; Abrahamsen et al., 2017).

Chapter 7

Conclusion

7.1 Summary

This dissertation has introduced the McSplit family of algorithms: McSplit for maximum common induced subgraph (MCIS) and McSplit-SI for the induced subgraph isomorphism problem (ISIP). We have seen the close relationship between these algorithms and both constraint programming and clique algorithms. We have also seen how the algorithms' data structures, combined with the special properties of the problems, give us good upper bounds for MCIS and generalised arc consistency on the all-different constraint in ISIP at almost no computational cost.

The McSplit algorithms outperform the existing state of the art on many families of benchmark instances. In the case of MCIS, this is particularly true for unlabelled, undirected instances; the clique encoding remains the best approach for instances with many labels on vertices and edges. For induced subgraph isomorphism, McSplit-SI is the fastest solver for many classes of graphs, including both random graphs and some classes of structured graphs such as the new knight's grid instances proposed by Knuth. But McSplit-SI is not the best ISIP solver for every instance—for some pairs of structured graphs from the Stanford GraphBase, for example, supplemental graphs give the Glasgow solver an advantage.

In addition to the new solvers presented in this dissertation, our study of algorithms for MCIS and ISIP yields the following simple conclusions which should be applicable to a broad range of current and future solvers.

- Branch and bound is not always the best optimisation strategy for MCIS. We have seen that McSplit↓, which solves a sequence of decision problems, can be orders of magnitude faster than the branch-and-bound solver McSplit, particularly on instances where the maximum common induced subgraph is almost as large as the input graphs.

- It can be helpful to swap the two input graphs before calling an MCIS solver, particularly if the graphs differ in order or density.

- In both MCIS and ISIP, choosing the best strategy for sorting the vertices of each graph is a non-trivial problem, and the decision should take into account the properties of the graphs. It can be helpful, for example, to sort in decreasing order of degree if the target graph is sparse and in increasing order of degree if the target graph is dense.

Turning from subgraphs to supergraphs, we have applied McSplit-SI to the problem of finding a small induced universal graph—that is, a graph that contains every member of a given set of graphs as an induced subgraph. This problem has received much theoretical attention, but there is almost no prior work on the development of solvers for small instances. We have developed new exact and heuristic algorithms for the problem and used these to generate new terms that have been included in the On-Line Encyclopedia of Integer Sequences.

7.2 Future Work

The strength of McSplit — its compact data structure that enables fast and simple filtering algorithms — has the disadvantage that it places limits on how the algorithm can be modified. For example, there appears to be no obvious way to add supplemental graphs to McSplit-SI because they would break the invariant that two domains must be equal or disjoint. Despite this limitation, many promising avenues remain for future development of the algorithms. For example:

- We could explore new variable and value ordering heuristics. Other authors have already shown that reinforcement learning can be used to improve the heuristics in McSplit (Liu et al., 2020; Trummer, 2021). The present author, in preliminary work not presented here, has found that a version of the dom/deg heuristic (Bessière and Régin, 1996) solves more of the ISIP instances in Section 5.8.2 than the smallest-domain-first approach that we have used.

- We could use new search strategies beyond depth-first traversal of the search tree. Initial work on using solution-biased search (Archibald et al., 2019) for McSplit showed promising results, and this could be extended to McSplit-SI.

- We could develop new swapping rules for McSplit like those in Chapter 4, using characteristics of the graphs other than order and degree.

- We could create a portfolio solver that chooses an appropriate solver for each instance from a set of solvers including McSplit. This has already been done successfully for MCIS by (Dilkas, 2018); future work could extend this by adding the swapping versions of McSplit from Chapter 4. There does not yet exist a portfolio solver for ISIP that uses McSplit-SI.
- We could solve the maximum common *edge* subgraph by using McSplit to solve MCIS on the line graphs of the two input graphs (Raymond and Willett, 2002). The same method could be used to solve subgraph monomorphism using McSplit-SI.

Finally, it would be interesting to look for other constraint satisfaction problems in which many domains and constraints are equal or similar, for which we might use a McSplit-like data structure as a domain store.

Bibliography

Abrahamsen, M., Alstrup, S., Holm, J., Knudsen, M. B. T., and Stöckel, M. (2017). Near-optimal induced universal graphs for bounded degree graphs. In Chatzigiannakis, I., Indyk, P., Kuhn, F., and Muscholl, A., editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 128:1–128:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Abu-Khzam, F. N., Bonnet, É., and Sikora, F. (2017). On the complexity of various parameterizations of common induced subgraph isomorphism. *Theor. Comput. Sci.*, 697:69–78.

Alon, N. (2017). Asymptotically optimal induced universal graphs. *Geometric and Functional Analysis*, 27(1):1–32.

Alstrup, S., Dahlgaard, S., and Knudsen, M. B. T. (2017). Optimal induced universal graphs and adjacency labeling for trees. *J. ACM*, 64(4):27:1–27:22.

Archibald, B., Dunlop, F., Hoffmann, R., McCreesh, C., Prosser, P., and Trimble, J. (2019). Sequential and parallel solution-biased search for subgraph algorithms. In Rousseau, L. and Stergiou, K., editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 20–38. Springer.

Armitage, J. E. and Lynch, M. F. (1967). Automatic detection of structural similarities among chemical compounds. *Journal of the Chemical Society C: Organic*, pages 521–528.

Audemard, G., Lecoutre, C., Modeliar, M. S., Goncalves, G., and Porumbel, D. C. (2014). Scoring-based neighborhood dominance for the subgraph isomorphism problem. In O'Sullivan, B., editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 125–141. Springer.

Babai, L. (2016). Graph isomorphism in quasipolynomial time [extended abstract]. In Wichs, D. and Mansour, Y., editors, *Proceedings of the 48th Annual ACM SIGACT Symposium*

on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016, pages 684–697. ACM.

Bahiense, L., Manic, G., Piva, B., and de Souza, C. C. (2012). The maximum common edge subgraph problem: A polyhedral investigation. *Discrete Applied Mathematics*, 160(18):2523–2541.

Balas, E. and Yu, C. S. (1986). Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15(4):1054–1068.

Barabási, A.-L. (2003). *Linked - how everything is connected to everything else and what it means for business, science, and everyday life*. Plume.

Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512.

Bessière, C. and Régin, J.-C. (1996). MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Freuder, E. C., editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 of *Lecture Notes in Computer Science*, pages 61–75. Springer.

Bonnici, V. and Giugno, R. (2017). On the variable ordering in subgraph isomorphism algorithms. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 14(1):193–203.

Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D. E., and Ferro, A. (2013). A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.*, 14(S-7):S13.

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

Bunke, H., Foggia, P., Guidobaldi, C., Sansone, C., and Vento, M. (2002). A comparison of algorithms for maximum common subgraph on randomly connected graphs. In Caelli, T., Amin, A., Duin, R. P. W., Kamel, M. S., and de Ridder, D., editors, *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshops SSPR 2002 and SPR 2002, Windsor, Ontario, Canada, August 6-9, 2002, Proceedings*, volume 2396 of *Lecture Notes in Computer Science*, pages 123–132. Springer.

Bunke, H., Jiang, X., and Kandel, A. (2000). On the minimum common supergraph of two graphs. *Computing*, 65(1):13–25.

Carletti, V., Foggia, P., Saggese, A., and Vento, M. (2018). Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. *IEEE Trans. Pattern Anal. Mach. Intell.*, 40(4):804–818.

Chain Markov (2019). What is the minimal possible size of an n-universal graph? Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/3246874 (version: 2019-06-01).

Chatterjee, S. and Diaconis, P. (2021). Isomorphisms between random graphs. *CoRR*, abs/2108.04323. `http://arxiv.org/abs/2108.04323`.

Cheng, B. M. W., Choi, K. M. F., Lee, J. H., and Wu, J. C. K. (1999). Increasing constraint propagation by redundant modeling: an experience report. *Constraints An Int. J.*, 4(2):167–192.

Conte, D., Foggia, P., and Vento, M. (2007). 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.

Cook, W. J., Coullard, C. R., and Turán, G. (1987). On the complexity of cutting-plane proofs. *Discret. Appl. Math.*, 18(1):25–38.

Cortadella, J. and Valiente, G. (2000). A relational view of subgraph isomorphism. In Desharnais, J., editor, *Participants Copies of Fifth International Seminar on Relational Methods in Computer Science,January 9-14, 2000, Valcartier (Quebec), Canada*, pages 45–54.

Csardi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9.

Dalke, A. (2012). Fmcs source code. Available at `https://github.com/rdkit/rdkit/blob/5526fab5465789bdf7bec484b86a39b697d3a172/rdkit/Chem/fmcs/fmcs.py`.

Dalke, A. and Hastings, J. (2013). FMCS: a novel algorithm for the multiple MCS problem. *J. Cheminformatics*, 5(S-1):6.

Damiand, G., Solnon, C., de la Higuera, C., Janodet, J., and Samuel, É. (2011). Polynomial algorithms for subisomorphism of nD open combinatorial maps. *Comput. Vis. Image Underst.*, 115(7):996–1010.

Delorme, M., García, S., Gondzio, J., Kalcsics, J., Manlove, D. F., Pettersson, W., and Trimble, J. (2022). Improved instance generation for kidney exchange programmes. *Comput. Oper. Res.*, 141:105707.

Dickerson, J. P., Manlove, D. F., Plaut, B., Sandholm, T., and Trimble, J. (2016). Position-indexed formulations for kidney exchange. In Conitzer, V., Bergemann, D., and Chen, Y., editors, *Proceedings of the 2016 ACM Conference on Economics and Computation, EC '16, Maastricht, The Netherlands, July 24-28, 2016*, pages 25–42. ACM.

Dilkas, P. (2018). Algorithm selection for maximum common subgraph. Bachelor's thesis, University of Glasgow.

Durand, P. J., Pasari, R., Baker, J. W., and Tsai, C.-c. (1999). An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2(17):1–16.

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

Foggia, P., Sansone, C., and Vento, M. (2001). A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199.

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

Gay, S., Fages, F., Martinez, T., Soliman, S., and Solnon, C. (2014). On the subgraph epimorphism problem. *Discrete Applied Mathematics*, 162:214–228.

Geelen, P. A. (1992). Dual viewpoint heuristics for binary constraint satisfaction problems. In Neumann, B., editor, *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pages 31–35. John Wiley and Sons.

Gent, I. P., Miguel, I., and Nightingale, P. (2008). Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.*, 172(18):1973–2000.

Gent, I. P., Petrie, K. E., and Puget, J. (2006). Symmetry in constraint programming. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier.

Glover, F. W. and Laguna, M. (1997). *Tabu Search*. Kluwer.

Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., and Trimble, J. (2020a). Certifying solvers for clique and maximum common (connected) subgraph problems. In Simonis, H., editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer.

Gocht, S., McCreesh, C., and Nordström, J. (2020b). Subgraph isomorphism meets cutting planes: Solving with certified solutions. In Bessiere, C., editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1134–1140. `ijcai.org`.

Golomb, S. W. and Baumert, L. D. (1965). Backtrack programming. *J. ACM*, 12(4):516–524.

Hagberg, A. A., Schult, D. A., and Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. In Varoquaux, G., Vaught, T., and Millman, J., editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA.

Harvey, W. D. and Ginsberg, M. L. (1995). Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 607–615. Morgan Kaufmann.

Hitotumatu, H. and Noshita, K. (1979). A technique for implementing backtrack algorithms and its application. *Inf. Process. Lett.*, 8(4):174–175.

Hoffmann, R., McCreesh, C., Ndiaye, S. N., Prosser, P., Reilly, C., Solnon, C., and Trimble, J. (2018). Observations from parallelising three maximum common (connected) subgraph algorithms. In van Hoeve, W. J., editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings*, volume 10848 of *Lecture Notes in Computer Science*, pages 298–315. Springer.

Hoffmann, R., McCreesh, C., and Reilly, C. (2017). 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.

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. W., editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York.

Katsirelos, G. and Bacchus, F. (2005). Generalized nogoods in CSPs. In Veloso, M. M. and Kambhampati, S., editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 390–396. AAAI Press / The MIT Press.

Knuth, D. E. (1993). *The Stanford GraphBase - a platform for combinatorial computing*. ACM.

Knuth, D. E. (2020). The art of computer programming. volume 4, fascicle 5. mathematical preliminaries redux; introduction to backtracking; dancing links.

Knuth, D. E. (2022). The art of computer programming. volume 4, pre-fascicle 7b. constraint satisfaction (very preliminary draft).

Koch, I. (2001). Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1-2):1–30.

Kotthoff, L., McCreesh, C., and Solnon, C. (2016). 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.

Krissinel, E. B. and Henrick, K. (2004). Common subgraph isomorphism detection by backtracking search. *Softw. Pract. Exp.*, 34(6):591–607.

Land, A. H. and Doig, A. G. (2010). An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer.

Larrosa, J. and Valiente, G. (2002). Constraint satisfaction algorithms for graph pattern matching. *Math. Struct. Comput. Sci.*, 12(4):403–422.

Lecoutre, C., Sais, L., Tabary, S., and Vidal, V. (2007). Recording and minimizing nogoods from restarts. *J. Satisf. Boolean Model. Comput.*, 1(3-4):147–167.

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

Li, C., Fang, Z., Jiang, H., and Xu, K. (2018). Incremental upper bound for the maximum clique problem. *INFORMS J. Comput.*, 30(1):137–153.

Liu, Y., Li, C., Jiang, H., and He, K. (2020). A learning based branch and bound for maximum common subgraph related problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 2392–2399. AAAI Press.

López-Presa, J. L. and Anta, A. F. (2009). Fast algorithm for graph isomorphism testing. In Vahrenhold, J., editor, *Experimental Algorithms, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings*, volume 5526 of *Lecture Notes in Computer Science*, pages 221–232. Springer.

Manlove, D. F., McBride, I., and Trimble, J. (2017). "Almost-stable" matchings in the hospitals / residents problem with couples. *Constraints An Int. J.*, 22(1):50–72.

Marenco, J. (1999). *Un algoritmo branch-and-cut para el problema de mapping*. PhD thesis, Master's thesis, Universidad de Buenos Aires, 1999.

McCreesh, C. (2017). *Solving hard subgraph problems in parallel*. PhD thesis, University of Glasgow.

McCreesh, C. (2022). Personal communication.

McCreesh, C., Ndiaye, S. N., Prosser, P., and Solnon, C. (2016a). Clique and constraint models for maximum common (connected) subgraph problems. In Rueher, M., editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 350–368. Springer.

McCreesh, C. and Prosser, P. (2015). A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In Pesant, G., editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 295–312. Springer.

McCreesh, C., Prosser, P., Simpson, K. A., and Trimble, J. (2017a). On maximum weight clique algorithms, and how they are evaluated. In Beck, J. C., editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 206–225. Springer.

McCreesh, C., Prosser, P., Solnon, C., and Trimble, J. (2018). When subgraph isomorphism is really hard, and why this matters for graph databases. *J. Artif. Intell. Res.*, 61:723–759.

McCreesh, C., Prosser, P., and Trimble, J. (2016b). Heuristics and really hard instances for subgraph isomorphism problems. In Kambhampati, S., editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 631–638. IJCAI/AAAI Press.

McCreesh, C., Prosser, P., and Trimble, J. (2016c). Morphing between stable matching problems. In Rueher, M., editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 832–840. Springer.

McCreesh, C., Prosser, P., and Trimble, J. (2017b). A partitioning algorithm for maximum common subgraph problems. In Sierra, C., editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 712–719. ijcai.org.

McCreesh, C., Prosser, P., and Trimble, J. (2020). The Glasgow Subgraph Solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In Gadducci, F. and Kehrer, T., editors, *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, volume 12150 of *Lecture Notes in Computer Science*, pages 316–324. Springer.

McGregor, J. J. (1979). Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3):229–250.

McGregor, J. J. (1982). Backtrack search algorithms and the maximal common subgraph problem. *Softw., Pract. Exper.*, 12(1):23–34.

McKay, B. D. (1998). Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324.

McKay, B. D. and Piperno, A. (2014). Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112.

Moon, J. W. (1965). On minimal n-universal graphs. *Proceedings of the Glasgow Mathematical Association*, 7(1):32–33.

Moore, C. and Mertens, S. (2011). *The Nature of Computation*. Oxford University Press.

Ndiaye, S. N. and Solnon, C. (2011). CP models for maximum common subgraph problems. In Lee, J. H., editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 637–644. Springer.

Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*. Oxford University Press.

Olatunbosun, S., Dowling, G. R., and Ellis, T. J. (1996). Topological representation for matching coloured surfaces. In *Proceedings 1996 International Conference on Image Processing, Lausanne, Switzerland, September 16-19, 1996*, pages 1019–1022. IEEE Computer Society.

Petit, T., Régin, J.-C., and Bessière, C. (2001). Specific filtering algorithms for over-constrained problems. In Walsh, T., editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 451–463. Springer.

Preen, J. (2020). What is the smallest graph that contains all non-isomorphic 4-node and 5-node connected graphs as induced subgraphs? Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/75513 (version: 2020-04-25).

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

Prosser, P. (2017). Personal communication.

Quer, S., Marcelli, A., and Squillero, G. (2020). The maximum common subgraph problem: A parallel and multi-engine approach. *Comput.*, 8(2):48.

Raymond, J. W., Gardiner, E. J., and Willett, P. (2002a). RASCAL: calculation of graph similarity using maximum common edge subgraphs. *Comput. J.*, 45(6):631–644.

Raymond, J. W., Gardiner, E. J., and Willett, P. (2002b). Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45(6):631–644.

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

Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In Hayes-Roth, B. and Korf, R. E., editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, pages 362–367. AAAI Press / The MIT Press.

Régin, J.-C. (1995). *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique*. PhD thesis, Université Montpellier.

Rossi, F., van Beek, P., and Walsh, T., editors (2006). *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier.

San Segundo, P., Rodríguez-Losada, D., and Jiménez, A. (2011). An exact bit-parallel algorithm for the maximum clique problem. *Comput. Oper. Res.*, 38(2):571–581.

Santo, M. D., Foggia, P., Sansone, C., and Vento, M. (2003). A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079.

Schmidt, D. C. and Druffel, L. E. (1976). A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23(3):433–445.

Schwarz, D. (2004). The On-Line Encyclopedia of Integer Sequences. A097911: Minimal order of a graph containing as induced subgraphs isomorphic copies of all graphs on n unlabeled nodes.

Segundo, P. S., Furini, F., and Artieda, J. (2019). A new branch-and-bound algorithm for the maximum weighted clique problem. *Comput. Oper. Res.*, 110:18–33.

Smith, B. M. (2006). Modelling. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 377–406. Elsevier.

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

Solnon, C. (2019). Experimental evaluation of subgraph isomorphism solvers. In Conte, D., Ramel, J., and Foggia, P., editors, *Graph-Based Representations in Pattern Recognition - 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19-21, 2019, Proceedings*, volume 11510 of *Lecture Notes in Computer Science*, pages 1–13. Springer.

Solnon, C., Damiand, G., de la Higuera, C., and Janodet, J. (2015). On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognit.*, 48(2):302–316.

Sussenguth, E. H. (1965). A graph-theoretic algorithm for matching chemical structures. *Journal of Chemical Documentation*, 5(1):36–43.

Syslo, M. (1982). The induced subgraph isomorphism problem for series-parallel graphs is NP-complete. Technical Report CS 82-095.

Takahashi, Y., Satoh, Y., Suzuki, H., and Sasaki, S.-i. (1987). Recognition of largest common structural fragment among a variety of chemical structures. *Analytical sciences*, 3(1):23–28.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*. John Wiley & Sons.

The Sage Developers (2020). *SageMath, the Sage Mathematics Software System (Version 9.0)*. `https://www.sagemath.org`.

Tomita, E., Sutani, Y., Higashi, T., and Wakatsuki, M. (2013). A simple and faster branch-and-bound algorithm for finding a maximum clique with computational experiments. *IEICE Trans. Inf. Syst.*, 96-D(6):1286–1298.

Trimble, J. (2020a). An algorithm for the exact treedepth problem. In Faro, S. and Cantone, D., editors, *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*, volume 160 of *LIPIcs*, pages 19:1–19:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Trimble, J. (2020b). PACE solver description: Bute-plus: A bottom-up exact solver for treedepth. In Cao, Y. and Pilipczuk, M., editors, *15th International Symposium on Parameterized and Exact Computation, IPEC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 180 of *LIPIcs*, pages 34:1–34:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Trimble, J. (2020c). PACE solver description: Tweed-plus: A subtree-improving heuristic solver for treedepth. In Cao, Y. and Pilipczuk, M., editors, *15th International Symposium on Parameterized and Exact Computation, IPEC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 180 of *LIPIcs*, pages 35:1–35:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Trimble, J. (2021). The On-Line Encyclopedia of Integer Sequences. A348638: Minimal order of a graph containing as induced subgraphs isomorphic copies of all trees on n unlabeled nodes.

Trimble, J., McCreesh, C., and Prosser, P. (2018). Three new approaches for the maximum common edge subgraph problem. *CP 2018 Doctoral Program Proceedings*, pages 52–61. `https://cp2018.a4cp.org/dp-proceedings.pdf`.

Trummer, J. (2021). Engineering maximum common subgraph algorithms for large graphs. Master's thesis, Universität Wien.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42.

Vazirani, V. V. (2001). *Approximation algorithms*. Springer.

Vismara, P., Régin, J.-C., Quinqueton, J., Py, M., Laurenço, C., and Lapied, L. (1992). Resyn : Un système d'aide à la conception de plans de synthèse en chimie organique. In *Actes de la 12e conférence d'intelligence artificielle*, volume 1, pages 305–318. EC2.

Vismara, P. and Valery, B. (2008). 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.

Whitney, H. (1932). Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168.

Zampelli, S. (2008). *A constraint programming approach to subgraph isomorphism*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium.

Zampelli, S., Deville, Y., and Dupont, P. (2007). Symmetry breaking in subgraph pattern matching. *Trends in Constraint Programming*, pages 203–218.

Zampelli, S., Deville, Y., and Solnon, C. (2010). Solving subgraph isomorphism problems with constraint programming. *Constraints An Int. J.*, 15(3):327–353.