University of Glasgow | School of Computing Science

Algorithm Selection for Maximum Common Subgraph

Paulius Dilkas

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — 4th March 2018

Abstract

Many algorithms have been proposed for solving the maximum common subgraph problem. Some algorithms tend to be better for some types of instances, but there is no overall winner. Machine learning could be employed to choose an algorithm for each problem instance based on various properties of the two graphs. We show the potential performance benefits of this approach over using any single algorithm, and achieve over 85% of that potential for graphs with just vertex labels and graphs with both vertex and edge labels. Furthermore, we present new data on the drastic changes in algorithms' performance, when changing the number of distinct labels, making the conclusions of many research papers seem questionable and overly simplistic. Moreover, we use our trained machine learning models to gain insight into how well the models can recognise instances favouring each algorithm, which features of the problem instances are the most important, and which algorithms prefer what values of those features. Finally, we explore several ways to improve existing algorithms and create new ones: we adapt the $k\downarrow$ algorithm to handle vertex labels, and propose two ways of switching algorithms mid-execution.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

Maximum common subgraph (MCS) is a way to determine the similarity between two graphs. More specifically, MCS algorithms find what the two graphs have in common (these ideas will be made more concrete in Section 1.1). Since graphs can represent many real-world phenomena, MCS algorithms have been used in molecular science [14, 18, 35, 47], malware detection [45], and substructure discovery [9, 13].

Different MCS algorithms tend to be best for different types of graphs, depending on how many vertices they have, and whether they contain labels [39]. In some cases the differences are quite small, and typically the overall best algorithm gets outperformed by other algorithms as often as half the time (see Chapter 3 for more details). Since there are no (known) clear boundaries that separate instances best handled by each algorith, perhaps training a machine learning (ML) model to quickly choose an algorithm based on features of the input graphs would result in a algorithm faster than any individual algorithm. This is an approach known as algorithm selection [48] and is the main focus of this project.

In the rest of this chapter we provide rigorous definitions for concepts such as algorithm selection, define the kind of graphs we are interested in, continue with some necessary graph theory definitions, building up to the definition of an MCS. We then descibe and illustrate the algorithms used in this project. In order to compare MCS algorithms and reliably choose an algorithm for each problem instance, we need to run all algorithms on some pairs of graphs, and use an ML algorithm to find how their running times (or, more precisely, which algorithm is expected to win) depend on various features of the graphs. Thus, Chapter 2 describes all graphs used to compare algorithms and train ML models, looking into their origins, how they were constructed, and undocumented but relevant features. Chapter 3 then describes how the algorithms were run to generate running time data, explores how the algorithms compare on different parts of the data, outlines the features used for ML, and describes their distributions in the data. We then move on to building the ML models in Chapter 4, where we start with the relevant definitions in ML and statistics, describing the relevant software and the ML algorithm used. Afterwards, we describe various ways to evaluate the resulting models and do so for three different models. Finally, Chapter 5 provides ideas on how to expand one of the algorithms to previously unsupported instances, and introduces a new algorithm that merges two algorithms into one.

1.1 Definitions

Definition 1.1. Given a set $\mathcal{I}$ of problem instances, a space of algorithms $\mathcal{A}$, and a performance measure $m\colon \mathcal{I} \times \mathcal{A} \to \mathbb{R}$, the *per-instance algorithm selection problem* is to find a mapping $s\colon \mathcal{I} \to \mathcal{A}$ that optimises $\mathbb{E}[m(i, s(i))]$ [3]. $\mathcal{A}$ (and sometimes this problem in general) is often referred to as an *algorithm portfolio* [31, 61].

Definition 1.2. Let S be a set and k a non-negative integer. Then $[S]^k$ denotes the set of *k-subsets* of S [58], i.e.,

$$[S]^k = \{A \subseteq S : |A| = k\}.$$

Definition 1.3. An *undirected multigraph* is a pair (V, E), where V is a set of vertices and E is a set of edges, together with a map $E \to V \cup [V]^2$, which assigns one or two vertices to each edge [12]. If an edge is assigned to a single vertex, it is called a *loop*. When several edges map to the same pair of vertices, they are referred to as *multiple edges*.

For the purposes of this project, we look at two kinds of labelled graphs: those with vertex labels, and those with both vertex and edge labels. Labels can be used to represent important properties of graphs such as types of atoms and bonds in chemistry. We define them as follows (the definitions are loosely inspired by [1]):

Definition 1.4. A *(vertex-)labelled graph* is a 3-tuple $G = (V, E, \mu)$, where V, E are as in Definition 1.3 and $\mu\colon V \to \{0, \ldots, N-1\}$ is a vertex labelling function, for some $N \in \mathbb{N}$.

Definition 1.5. A *fully labelled graph* is a 4-tuple $G = (V, E, \mu, \zeta)$, where the first three elements are as in Definition 1.4 and $\zeta\colon E \to \{0, \ldots, M-1\}$ is an edge labelling function, for some $M \in \mathbb{N}$.

Specifically, note that:

- If a graph is labelled, then all its vertices (and possibly edges) are assigned a label.
- We are only considering finite sets of labels, represented by non-negative integers.
- A vertex-labelled graph is just a special case of a fully labelled graph with $M = 1$.
- An unlabelled graph is just a special case of of a vertex-labelled graph with $N = 1$.

The last two observations allow us to tailor subsequent definitions to fully labelled graphs without having to redefine everything for unlabelled and vertex-labelled graphs as well. When vertex and/or edge labels are immaterial to a definition, we will omit the labelling function from the full 4-tuple and denote a graph as just $G = (V, E)$.

We formulate all definitions in terms of undirected multigraphs, as some of the graphs in Chapter 2 contain multiple edges, and many contain at least one loop. In the rest of the dissertation, we will use the word "graph" to denote undirected multigraphs. Next we adapt some basic graph theory definitions to work with our definitions of graphs and labelling. As most definitions in the literature are typically defined for simple graphs with no labels, the cited sources are heavily adapted to the full generality of graphs relevant to this dissertation.

Definition 1.6. Two graphs $G_1 = (V_1, E_1, \mu_1, \zeta_1)$ and $G_2 = (V_2, E_2, \mu_2, \zeta_2)$ are said to be *isomorphic* if there are bijections $f\colon V_1 \to V_2$, $g\colon E_1 \to E_2$ such that [47]:

- $\forall e \in E_1$, if e maps to some $v \in V_1 \cup [V_1]^2$, then $g(e)$ maps to $f(v)$, where $f(v) = \{f(v_1), f(v_2)\}$ if $v = \{v_1, v_2\}$ (preserving structure);
- $\forall v \in V_1$, $\mu_2(f(v)) = \mu_1(v)$ (preserving vertex labels) [2];
- $\forall e \in E_1$, $\zeta_2(g(e)) = \zeta_1(e)$ (preserving edge labels).

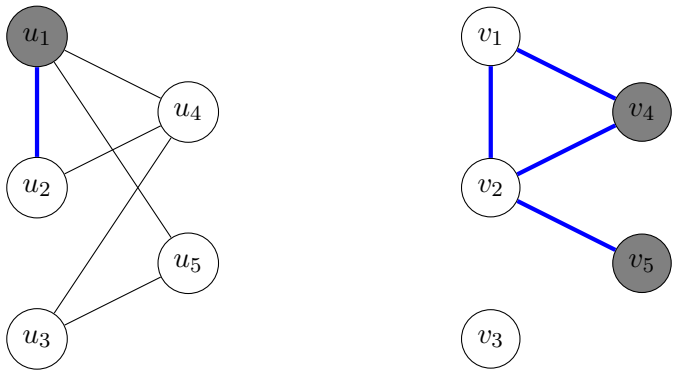


Figure 1.1: Two graphs with binary ($\{0, 1\}$) labels on both vertices and edges with pattern graph G_p on the left and target graph G_t on the right. Grey vertices and thick blue edges represent label 1, while white vertices and black edges have label 0.

Definition 1.7. Let $f\colon X \to Y$ be a function. Then the *restriction* of f to $A \subseteq X$ is the function $f|_A\colon A \to Y$ such that $\forall a \in A$, $f|_A(a) = f(a)$ [53].

Definition 1.8. A graph $G' = (V', E', \mu', \zeta')$ is a *subgraph* of graph $G = (V, E, \mu, \zeta)$ if: $V' \subseteq V$, $E' \subseteq E$ (with the edge-mapping function restricted to E'), $\mu' = \mu|_{V'}$, $\zeta' = \zeta|_{E'}$ [12].

Definition 1.9. An *induced subgraph* of a graph $G = (V, E)$ is a subgraph $H = (S, E')$, where $E' \subseteq E$ is a set of edges mapped to $S \cup [S]^2$ [47].

Definition 1.10. A *clique* C in a graph $G = (V, E)$ is a subset of V such that $\forall v_1, v_2 \in C$ with $v_1 \neq v_2$, there is an edge in E mapping to $\{v_1, v_2\}$ [44].

Finally, we define the main problem of this project:

Definition 1.11. A *maximum common (induced) subgraph* (MCS) between graphs G_1 and G_2 is a graph $G_3 = (V_3, E_3)$ such that G_3 is isomorphic to induced subgraphs of both G_1 and G_2 with $|V_3|$ maximised [47]. The *maximum common (induced) subgraph problem* is the problem of finding an MCS between two given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, usually expressed as a bijection between two subsets of vertices $U_1 \subseteq V_1$ and $U_2 \subseteq V_2$.

The decision version of this problem (finding a common induced subgraph of a fixed size) is NP-complete [21], and thus the MCS optimisation problem is NP-hard [39, 43].

Example 1.1. Consider the graphs in Figure 1.1. Since G_p has only one blue edge, while all edges of G_t are blue, any common subgraph can have up to one edge. It is then obvious that any MCS must be isomorphic to the subgraph induced by $\{u_1, u_2, u_3\}$ in G_p and to multiple induced subgraphs of G_t.

1.2 Algorithms

We consider four algorithms that were shown to be competitive in a recent paper by McCreesh, Prosser and Trimble [39]: $k\downarrow$ [20], McSplit [39], McSplit$\downarrow$ [39], and the clique encoding [36]. In order to explain how they work, we need to define two other problems:

Definition 1.12. Given a graph G, the *maximum clique problem* is an optimisation problem asking for a clique in G with maximum cardinality [44].

Definition 1.13. Given two (finite) graphs G_1 and G_2, the *subgraph isomorphism problem* is the decision problem of determining whether G_1 is isomorphic to a subgraph of G_2 [10]. G_1 and G_2 are usually referred to as the *pattern* and *target* graphs [51, 56, 62], respectively. We will use these definitions in the context of both the subgraph isomorphism problem and the MCS problem, where the word 'pattern' will typically refer to the smaller of the two graphs.

The $k\downarrow$ algorithm [20] starts by trying to solve the subgraph isomorphism problem, i.e., finding the pattern graph in the target graph. If that fails, it allows a single vertex of the pattern graph to not match any of the target graph vertices and tries again, allowing smaller and smaller pattern graphs until it finds a solution. The number of vertices of the pattern graph that are allowed this additional freedom is represented by k. More specifically, the algorithm creates a domain for each pattern graph vertex, which initially includes all vertices of the target graph and k wildcards. The domains are filtered with various propagation techniques. Then the search begins with a smallest domain (not counting wildcards), a value is chosen, and domains are filtered again to eliminate the chosen value.

Definition 1.14. Although typically the *image* of a set S under a function f [6] is defined as a set

$$f(S) = \{f(s) : s \in S\},$$

we define the *multiset image* of S under f by the same expression, but with $f(S)$ as a multiset, i.e., if $f(x) = f(y)$ for two different $x, y \in S$, then $f(x)$ is in $f(S)$ at least two times [23].

Definition 1.15. The *association graph* of an MCS problem instance between graphs $G_1 = (V_1, E_1, \mu_1, \zeta_1)$ and $G_2 = (V_2, E_2, \mu_2, \zeta_2)$, denoted $G_1 \nabla G_2 = (V_3, E_3)$ [36], is a simple graph that can be defined as follows:

- we add a vertex (v_1, v_2) to V_3 if $\mu_1(v_1) = \mu_2(v_2)$, and the multiset image of loops of v_1 under ζ_1 is equal to the multiset image of loops of v_2 under ζ_2;
- E_3 is defined so that there is an edge between vertices (u_1, u_2) and (v_1, v_2) if $u_1 \neq v_1$, $u_2 \neq v_2$, and the multiset image of edges in E_1 that are between u_1 and v_1 under ζ_1 is equal to the multiset image of edges in E_2 that are between u_2 and v_2 under ζ_2.

The clique encoding [36] solves the MCS problem by creating an association graph and transforming the problem into an instance of the maximum clique problem [30], which is then solved by a sequential version of the maximum clique solver by McCreesh and Prosser [37], which is a branch and bound algorithm that uses bitsets and greedy colouring. Colouring is used to provide a quick upper bound: if a subgraph can be coloured with k colours, then it cannot have a clique of size more than k.

MCSPLIT [39] is a branch and bound algorithm that builds its own bit string labels for vertices in both pattern and target graphs. Once it chooses to match a vertex u in graph G_1 with a vertex v in graph G_2, it iterates over all unmatched vertices in both graphs, adding a 1 to their labels if they are adjacent to u or v and 0 otherwise. That way a vertex can only be matched with vertices that have the same labels. The labels are also used in the upper bound heuristic function: if a particular label is assigned to m vertices in G_1 and n vertices in G_2, then up to $\min\{m, n\}$ pairs can be matched for that label.

MCSPLIT$\downarrow$ is a variant of MCSPLIT mentioned but not explained in the original paper [39]. It is meant to be similar to $k\downarrow$ in that it starts by trying to find a subgraph isomorphism and keeps decreasing the size of

common subgraphs that it is interested in until a solution is found. Based on the source code[1], there are a few key differences between McSplit↓ and McSplit:

- Instead of always looking for larger and larger common subgraphs, we have a goal size and exit early if a common subgraph of that size is found.
- The goal size is decreased if the search finishes without a solution.
- Having a big goal size allows the heuristic to be more selective and prune more of the search tree branches.

The McSplit paper [39] compared these (and a few constraint programming) algorithms and found McSplit to win with unlabelled graphs from Section 2.1 of this dissertation, the clique encoding to win with labelled graphs, and McSplit↓ to win with unlabelled graphs from Section 2.2 (closely followed by k↓).

[1] https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph/blob/master/code/james-cpp/mcsp.c

Chapter 2

Problem Instances

In order to build ML models that predict the winning algorithm for each pair of graphs as well as to see how the algorithms compare for different kinds of graphs, we need to have or generate some graphs. For that we use two graph databases that contain a large variety of graphs differing in size, various characteristics, and the way they were generated.

2.1 Labelled Graphs

All labelled graphs are taken from the ARG Database [15, 50], which is a large collection of graphs for benchmarking various graph-matching algorithms. The database contains randomly generated graphs; 2D, 3D, and 4D meshes; and bounded valence graphs. Furthermore, each graph-generating algorithm is executed with several (3–5) different parameter values. The database includes $81{,}400$ pairs of labelled graphs. Their unlabelled versions are used as well.

2.1.1 Characteristics of Graph Labelling

In Definitions 1.4 and 1.5 we used N and M to denote the number of different labels for vertices and edges, respectively. The ARG Database supports interpreting the same graphs to have different numbers of different labels via the following parameter[1]:

Definition 2.1. A graph $G = (V, E)$ is said to have a $p\%$ *(vertex) labelling* if

$$N = \max\left\{2^n : n \in \mathbb{N},\, 2^n < \left\lfloor \frac{p}{100\%} \times |V| \right\rfloor \right\}.$$

The default value for p is 33%. The publications associated with the database [15, 50] say nothing about how the labels are distributed among the N values. We calculate the number of vertices that were assigned each label for each graph (represented by C) and compare those values with the numbers we would expect from a uniform distribution (represented by $\mathrm{E}(C)$). We plot a histogram of the difference $\mathrm{E}(C) - C$ in Figure 2.1 and observe that the difference is normally distributed around 0.

[1] `http://mivia.unisa.it/datasets/graph-database/arg-database/documentation/`, "How to read labeled graphs"

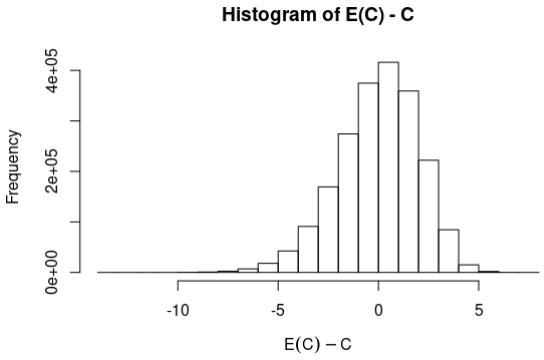


Figure 2.1: Histogram of the difference between the expected number of vertices assigned each label and the actual number (for all labelled graphs).

2.2 Unlabelled Graphs

We also include a collection of benchmark instances for the subgraph isomorphism problem[2] (with the biochemical reactions dataset excluded since we are not dealing with directed graphs). It contains only unlabelled graphs, and consists of the following sets[3]:

images-CVIU11 Graphs generated from segmented images. There are 43 pattern graphs and 146 target graphs, giving a total of 6278 instances.

meshes-CVIU11 Graphs generated from meshes modelling 3D objects. 6 pattern graphs and 503 target graphs, giving a total of 3018 instances. Both `images-CVIU11` and `meshes-CVIU11` datasets are described in a paper by Damiand *et al.* [11].

images-PR15 Graphs generated from segmented images [52]. There are 24 pattern graphs and a single target graph, giving 24 instances.

LV Graphs with various properties (connected, biconnected, triconnected, bipartite, planar, etc.). 49 graphs are paired up in all possible ways, giving $49^2 = 2401$ instances.

scalefree Scale-free networks generated using a power law distribution of degrees (100 instances).

si Bounded valence graphs, 4D meshes, and randomly generated graphs (1170 instances). This is the unlabelled part of the ARG database. `LV`, `scalefree`, and `si` datasets are described in two papers by Solnon [51] and by Zampelli, Deville, and Solnon [62].

phase Random graphs generated to be close to the satisfiable-unsatisfiable phase transition (200 instances) [38].

largerGraphs Larger instances of the `LV` dataset. There are 70 graphs, giving $70^2 = 4900$ instances. The separation was made and used in the $k\downarrow$ paper [20], a paper on algorithm selection for the subgraph isomorphism problem by Kotthoff, McCreesh, and Solnon [28], and the McSplit paper [39].

Remark 2.1. Since $k\downarrow$ comes from the subgraph isomorphism problem background, it treats the two (pattern and target) graphs differently. Therefore, when graphs are not divided into patterns and targets, we run the algorithms with both orderings ((G_1, G_2) and (G_2, G_1)).

[2] `http://liris.cnrs.fr/csolnon/SIP.html`

[3] This set of instances was taken from the repository (`https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph`) for the McSplit paper [39] and has some minor differences from the version on Christine Solnon's website.

Chapter 3

Generating Data

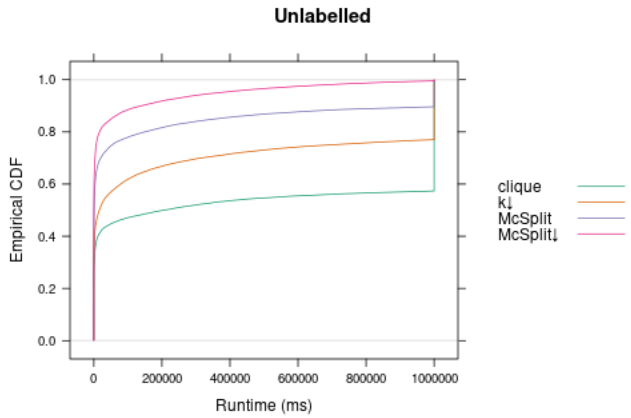
A machine learning (ML) model requires data to learn from. We are using an R package called LLAMA [25, 26], which helps to train and evaluate ML models in order to compare algorithms and was used to create algorithm portfolios for the travelling salesperson problem [27] and the subgraph isomorphism problem [28]. First, we run each algorithm on all pairs of pattern-target graphs and record the running times (described in Section 3.1). Then, we adapt a graph feature extractor program used in an algorithm selection paper for the subgraph isomorphism problem by Kotthoff, McCreesh, and Solnon [28] to handle the binary format of the ARG Database [15, 50], run it on all graphs, and record the features in a way described in Section 3.2.

3.1 Running Time of Algorithms

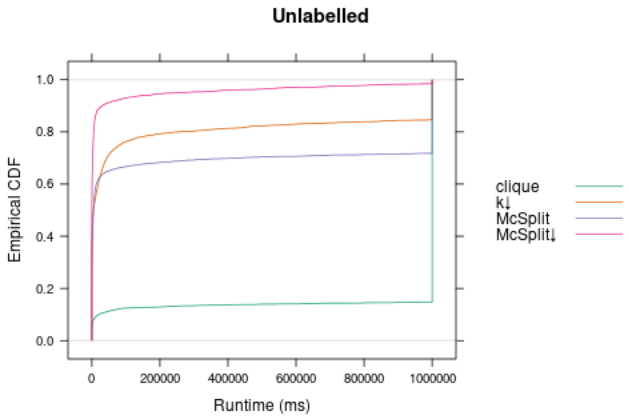
The algorithms were compiled with gcc 6.3.0 and run on nodes with Ubuntu 17.04 (Zesty Zapus) operating system, Intel Xeon E5-2697A v4 (2.60 GHz) processors, and 512 GB RAM. Each algorithm was set to time out after 1000 s. A Makefile was created to run multiple experiments in parallel (with, e.g., `make -j 64`), which generates pairs of graph filenames for all datasets, runs the selected algorithms with various command line arguments, redirects their output to files that are later parsed using `sed` and regular expressions into the CSV format. For each algorithm, we keep the full names of pattern and target graphs, the number of vertices in the returned MCS, running time as reported by the algorithms themselves, and the number of explored nodes in the search tree. Entries with running time greater than or equal to the timeout value are considered to have timed out. The aforementioned node counts are collected but not currently used. Afterwards, the answers of different algorithms are checked for equality (for algorithms that did not time out).

Some limitations had to be enforced to avoid running out of memory. First, the clique algorithm requires $O(n^2m^2)$ memory for a pair of graphs with n and m vertices [20, 36], so its virtual memory usage was limited to 7 GB with `ulimit -v` and the instances from Section 2.2 (which contain much larger graphs) were restricted to $m \times n < 16{,}000$. Second, having almost 10^5 problem instances and analysing 7 different kinds of labelling (according to Definition 2.1) results in too much data for the ML algorithm. Therefore, we sample $30{,}000$ out of $81{,}400$ instances from Section 2.1. The sample is drawn once and used to train ML models for both vertex-labelled and fully labelled graphs. As with 50% labelling most instances are solved within the time limit, sampling from the whole database still leaves us with enough instances where at least one algorithm finished within the time limit.

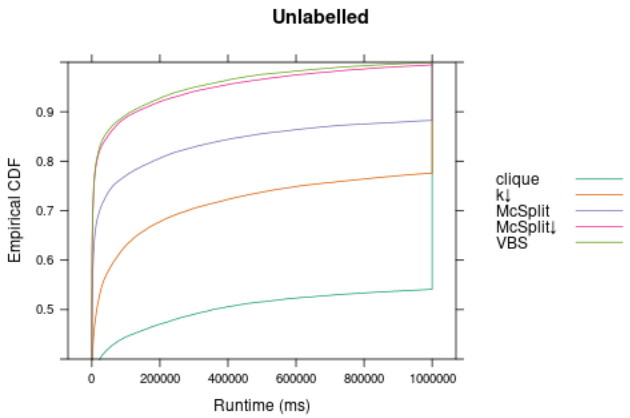
In the rest of this section we explore and compare how the algorithms performed on the three different subproblems under consideration, namely with unlabelled, vertex-labelled and fully labelled graphs. We introduce *empirical cumulative distribution function (ECDF)* plots [60] (also known as cumulative plots): for each unit of



(a) Data from Section 2.1, the ARG Database



(b) Data from Section 2.2



(c) All unlabelled data

Figure 3.1: Comparison of the runtimes of algorithms on unlabelled data.

time on the horizontal axis, the value on the vertical axis represents the proportion of problem instances solved in that amount of time or less on an instance-by-instance basis. For example, if an algorithm finished one instance in 1 s and two instances in 2 s, then its ECDF curve will connect points $(0, 0)$, $(1, \frac{1}{3})$, and $(2, 1)$ (assuming that time is measured in seconds).

3.1.1 Unlabelled Graphs

We plot the ECDF plots for unlabelled graphs in both databases in Figure 3.1. We can check that the orderings of the algorithms in parts (a) and (b) of Figure 3.1 are the same as in Figures 3a and 4 of the McSplit paper [39]. Namely, McSplit outperforms $k\downarrow$ in Figure 3.1a, and the opposite happens in Figure 3.1b. In Figure 3.1c we also plot a curve for the *virtual best solver (VBS)*, i.e., a perfect algorithm portfolio that always chooses the best-performing algorithm for each problem instance. Note that the difference between McSplit↓ and the VBS is very small. Therefore, a portfolio cannot provide significant performance benefits for unlabelled graphs.

Table 3.1 shows that most of the datasets have multiple algorithms that managed to outperform the others for some problem instances. Thus, looking at the differences between different datasets will not be enough to predict the best algorithm. More specifically, Table 3.1 shows the numbers of times that each algorithm's runtime was lower than or equal to the runtimes of other algorithms. Therefore, if 2 or more lowest runtimes are equal (as can often happen with single-digit runtimes), both algorithms are marked as winning in the table.

Given this information, we would expect the ML algorithm to suggest using McSplit and McSplit↓ most

Dataset	clique	$k\downarrow$	McSplit	McSplit$\downarrow$
images-CVIU11	0	32	79	1081
images-PR15	0	0	0	24
largerGraphs	0	14	30	167
LV	90	30	489	439
meshes-CVIU11	0	13	0	23
phase	0	0	0	0
scalefree	0	0	0	80
si	0	10	102	1135
ARG Database	1443	141	21,965	27,305
Total	1533	240	22,665	30,254

Table 3.1: The number of times each algorithm outperformed other algorithms, for each dataset.

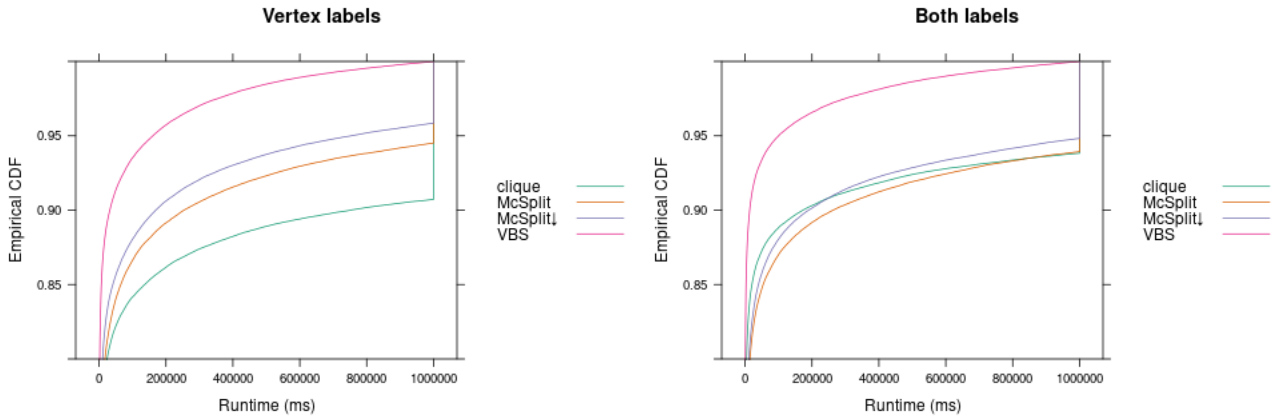


Figure 3.2: Cumulative plots with the vertical axis starting at 0.8.

of the time, occasionally consider the clique encoding, and mostly forget about $k\downarrow$.

3.1.2 Labelled Graphs

We deal with the two types of labelling separately; however, since the results are fairly similar, we describe them in the same section to highlight the differences. We plot the ECDFs in Figure 3.2. The situation with vertex-labelled graphs is quite straightforward: McSplit$\downarrow$ is slight better than McSplit, which is better than the clique encoding. Moreover, the VBS curve is significantly higher, providing plenty of room for an ML model to outperform individual algorithms (unlike with unlabelled data). This latter fact is true for fully labelled graphs as well, whereas the other three curves provide a more interesting story. The clique algorithm is briefly winning for shorter timeout values, then is between McSplit$\downarrow$ and McSplit until it drops to the 3$^{\text{rd}}$ place right before the final timeout at 1,000,000 ms. This is likely because the clique encoding is better with higher labelling percentages (we will see this shortly) and such graphs are generally easier to solve (because there are fewer "matchable" combinations of vertices, each pair of vertices is likely to have different labels).

Just like with unlabelled graphs we could split the data into different subsets based on how (and by whom) the graphs were generated, now we can analyse how the situation changes with different labelling percentages. While some publications explored several different values of the labelling percentage (33%, 50%, 75%) [5, 7, 8], and some explored only one (33%) [36, 39], we explore a much wider range from 5% to 50%, showing exactly when algorithms considered best for labelled graphs stop being good, when faced with a smaller number of

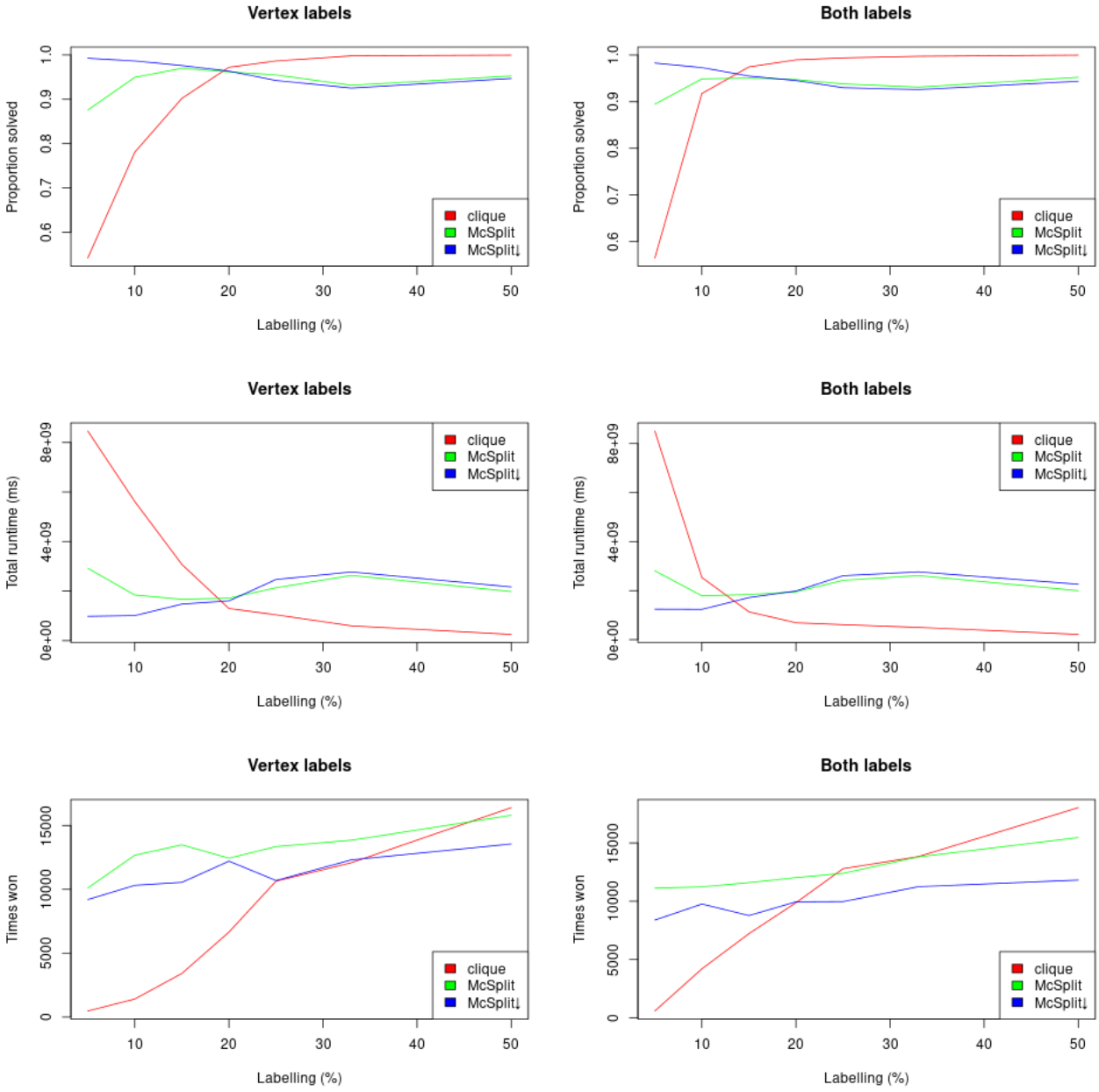


Figure 3.3: For both types of labelling and for the three algorithms we plot how three characteristics change with respect to the labelling percentage: proportion of instances solved within the time limit, total runtime, and the number of times each algorithm outperformed the others.

labels. In Figure 3.3 we analyse two performance measures (proportion of instances solved and total runtime) as well as the number of times each algorithm "wins", for each labelling percentage. The two performance statistics tell exactly the same story, and it is the same for both types of labelling. The only difference is that the clique algorithm's major drop in performance starts at 20% for vertex-labelled graphs and at 15% for fully labelled graphs. With higher labelling percentages, the clique encoding is better than McSplit, which is marginally better than McSplit↓. With lower levels of labelling, two differences emerge: McSplit experiences a drop in performance below McSplit↓, and the performance of the clique algorithm starts to drop exponentially. The latter is exactly what makes the clique encoding finish last in Figure 3.2, even though it is in the lead for most labelling percentages.

Remark 3.1. Note that we are only considering instances solved by at least one algorithm. Out of the 30,000 instances selected by our random sampling procedure, the number of such instances ranges from 56% with 5% labelling to 99% with 50% labelling for vertex-labelled graphs (the percentages for fully labelled graphs are similar). Thus a (roughly) horizontal line should not be interpreted as the algorithm performing equally well for different labelling percentages: the performance of all algorithms improves with higher labelling percentages as the problem becomes significantly easier. In this case we are more interested in the differences between individual algorithms.

As for the last two plots of Figure 3.3, McSplit wins more often than McSplit↓. This is not surprising for higher labelling percentages (based on the other two types of plots), but this remains true for lower percentages as well. Perhaps this is because by definition McSplit↓ is better at handling problem instances with a big answer. With a smaller labelling percentage, McSplit is more likely to time out on those instances, contributing to McSplit↓ solving more instances than McSplit. Perhaps McSplit↓ is also much faster on those instances, ensuring its lower total runtime, while consistently falling slightly behind McSplit on easier instances, resulting in a lower win count. The one significant difference between the two types of labelling is that the clique encoding wins slightly less than McSplit↓ with vertex labels, but slightly more than McSplit with both vertex and edge labels, for 25%–33% labelling. Finally, the main observation from this plot is that the overall highest point is only at 16,401 (18,031) for vertex-labelled graphs (both vertex and edge labels), just slightly above half of the number of instances (30,000), and other than the clique encoding falling behind with $\leq 15\%$ labelling, win rates of the three algorithms stay similar. This makes the problem especially potent for an algorithm selection approach.

3.2 Graph Features

The initial set of features is based on the algorithm selection paper for the subgraph isomorphism problem [28], and consists of the following:

1. number of vertices,
2. number of edges,
3. mean degree,
4. maximum degree,
5. density,
6. mean distance between pairs of vertices,
7. maximum distance between pairs of vertices,
8. standard deviation of degrees,

9. number of loops,
10. proportion of all vertex pairs with a distance of at least 2, 3, and 4,
11. whether a graph is connected.

Definition 3.1. For a graph G with n vertices and m edges, the *(edge) density* is defined to be the proportion of potential edges that G actually has [12]. The standard formula used for *simple* graphs (i.e., graphs with no multiple edges or loops [42]) is

$$\frac{m}{\binom{n}{2}} = \frac{2m}{n(n-1)}.$$

Even though some of our graphs do contain multiple edges and loops, we stick to this formula, as it was used in the algorithm selection paper for the subgraph isomorphism problem [28], and it does not break the ML algorithm in any way to have the theoretical possibility of density greater than 1.

We exclude feature extraction running time as a viable feature by itself (used in the algorithm selection paper for the subgraph isomorphism problem [28]), since it would not provide any insight into what properties of the graph affect which algorithm is likely to achieve the best performance. Since $k\downarrow$ and McSplit$\downarrow$ both start by looking for (complete) subgraph isomorphisms, they are likely to outperform other algorithms when both graphs are very similar and the maximum common subgraph has (almost) as many vertices as the smaller of the two graphs. Thus, for each feature f in features 1–7 (excluding the rest to avoid division by 0), we also add a feature for the ratio $\frac{f(G_p)}{f(G_t)}$, where G_p and G_t are the pattern and target graphs, respectively.

We analyse three different types of labelling and treat them as separate problems: no labels, vertex labels, vertex and edge labels. For the last two types, we add a feature corresponding to p defined in Definition 2.1 and collect data for the following values of p: 5%, 10%, 15%, 20%, 25%, 33%, 50%[1]. The values correspond to having about 20, 10, 5, 4, 3, and 2 vertices/edges with the same label on average, respectively.

3.2.1 Distributions of Features

In this section we discuss how the selected features are distributed in both databases. In order to save space, all plots are in Appendix A. As the graphs from Section 2.2 contain some very hard instances, we only consider graphs that are part of a pair of graphs solved by at least one algorithm. In order to visualise highly skewed data, we sometimes use density plots. Furthermore, we take $\log$ transformations of ratio features.

Firstly, one feature is not plotted as by definition it has only two possible values. 99.81% of graphs from Section 2.1 are connected, compared to 93.19% of graphs from Section 2.2. As both numbers are quite high, they may not be ideal for establishing if connectedness is a significant factor in determining which algorithm performs the best, but might be representative of real data in application domains such as chemistry [14]. Similarly, the number of loops for graphs from Section 2.1 is not plotted as it varies between two values: 0.98% of the graphs have a single loops, while the remaining majority of graphs have no loops. On the other hand, as shown in Figure A.1, some (although not many) graphs from Section 2.2 have significantly more loops.

Most of the features of graphs from Section 2.1 are displayed in Figure A.2. Other than the number of vertices, which is manually controlled by the creators of the database, all other distributions are centred around lower values, with some outliers on the high end. More importantly, we have some graphs that are quite dense and some graphs with higher mean distance values.

[1]When working with both vertex and edge labels, we only consider using the same value of p for both vertices and edges. This "convention" seems to have originated in a paper by the creators of the ARG Database [7] and was replicated in subsequent papers on MCS algorithms [36, 41].

The same plots for graphs from Section 2.2 in Figure A.3 show a similar story, albeit for clearer reasons. Since we filter out graphs that none of the algorithms were able to handle, our sample consists of all the easy instances and some harder instances that were solved by one or two algorithms. Harder instances typically have more vertices, which means they are also capable of higher values for many other features, hence all of the density plots are right-skewed.

Figure A.4 shows density plots of proportions of pairs of vertices with distance at least k for $k = 2, 3, 4$. For both databases, as k increases, the distributions shift to the left as expected. However, there is one important difference: even with $k = 4$ the plot for graphs from Section 2.2 has its highest peak around 0.9, which means that adding features for $k \geq 5$ could be valuable.

Finally, the density plots of $\log$-transformed ratio features are in Figures A.5, A.6, and A.7. Almost all of these plots for graphs from Section 2.2 (with the exception of the ratio of mean degree) are clearly bimodal with one of the two modes centred around 0. Hence we can infer the existence of two subpopulations: one where pattern and target graphs have very similar properties (the majority for most features) and one where pattern and target graphs are very different. As for the graphs from Section 2.1, the plot of the ratio of the number of vertices in Figure A.5 is perfectly symmetrical and centred around 0, since the number of vertices is a controlled variable for this database. All of the remaining plots for ratio features have most of the data very close to 0. Thus, the differences between pattern and target graphs are very small. Furthermore, all distributions are symmetrical—the pattern graph is just as likely to be larger/denser/etc. as the opposite.

3.2.2 Correlations

Another important question is how the features are correlated with each other. Even though research suggests that Pearson's correlation coefficient can be quite robust to violations of the data normality assumption [29] (which is clearly violated with our data), we choose Spearman's correlation coefficient as the more appropriate alternative:

Definition 3.2. *Spearman's rank correlation coefficient* [40] between vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ is defined as

$$S = \frac{\sum_i [r(x_i) - \overline{r}(\boldsymbol{x})][r(y_i) - \overline{r}(\boldsymbol{y})]}{\sqrt{\sum_i [r(x_i) - \overline{r}(\boldsymbol{x})]^2 \sum_i [r(y_i) - \overline{r}(\boldsymbol{y})]^2}}$$

, where $\overline{r}(\boldsymbol{x}) = \sum_i r(x_i)/n$, $\overline{r}(\boldsymbol{y}) = \sum_i r(y_i)/n$, and r is a rank function, which, for each of the two vectors, assigns a natural number to each element such that $x_i > x_j \implies r(x_i) > r(x_j)$, for any two elements of $\boldsymbol{x}$.

The resulting correlation coefficients are pictured in Figure 3.4. Ratio features, connectedness, and numbers of loops have almost no significant correlations, unlike the rest of the features that have significant positive and negative correlations. There is also a noticeable symmetry between features of the pattern and target graphs, i.e., if feature f of the pattern graph correlates with features g and h of the pattern graph, then it also correlates with features f, g, h of the target graph.

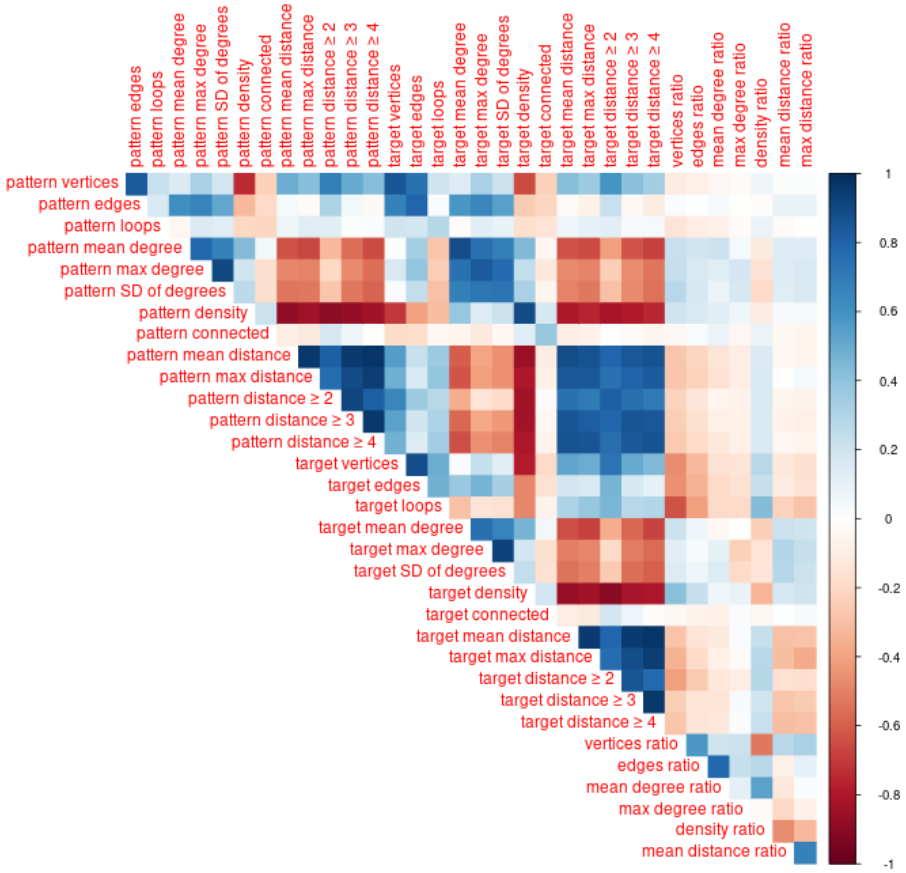


Figure 3.4: Spearman's rank correlation coefficients between all pairs of features, for both databases of graphs. Each correlation coefficient is represented by colour, with dark red for strong negative correlations, dark blue for strong positive correlations, and white for a correlation coefficient that is close to zero.

Chapter 4

Machine Learning Models & Their Evaluation

After running the algorithms with different types and percentages of labelling and recording their running times, an ML model can be trained to predict which algorithm should be chosen for each pair of graphs. For each such pair, LLAMA [26] (the package that facilitates constructing algorithm portfolios) can take:

- A list of features. With separate features for pattern and target graphs as well as ratio features and the labelling percentage, we have 34 features in total.
- A list of performance measures for all algorithms, i.e., the values that we are trying to optimise. In this case (as in most cases), this corresponds to running time. The values are capped at the timeout value (1,000,000 ms). Furthermore, instances that were not run on the clique algorithm are also set to the timeout value. Finally, we filter out instances where all algorithms timed out.
- A list of Boolean values, denoting whether each algorithm successfully finished or not. Timeouts, the clique algorithm running out of memory, and instances that were not run with the clique algorithm because of their size are all marked as false.
- A data frame, measuring the running time taken to compute each feature for each problem instance, a single number for the approximate time taken to compute all features for any instance, or a list with one or more costs per problem instance. We use the last option with a single cost per instance. This parameter is used to ensure a fair comparison when comparing the portfolio against other algorithms: the runtime of the portfolio is defined as the runtime of its chosen algorithm together with feature extraction time. While costs for graphs from Section 2.2 reached up to 65 s, the maximum cost for graphs from Section 2.1 is only 6 ms, with distance-based features being more expensive to compute.

After constructing the required data frames as described above, the data needs to be split into training and test sets. We use a technique called 10-fold *cross-validation*, which splits the data into 10 parts [57]. 9/10$^{\text{ths}}$ of the data is used to train the ML algorithm, while the remaining 1/10$^{\text{th}}$ is used to evaluate how good the trained model is. This process of training and evaluation is repeated 10 times, letting each of the 10 parts be used for evaluation exactly once. The goodness-of-fit criteria are then averaged out between the 10 runs.

The 10 folds could, of course, be chosen completely randomly. However, research suggests that stratified cross-validation typically outperforms random-sampling-based cross-validation and results in a better model [24]. Suppose we have a dataset of N elements. *Stratified sampling* partitions it into a number of subpopulations $s_1, \ldots, s_k$ with $n_1, \ldots, n_k$ elements, respectively (typically based on the value of some feature or collection of

features). It then draws from each subpopulation independently, ensuring that approximately n_i/N of the sample comes from subpopulation s_i for $i = 1, \ldots, k$ [34]. In this case the data is partitioned into four groups based on which algorithm performed best.

The cross-validation folds are then used to generate predictions on all data, where each prediction is made by a model that did not have that observation in its training data set. The predictions are then used to compare the ML model with individual algorithms and the VBS using ECDF plots and numeric statistics. LLAMA [25, 26] supports algorithm portfolios based on three different types of ML algorithms:

Classification The ML algorithm predicts which algorithm is likely to perform best on each problem instance.

Regression Each algorithm's data is used to train a separate ML model, predicting the algorithm's performance. The winning algorithm can then be chosen based on those predictions.

Clustering All instances of the training data are clustered and the best algorithm is determined for each cluster. New problem instances can then be assigned to the nearest cluster.

We are using a classification algorithm called random forests [4], implemented in R [32]. We chose this algorithm as it is recommended in the LLAMA manual [25] and successfully used in a similar study [28]. We use the default number of trees (500), and our trees have about 6000, $20{,}000$, and $21{,}000$ vertices on average for unlabelled, vertex-labelled, and fully labelled graphs, respectively.

In order to discuss and analyse the ML algorithm in more detail, we introduce some new terminology. Each problem instance with features and running times in the training dataset is called an *observation*. The *class* of an observation is the algorithm with lowest running time for that problem instance. Previously discussed features of graphs are sometimes referred to as *(independent) variables*. The *(problem) instance space* is the Cartesian product of the domains of features [49], where a *domain* of X (denoted $\operatorname{dom} X$) is a set of all possible values that X can take.

A *(classification) decision tree* is "a classifier expressed as a recursive partition of the instance space" [49]. Typically, it can be represented as a rooted binary tree, where each internal node *splits* the instance space into two regions based on the value of one of the features[1]. For example, for some feature X and a particular value $x \in \operatorname{dom} X$, the left child might be assigned all observations with $X < x$, while the right child would get observations with $X \geq x$. For each leaf node, we can count how many of its observations belong to each class and assign the most common class to that node.

When a decision tree is used to make a classification prediction, a data point travels from node to node (starting at the root node) according to the splitting rules. When it reaches a leaf node, the class assigned to that node is outputted as the tree's prediction.

A *random forest* builds a collection of decision trees [22]. Given p variables, each time a split is considered, the variable to split on is chosen from $\sqrt{p}$ rather than all p variables. Therefore, the strongest predictors are sometimes not even considered, ensuring a level of diversity among trees. An individual tree's prediction is called a *vote*. A classifying random forest predicts by collecting the votes from all its trees and predicting the class with the highest number of votes.

As we saw in Section 3.2, some of the feature data is highly skewed. Thus we should formally address the question of using transformations (such as $\log$ used for some of the plots or $x \mapsto 1/x$, giving the labelling percentage a different interpretation) before feeding the data into the ML algorithm. While it is a commonly held belief that predictor transformations are unnecessary for decision tree-based learning algorithms [16, 19, 55],

[1] Other possibilities include a node having more than two children and a split being made in a more complicated way. Although trees with such properties fit the definition of a decision tree, standard machine learning algorithms tend to be more restrictive [22, 49].

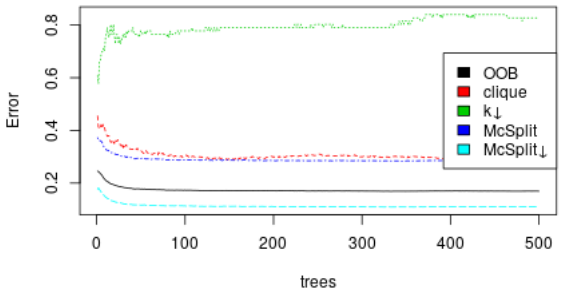


Figure 4.1: Convergence plot of various error measures as the number of trees in a random forest increases. The plot shows the OOB error and $1 -$ recall for each algorithm.

they can affect the predictions of previously unseen data [17]. To see this, consider a split between some values $x_1, x_2 > 0$ at $b = \frac{x_1+x_2}{2}$ (which is the type of split used by the `randomForest` package we are using [17]) and consider the same situation after applying the transformation $x \mapsto x^2$. The new boundary is $b' = \frac{x_1^2+x_2^2}{2}$ and it is easy to find $x > b$ such that $x^2 < b'$. Then, if this is the final split and x_1 and x_2 have different predictions, x will have different predictions with and without the transformation. In this project we do not use transformations as having over 100,000 data points and 500 trees reduces this effect and good algorithm portfolios are created without even mentioning transformations [27, 28]. However, we would encourage future researchers to consider log-transforming highly skewed features, especially if using an algorithm more sensitive to the distribution of data.

Lastly, we use the `parallelMap` package to train the model using multiple threads and the R code was heavily optimised to remove temporary variables as soon as they are no longer needed in order to reduce memory consumption.

4.1 Unlabelled Graphs

In this section we introduce various metrics and plots that can be used to examine a random forest, and use them to evaluate our model for unlabelled graphs. We examine how well the model predicts the winning of each algorithm, how important each feature is in making a prediction, how convinced the model is of each prediction, which algorithms prefer what values of various features, and how well the model performs compared to individual algorithms and the VBS.

4.1.1 Error Rates

Random forests support a convenient way to estimate the test error without cross-validation or any other kind of data splitting. Each tree in a random forest uses around 2/3 of the data [22]. The remaining 1/3 is referred to as *out-of-bag (OOB)* observations . For each observation in the data, we can predict the answer (the vote on which algorithm is expected to win) using all trees that have the observation as OOB. The majority vote is then declared to be the answer. The *OOB error* is the relative frequency of incorrect predictions [22]. As each prediction was made using trees that had not seen that particular observation before, OOB error is a valid estimate of test error. The black line in Figure 4.1 shows how the error converges to about 17% as we number of trees reaches 500.

The other lines in the figure, one for each algorithm, are defined as $1 -$ recall, where, for an algorithm A,

recall [46] is

$$\frac{\text{the number of instances that were correctly predicted as } A}{\text{the number of instances where } A \text{ is the correct prediction}}.$$

The error rates for MCSPLIT↓, MCSPLIT, the clique encoding, and $k\downarrow$ converge to 11%, 29%, 30%, and 80%, respectively. Unlike the errors of other classes, the error of $k\downarrow$ has an upward trend and converges to a very high value. Perhaps the model eventually learns not to predict $k\downarrow$ and the data points with $k\downarrow$ winning are treated as randomness in the data more so than a statistically significant trend.

4.1.2 Variable Importance

Next we are going to explore how important each feature is in making predictions, but for that we need to introduce some new definitions. Consider a single tree T in a random forest. The root of T can be reached by any observation, regardless of the values of its features. After passing some node n, some feature is restricted, i.e., it is imposed an upper or lower limit on the kind of values it can have for it to move towards a particular child of n [22]. We will refer to a part of feature space that an observation can be in while at some node n as a *region*.

Definition 4.1. Suppose we have K classes. Consider some region m. The *Gini index* is then defined as

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}),$$

where $\hat{p}_{mk}$ represents the proportion of observations in region m that are from class k (i.e., have algorithm k as the best algorithm) [22].

As we move down a tree, we want the region to be restricted to a single class. Then the observations from the training data satisfying the conditions imposed by the parent nodes would be classified with perfect accuracy. The Gini index is at its lowest when all proportions $\hat{p}_{mk}$ are close to either 0 or 1[2], meaning that almost all observations in the region belong to a single class. Hence the Gini index is often used to evaluate the quality of a split.

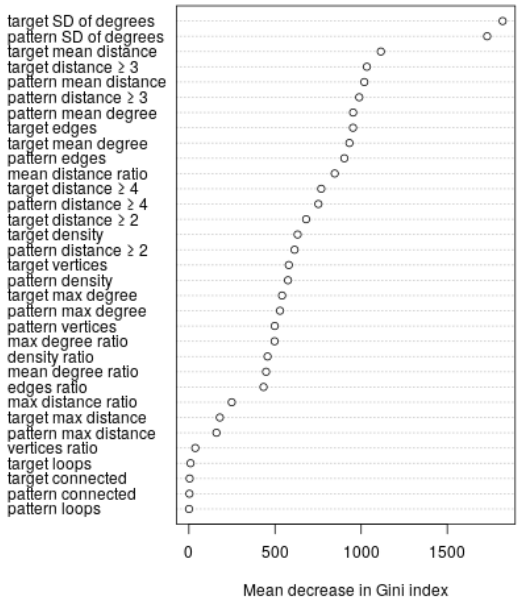
The variable importance measure of feature f in Figure 4.2a is calculated as the amount by which the Gini index decreases after passing nodes that use feature f, averaged over all trees in the random forest [22]. Looking at the figure more closely, the standard deviations of degrees of both target and pattern graphs are by far the most important predictors. Unsurprisingly, the worst predictors are the features with very low variance: number of loops and connectedness of both graphs. Perhaps more surprisingly, the ratio features are not as successful as one might have hoped: the ratio of the numbers of vertices is at the bottom 5$^{\text{th}}$ place and the best ratio feature, the mean distance ratio, is only 10$^{\text{th}}$. The last thing to note is that features of the pattern graph are always behind the same features of the target graph and usually not far behind. Perhaps this is due to some datasets having fewer pattern graphs, or pattern graphs having fewer vertices. The variable usage plot in Figure 4.2b tells a similar story: the orders are not identical, but there are no big outliers.

4.1.3 Margins

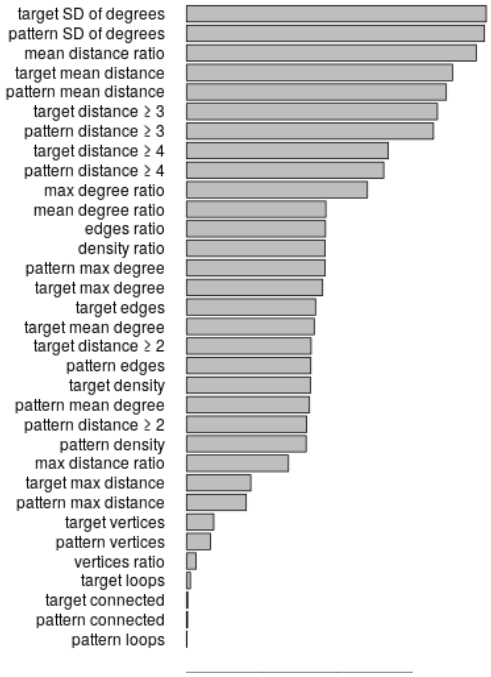
Definition 4.2. Let $c_1, \ldots, c_n$ be n classes and let p be a data point that belongs to class c_p. Let $v_1, \ldots, v_n$ denote the number of votes for each class when given p as input. The *margin* of p is

$$\frac{v_p}{\sum_{i=1}^{n} v_i} - \max_{i \neq p} \frac{v_i}{\sum_{j=1}^{n} v_j},$$

[2]Note that $G = 0$ when any single $\hat{p}_{mk} = 0$, regardless of the values of other proportions. Therefore, $G = 0$ does not automatically make a tree into a good classifier.



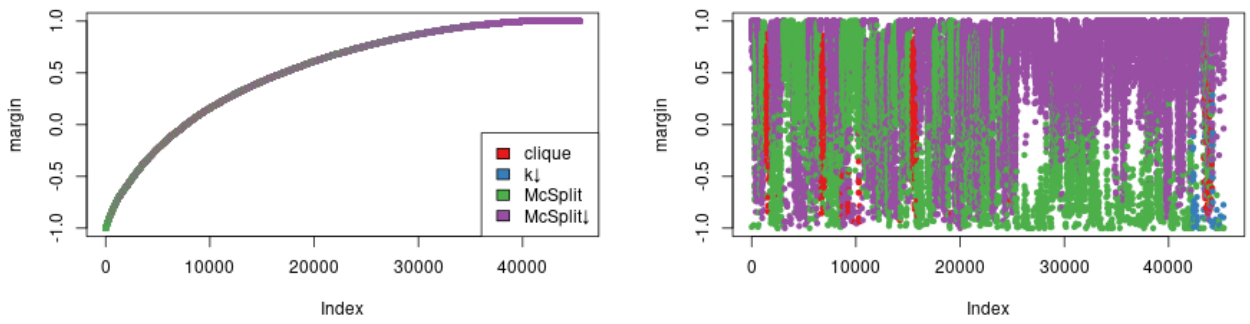
(a) Dot chart of variable importance calculated based on the Gini index and sorted from most important to least important.



(b) How often was each variable used to make splitting decisions?

which is a number in $[-1, 1]$ [33].

A value above 0 means that the forest as a whole predicted correctly. A margin of 1 would mean that all trees voted correctly. In Figure 4.3 we plot the margins in the following way: for each problem instance, the colour signifies the winning algorithm and the height shows its margin. Figure 4.3a shows the points sorted by the margin, while in Figure 4.3b they are left in the original order of the data (which mostly corresponds to the order of files in the databases, but with some variation since experiments are started in order but end at different times). We can recognise the same error rates as in Figure 4.1 as well as areas where McSplit and McSplit↓ dominate. We also plot the histograms of how the margins are distributed for each algorithm in Figure 4.4. We note that:



(a) Sorted

(b) Unsorted

Figure 4.3: Margins of the data points.

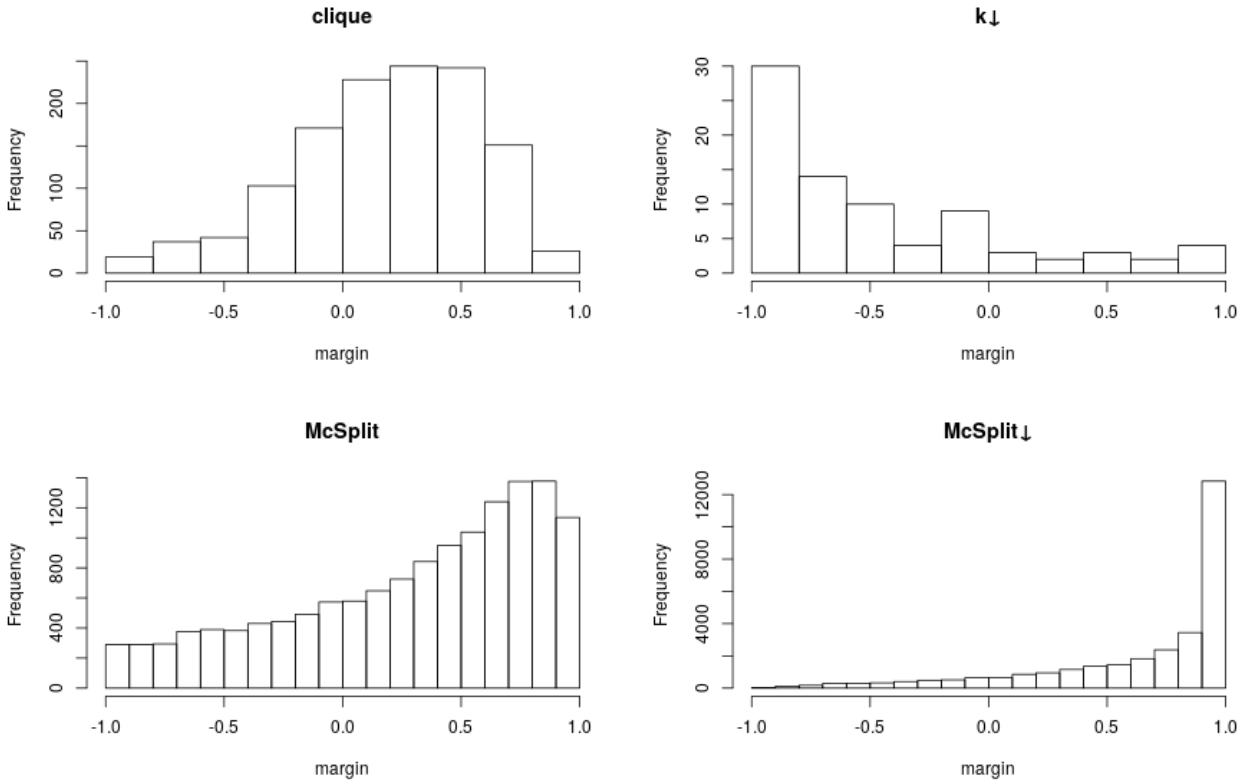


Figure 4.4: Histograms of margins for each winning algorithm.

- Instances best handled with the clique encoding are usually recognised, but with significant uncertainty.
- We are usually wrong about $k\downarrow$ (probably because it is a winning algorithm in only 0.44% of all cases).
- When faced with an instance that is best handled with MCSPLIT↓, the vast majority of trees vote correctly.
- MCSPLIT detection rates are decent, but far behind those of MCSPLIT↓.

4.1.4 Partial Dependence

Since the standard deviations of degrees in both target and pattern graphs are the most important features, we plot partial dependence plots of the standard deviation of degrees in the target graph[3] for MCSPLIT↓ and the clique encoding[4] in Figure 4.5. The plotted function [33] is defined as

$$f(x) = \log p_k(x) - \frac{1}{K}\sum_{i=1}^{K} \log p_i(x),$$

where:

- x is the value on the horizontal axis (in this case standard deviation of degrees in the target graph),
- $p_i(x)$ is the proportion of votes for class i for a problem instance with a standard deviation of degrees in the target graph equal to x,

[3]The plots for the standard deviation of degrees of the pattern graph are omitted since they are identical to those of the target graph.

[4]We omit the plots for the other two algorithms as the plot for MCSPLIT looks the same as the one for MCSPLIT↓ and prediction success rate for $k\downarrow$ is so low that a plot for $k\downarrow$ would be meaningless.

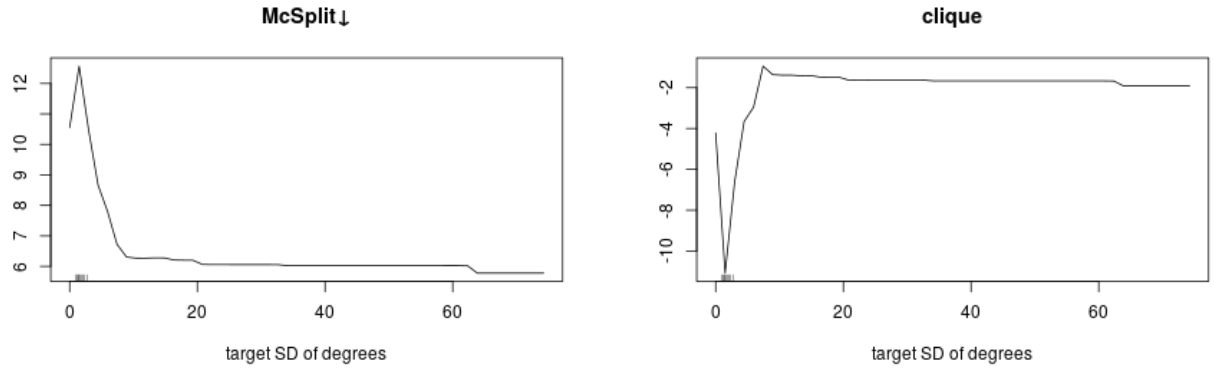


Figure 4.5: Partial dependence plots of the standard deviation of degrees in the target graph.

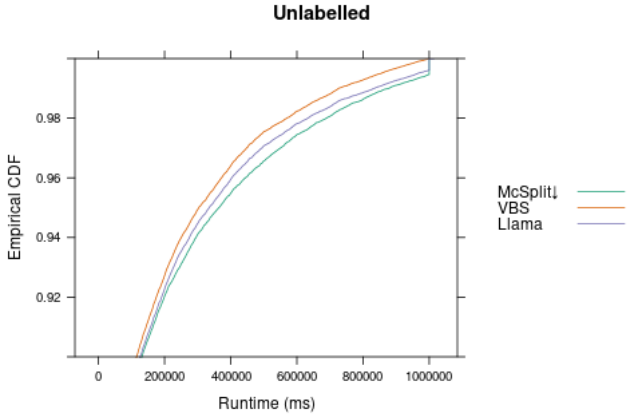


Figure 4.6: LLAMA model compared to the VBS and MCSPLIT↓.

- K is the number of classes,
- and k is the main class under consideration (MCSPLIT↓ and the clique encoding).

Essentially, $f(x)$ compares the proportion of votes for class k with the average value over all classes. We can deduce that a low standard deviation of degrees is a strong sign that MCSPLIT and MCSPLIT↓ should perform well. On the other hand, the clique encoding is expected to perform better on graphs with high variance in degrees. However, $\max f(x)$ for the clique encoding is just barely above 0 and much lower than $\min f(x)$ for MCSPLIT↓, meaning that the standard deviation of degrees does not provide enough information to choose the clique encoding over MCSPLIT or MCSPLIT↓.

4.1.5 Runtime Comparison

In order to compare our ML model with other algorithms, we treat the VBS as the upper bound and the single best solver MCSPLIT↓ as the lower bound. Out of $45{,}468$ instances solved by at least one algorithm, our model managed to solve $45{,}290$, compared to $45{,}223$ solved by MCSPLIT↓. In other words, it was able to close 27.3% of the gap between our lower and upper bounds in terms of instances solved within the time limit. Figure 4.6 shows how the ML model compares to the VBS and the single best solver MCSPLIT↓ (note that the vertical axis starts at 0.9 rather than 0). Unsurprisingly, the model outperforms MCSPLIT↓, but does not reach the best possible performance represented by the VBS.

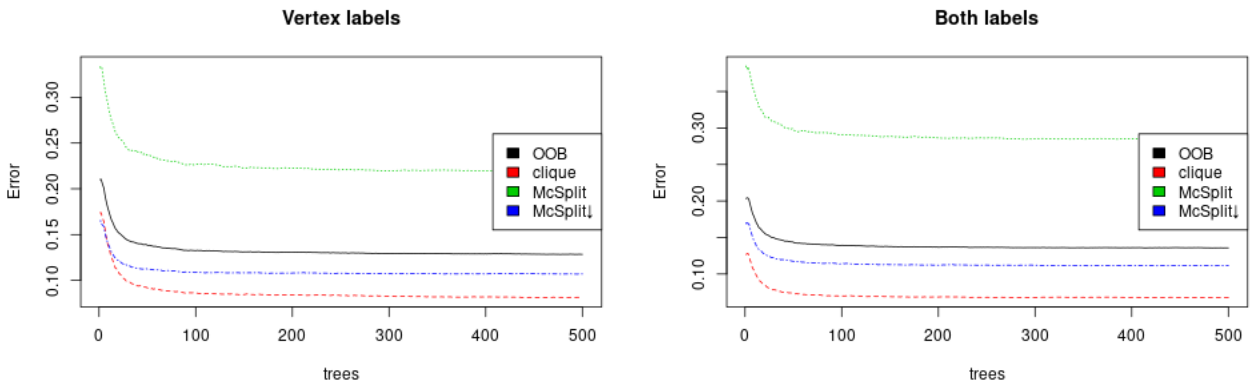


Figure 4.7: Convergence plots of various error measures as the number of trees in a random forest increases. The plots show the OOB error and $1 -$ recall for each algorithm.

Error type	Final value for vertex labels (%)	Final value for both labels (%)
OOB	13	14
clique	8	7
McSplit	22	29
McSplit↓	11	11

Table 4.1: The values that all 4 errors (approximately) converge to, for both types of labelling.

4.2 Labelled Graphs

Just like in the previous section, we plot the error rates (defined in Section 4.1.1) in Figure 4.7. The final error values are summarized in Table 4.1. This time all errors converge downwards and are below 50%. Comparing vertex-labelled and fully labelled subproblems, the only noticeable difference is that McSplit has a lower error rate for vertex-labelled graphs, which is also the worst-recognised algorithm for both types of labelling.

According to the variable importance measures plotted in Figure 4.8, the standard deviations of degrees for both pattern and target graphs still act as top predictors, however, they are overshadowed by the labelling feature, which is to be expected considering how impactful it is to the performance of the clique algorithm and the difficulty of the problem in general. For graphs with both vertex and edge labels, the vertex/edge counts make up the next most important predictors, while for vertex-labelled graphs, the number of vertices in the target graph seems to be less important (perhaps simply due to chance). Comparing this with the variable usage statistic in Figure 4.9 (which is almost identical for the two subproblems), the top 3 spots remain the same, the numbers of edges drop to the middle of the list, and the numbers of vertices drop to the 6^{th}–7^{th} places from the bottom. Apparently, even though all four of these predictors end up doing a great job at splitting the data to reduce the Gini index (a variation of which is used by the random forest algorithm in choosing which predictor to split on [59]), they are rarely used.

We show the margin plots and histograms in Figures 4.10 and 4.11. Again, the data for both types of labelling is close to identical. Note that there are plenty of data points in all three colours and the ML models are usually very convinced when predicting McSplit↓ and the clique encoding and are less sure when dealing with problem instances best handled with McSplit.

This time we show the partial dependence plots for the top two most important predictors (the labelling percentage and the standard deviation of degrees in the target graph[5]) in Figures 4.12 and 4.13, respectively. In

[5]The second most important predictor for the fully labelled case is the standard deviation of degrees in the pattern graph, but the

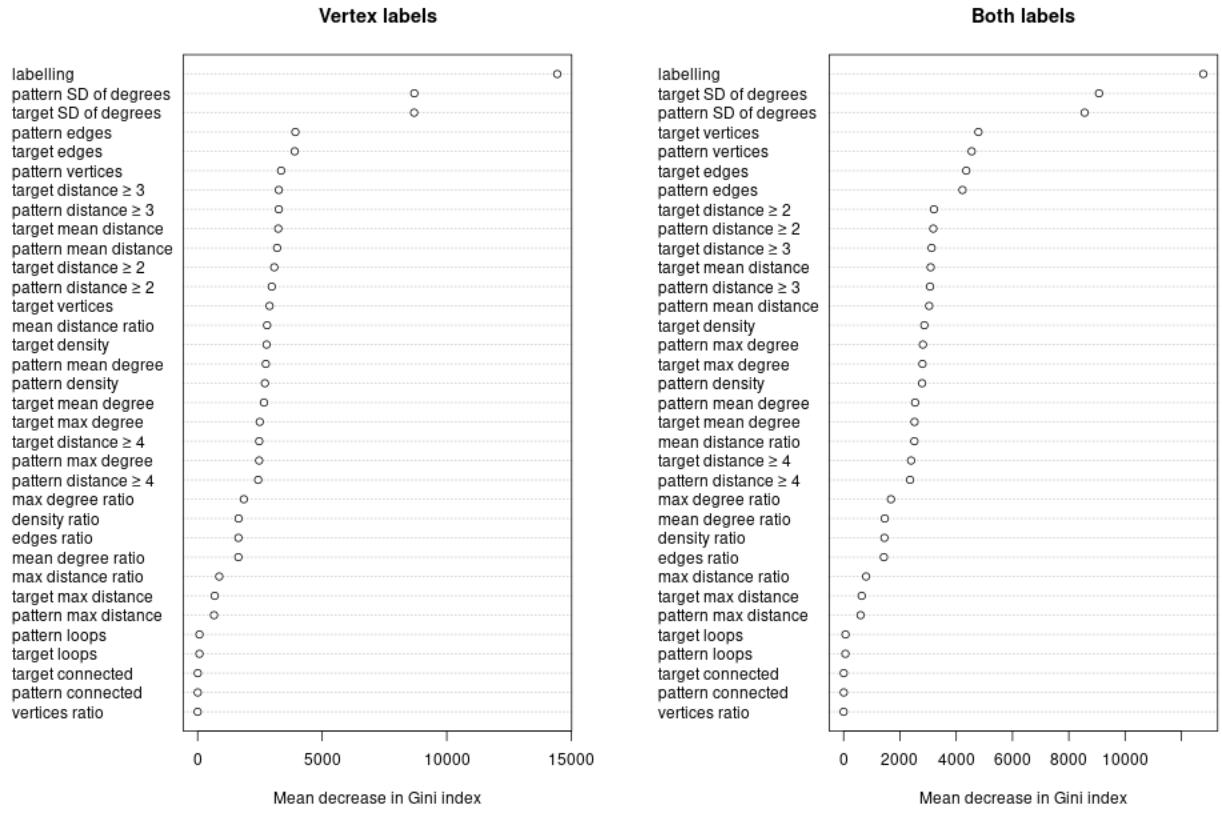


Figure 4.8: Variable importance for both types of labelling, sorted from most to least important.

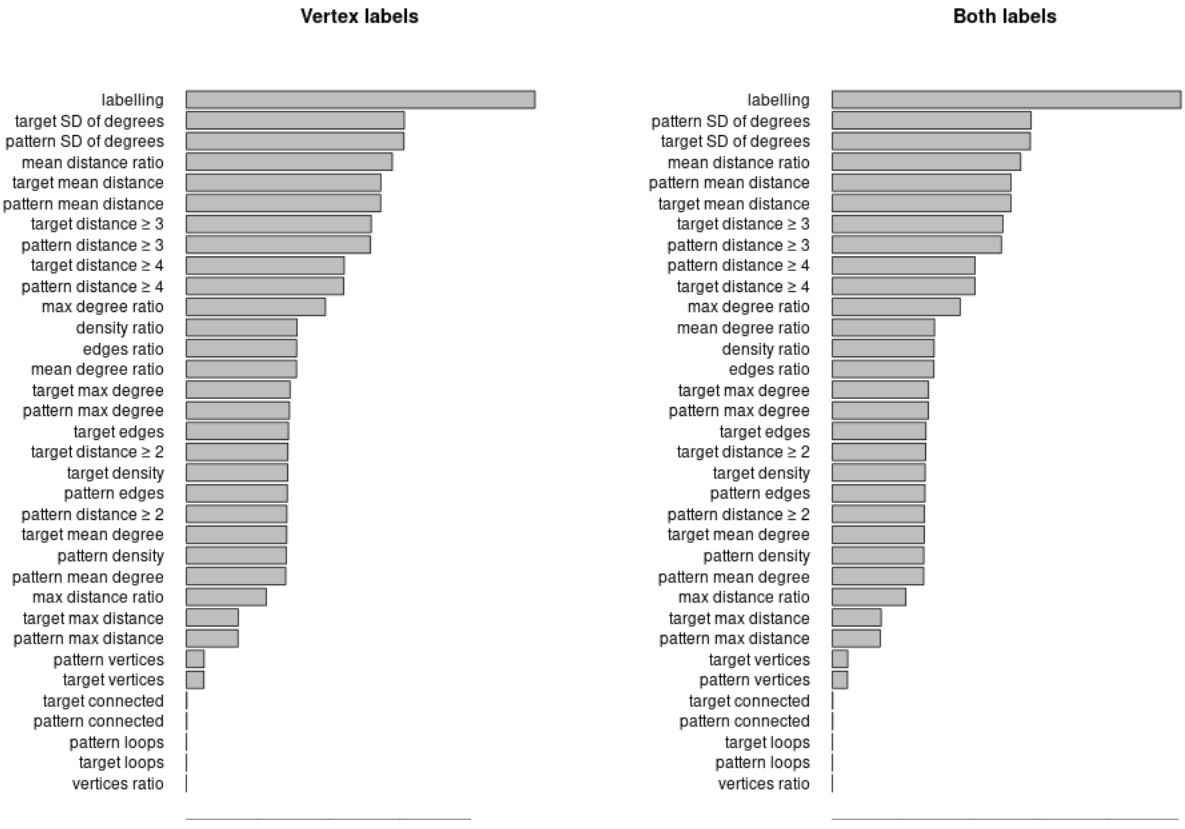


Figure 4.9: How often was each variable used to make splitting decisions?

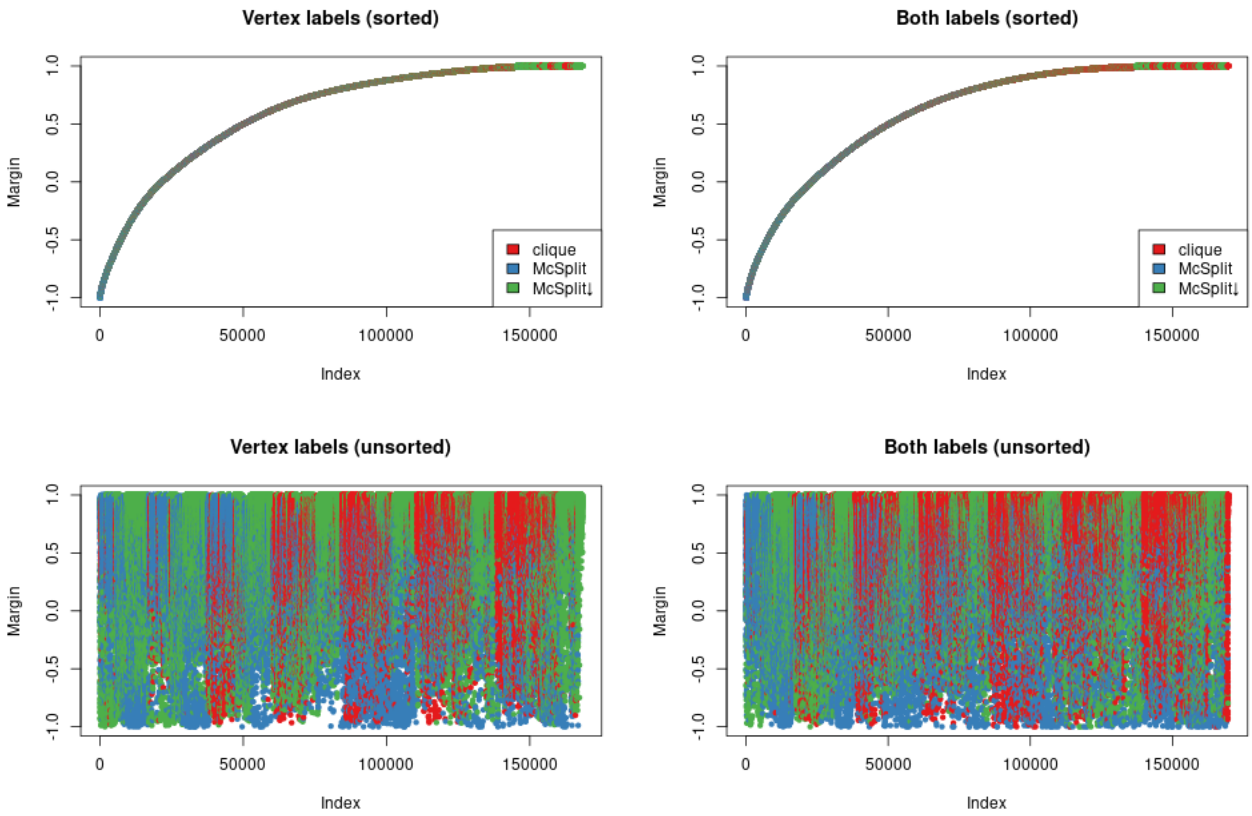


Figure 4.10: Sorted and unsorted margins of all data points, for both types of labelling.

both cases there is little difference between the plots for vertex-labelled and fully labelled graphs. Consider the labelling plots first. The results for the clique encoding are the most straightforward to interpret. Clearly, the algorithm performs much better with higher labelling percentages (more different labels). The interesting bit (just like back in Figure 3.3) is that the curve stays constant between 20% and 50%. In other words, anywhere above 20% labelling, a higher labelling percentage does not make the clique encoding more preferable than it already is. Furthermore, once again we can recognise how the curve descends more slowly for both labels than it does for vertex labels. On the other hand, both MCSPLIT and MCSPLIT↓ prefer lower labelling percentages.

The partial dependence on the standard deviation of degrees of the target graph says what we already knew from Section 4.1.4: the clique encoding prefers graphs with more variance in degrees. The one small difference is that this time the change is not as steep, but this can be easily explained by the fact that the highest standard deviation of degrees is around 7 in this section compared to around 70 in Section 4.1.4. While MCSPLIT↓ prefers graphs with smaller standard deviations of degrees, the situation with MCSPLIT is suddenly less clear, which is reflective of the fact that the ML models have less confidence about MCSPLIT (compared to the other 2 algorithms). While small standard deviations are still preferred, there is a significant drop between the values of 1 and 2.

Finally, for the runtime comparison, we plot the ECDFs for all individual algorithms, the VBS, and the LLAMA models in Figure 4.14. While there are important differences among individual algorithms' curves (discussed in Section 3.1.2), the way LLAMA behaves with respect to the VBS and MCSPLIT↓ curves is about the same: LLAMA's curve is quite close to optimal, and the gap between LLAMA and MCSPLIT↓ is significantly wider than the gap between MCSPLIT↓ and MCSPLIT. To add more numbers to this picture, LLAMA closed 86% and 88% of the difference between its lower and upper bounds for vertex-labelled and fully labelled graphs, respectively.

difference is negligible and very likely to be due to randomness.

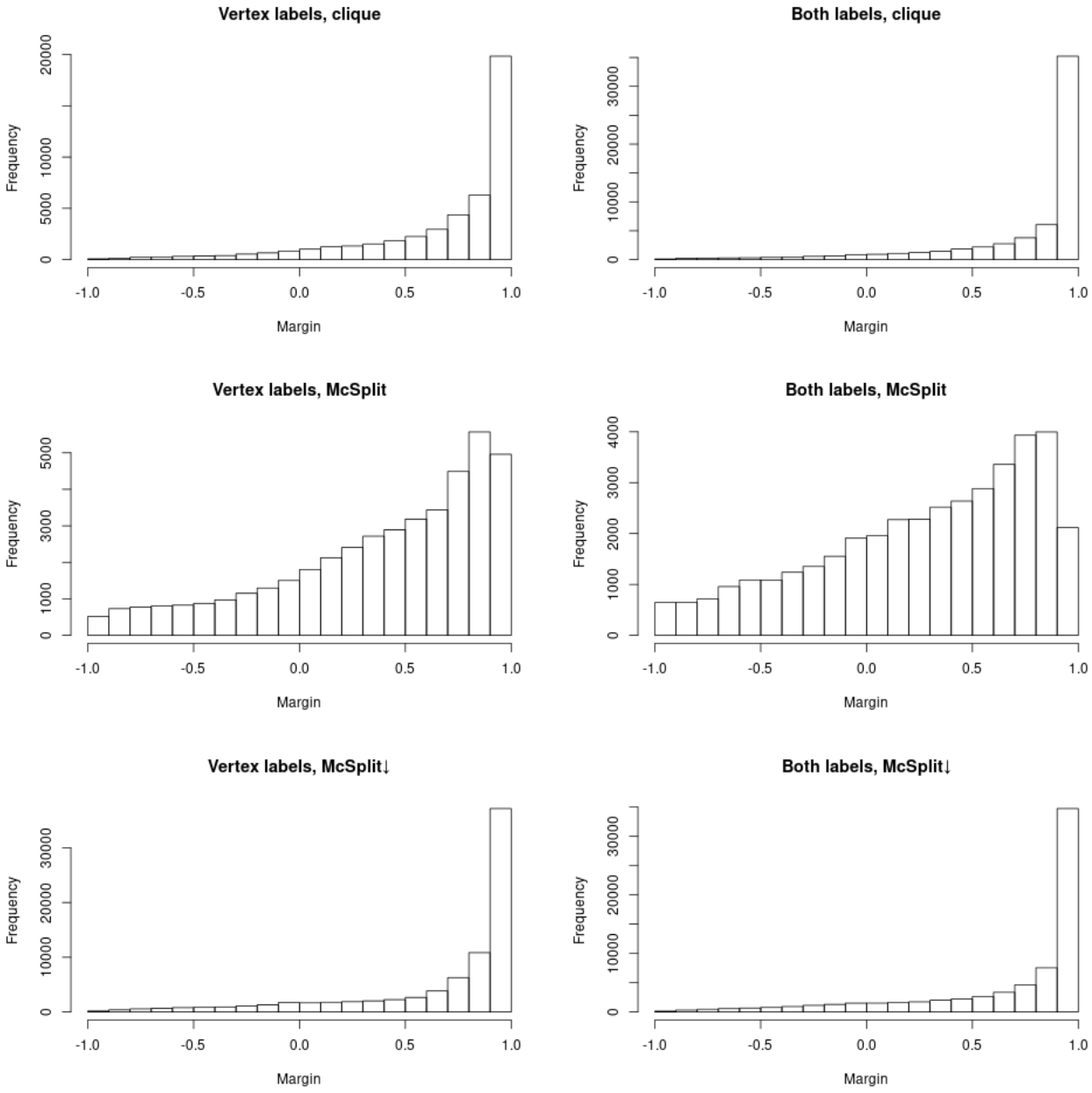


Figure 4.11: Histograms of margins, for each algorithm and type of labelling.

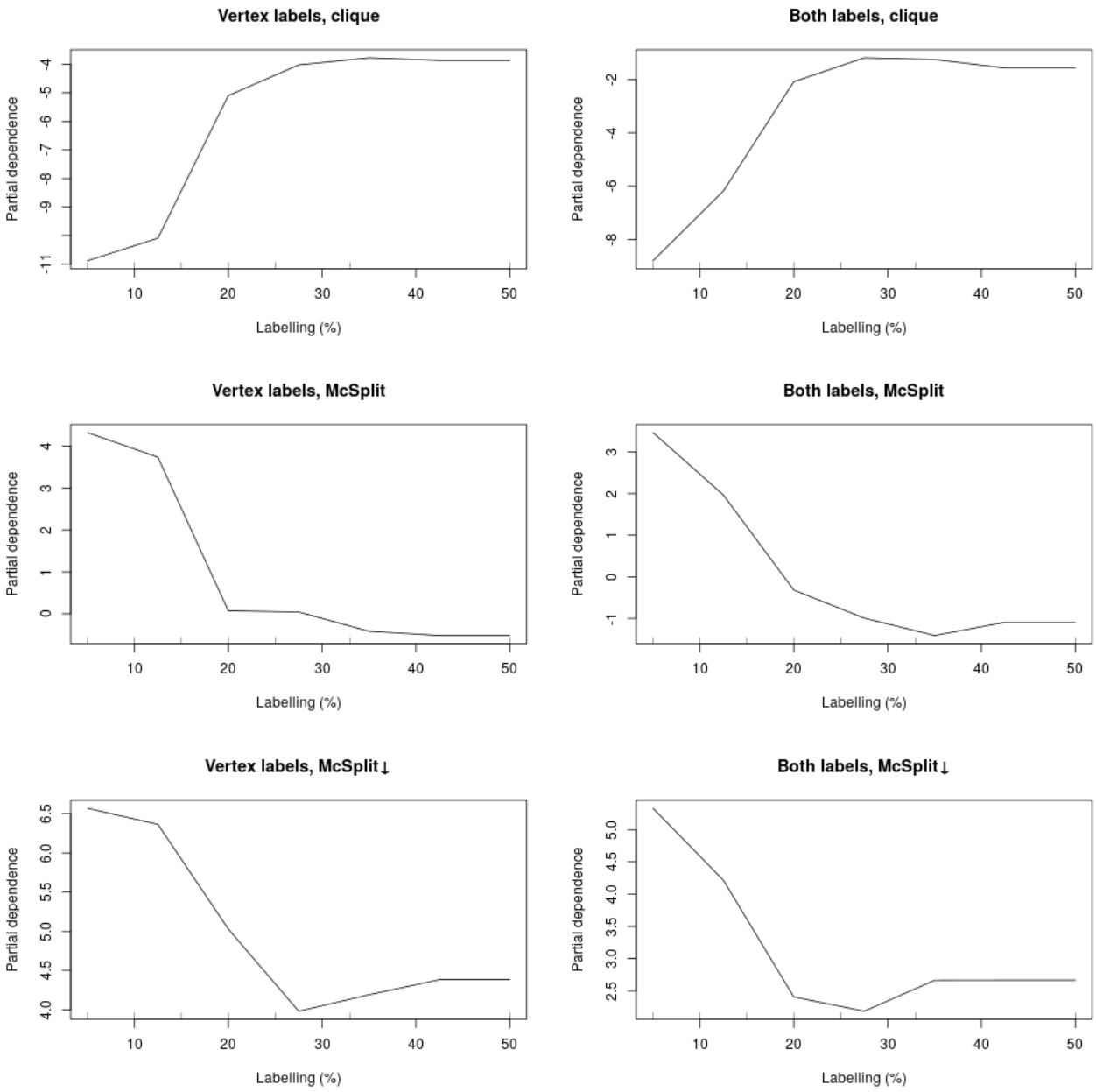


Figure 4.12: Partial dependence plots of the labelling percentage.

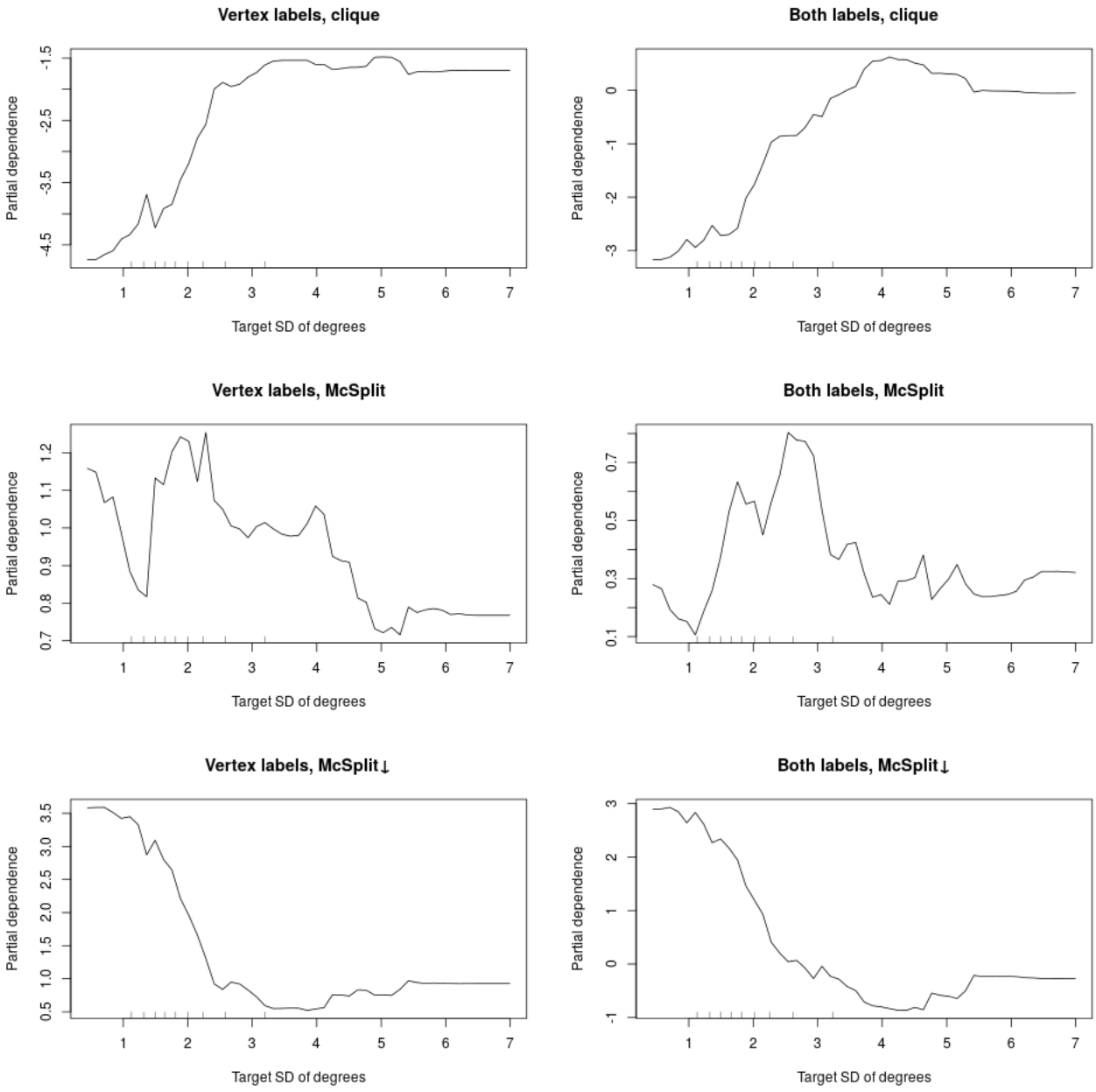


Figure 4.13: Partial dependence plots of the standard deviation of degrees in the target graph.

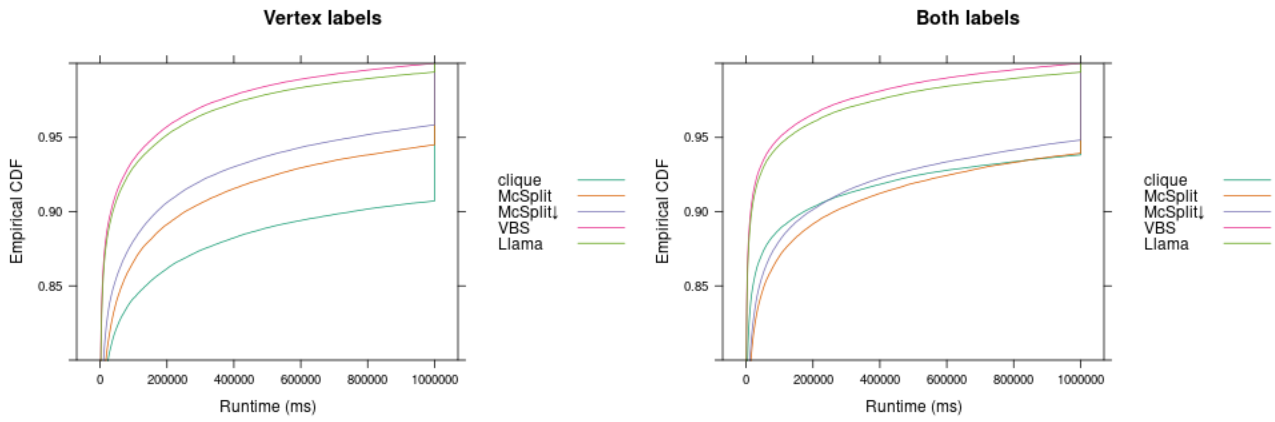


Figure 4.14: LLAMA models compared to other algorithms and the VBS.

Chapter 5

Further Suggestions, Investigations, and Experiments

In this chapter we build on the knowledge gained by developing ML models and analysing algorithms' performance to extend current algorithms and propose new ones. We suggest a way to extend $k\downarrow$ to support vertex-labelled graphs in a non-trivial manner, and introduce the idea of combining McSplit and the clique encoding into a single algorithm.

5.1 Modifications to $k\downarrow$

The $k\downarrow$ algorithm was modified to accept graphs with vertex labels by adding an additional constraint for matching labels on line 8 of the `klessSubgraphIsomorphism` function [20] and extending the `Graph` class with two new fields: a `vector`, assigning a label to each vertex, and a `vector` of `vectors`, which, for each label, list all vertices that have that label. After generating most of the data, it was noticed that the vertex-labelled version of $k\downarrow$ was winning precisely 0 times and thus was not considered in Sections 3.1.2 and 4.2.

In order to make $k\downarrow$ more competitive, the neighbourhood degree sequence filtering was improved to make use of the additional information that labels provide. It is also part of line 8 of the same function in the original paper [20], however, it is not enabled by default[1] and is not used in the experiments of the McSplit paper [39]. We take the definition directly from the $k\downarrow$ paper [20]:

Definition 5.1. "The *neighbourhood degree sequence* (NDS) of a vertex p, $S(p)$, is the (non-ascending) sequence of degrees of its neighbours."

Definition 5.2. For two sequences $S = (s_1, \ldots, s_n)$ and $T = (t_1, \ldots, t_m)$ and some non-negative integer k, we say that $S \preceq_k T$ if $n - k \leq m$ and there is a subsequence S_k of S with $|S_k| \leq k$ such that [20]

$$\forall s_i \in S \setminus S_k, \text{ there exists a distinct } j \in \{1, \ldots, m\} \text{ such that } s_i - k \leq t_j.$$

As a simplification of the definition above, we state the following lemma, based on Corollary 1 from the $k\downarrow$ paper [20].

[1] `https://github.com/ciaranm/aaai17-between-subgraph-isomorphism-and-maximum-common-subgraph-paper`

Lemma 5.1. *Let $S = (s_1, \ldots, s_n)$ and $T = (t_1, \ldots, t_m)$ be two sequences and k an integer such that $k \geq \max\{0, n - m\}$. Then $S \,{}_k\preceq T$ if and only if*

$$s_i \leq t_{i-k} + k \quad \forall i = k + 1, \ldots, n.$$

Proof. A simple extension of Corollary 1 from the $k\downarrow$ paper [20] by a reordering argument. □

We also have Proposition 3 from the same paper [20], which is useful to our proposed algorithm:

Proposition 5.1. *Given some k, if a pattern graph vertex p is mapped to a target graph vertex t in a subgraph isomorphism that excludes up to k vertices from the pattern graph, then*

$$S(p) \,{}_k\preceq S(t).$$

Data: a non-negative integer k, a list of NDSs for each label pNds, a list of NDSs for each label tNds
Result: a Boolean value, indicating whether p and t can be matched

```
1  NeighboursUniqueToPattern ← Σ_{i=|tNds|}^{|pNds|−1} |pNds[i]|;
2  if NeighboursUniqueToPattern > k then return false;
3  for i ← 0 to min{|pNds|, |tNds|} − 1 do
4      if |pNds[i]| − |tNds[i]| > 0 then
5          NeighboursUniqueToPattern ← NeighboursUniqueToPattern + |pNds[i]| − |tNds[i]|;
6          if NeighboursUniqueToPattern > k then return false;
7      end
8      if pNds[i] ⋠_k tNds[i] then return false;
9  end
10 return true;
```

Algorithm 1: NDS filtering with vertex label support.

We extend these ideas in Algorithm 1 by considering an NDS for each label. The algorithm checks whether a vertex p in the pattern graph can be matched with a vertex t in the target graph. NeighboursUniqueToPattern tracks the number of neighbours of p that have to be excluded, i.e., whenever the pattern graph has more neighbours with a particular label than the target graph, we can add the difference to NeighboursUniqueToPattern, and if it ever becomes larger than k, we know that it is impossible to map p to t. The other early exit condition is the contrapositive of Proposition 5.1. If neither of these conditions is satisfied throughout the **for** loop that starts on line 5, then p and t are compatible for this value of k.

As this improvement is an afterthought after the main experiments, the resulting algorithm was run on a smaller subset of all data: 30,000 instances with 10% labelling. The ECDF plot in Figure 5.1 shows the new $k\downarrow$ to be just slightly behind the clique encoding and it outperforms other algorithms in 193 instances, making it worth consideration for an inclusion in a portfolio, although it might achieve similar results as with unlabelled graphs, where it wins too infrequently to be a valuable inclusion.

5.2 Algorithm Switching

Given the fact that different algorithms perform best on different instances, one might ask: if algorithm A performs best on some instance I, does A remain best throughout execution, or could it benefit us to switch to a different algorithm mid-execution? More specifically, we investigate the idea of switching from MCSPLIT to the clique encoding, hoping that several decisions made by the MCSPLIT algorithm simplify the remaining problem so much that the corresponding association graph becomes easily solvable by the clique algorithm.

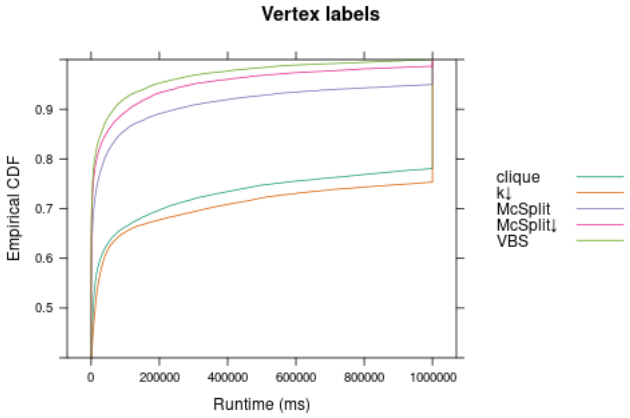


Figure 5.1: Cumulative plot of all algorithms with 10% vertex labelling.

5.2.1 The Clique Algorithm and its Association Graph

We begin with some preliminary analysis of properties of association graphs and their relationship with the clique algorithm's running time. We choose some of the features described in Section 3.2 to record about association graphs. More specifically, we record features that are easy to compute (since association graphs can get quite big) and can provide meaningful information: number of vertices, number of edges, mean degree, maximum degree, standard deviation of degrees, and density. We record this information for association graphs created for all unlabelled instances from Section 2.1 (since the graphs from Section 2.2 often stretch the memory limits) as well as the 30,000 instances previously selected for experiments with labelled graphs, for all seven labelling percentages. As the timed-out experiments would make interpreting any results much harder, we remove them from the data and are left with about 300,000 (instead of about 500,000) observations. The distribution of each feature is plotted in Figure 5.2.

To keep things simple and not violate assumptions about independent variables, we will only consider correlations between the variables instead of trying to fit some kind of a regression model. The Spearman's rank correlation coefficients between all measured variables are in Figure 5.3. Note that all variables are positively correlated: some correlations are very strong, some average, and one very weak (between density and the standard deviation of degrees). Also note that all measured characteristics of the association graph are positively correlated with the algorithm's running time.

After a basic overview of what association graphs are like in terms of their characteristics and correlations between them, we are ready to tackle the main question of this section: can we predict the instances where the clique encoding is better than other algorithms based on properties of the association graph? For each of the six measured properties, we divide its range of values to 100 bins of equal width (some of them turn out to be empty). For each bin, we plot the proportion of instances in that range with the clique encoding outperforming other algorithms in Figure 5.4. We can make three observations:

- For all measured properties except density, the clique algorithm is not just overall faster, but also faster than other algorithms (i.e., more likely to win) with smaller vales.
- The situation with density is less clear. Other than higher winning rates with density smaller than 0.3, density of the association graph tells us little about the algorithm's winrate.
- Except for the low density values described in the previous point, the clique algorithm's winning rate stays below 0.5, meaning that simply looking at each measured property of the association graph is not enough to identify values of those properties that would provide a clear indication that re-encoding the MCS instance into an instance of maximum clique would provide clear benefits.

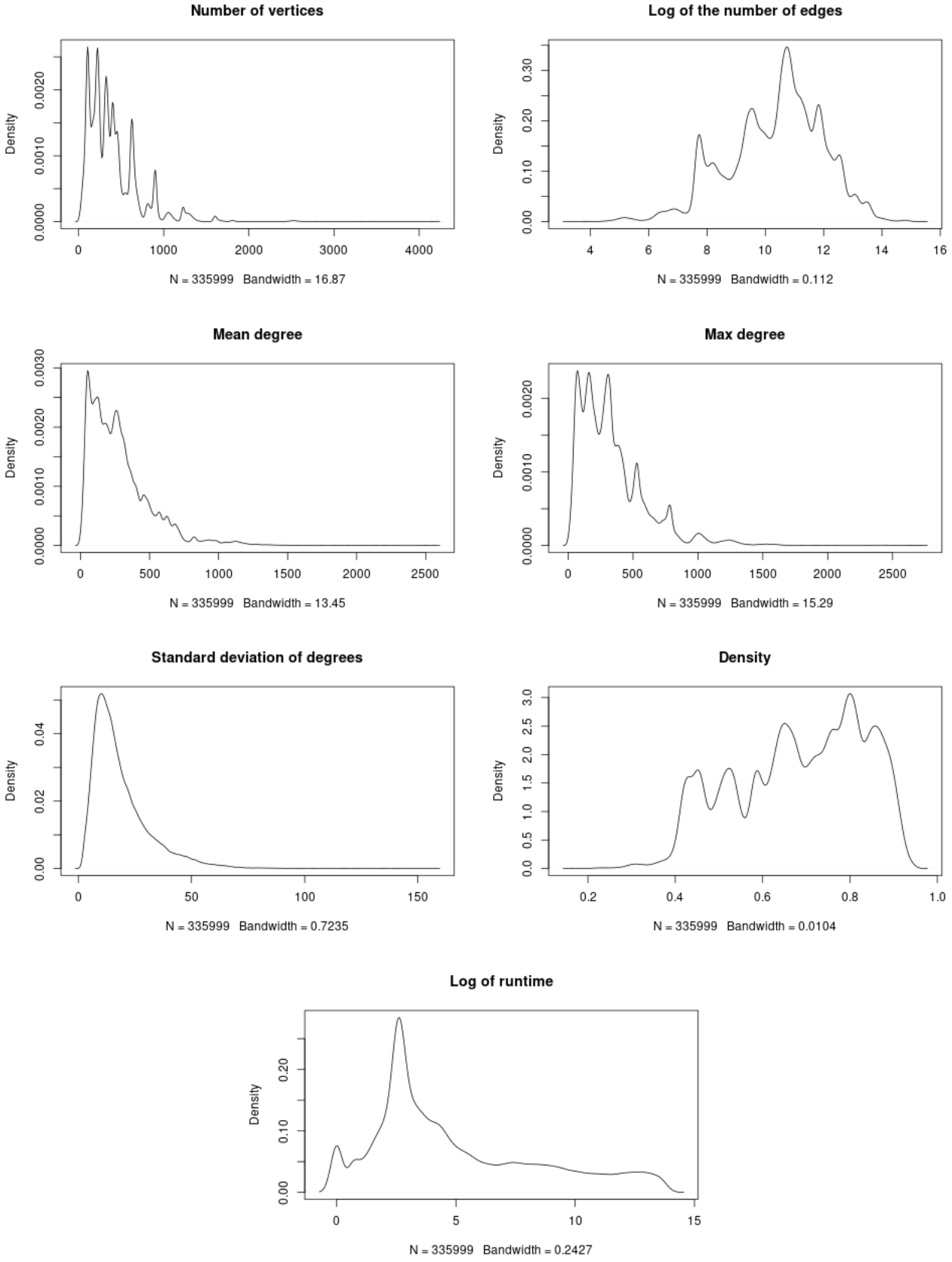


Figure 5.2: Density plots of the selected features of association graphs and the clique algorithm's runtime. Note that the number of edges and runtime were log-transformed in order to improve clarity.

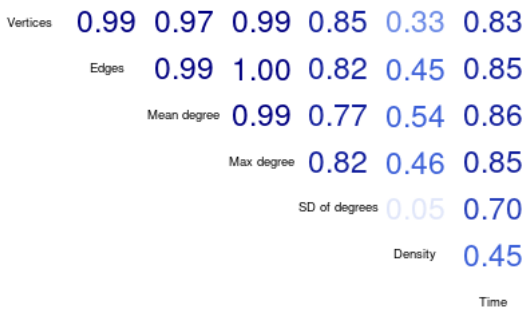


Figure 5.3: Spearman's rank correlation coefficients between the measured variables.

Remark 5.1. The reader might notice that some bins start at negative values, even though the feature itself cannot be negative. This is because the left endpoint of the leftmost interval is defined as

$$\min(X) - 0.001 \times (\max(X) - \min(X)),$$

where X is the set of observed values of some particular feature [54] (and similarly for the right endpoint of the rightmost interval).

5.2.2 FUSION: MCSPLIT and the Clique Encoding United

Even though we were not able to justify re-encoding a problem instance that is partially solve by MCSPLIT into the clique encoding using the simple techniques of Section 5.2.1, this section explores the simplest way of merging the two algorithms, where the clique algorithm overtakes MCSPLIT after making a fixed number of choices. The hope is that making several decision with MCSPLIT would simplify the association graph enough to warrant the extra cost in creating the association graph and switching algorithms.

Firstly, we add an additional argument to the `Search` function of the MCSPLIT algorithm [39] called depth with a default (initial) value of 0. It is increased by one in the recursive call on line 19 of Algorithm 1 in the MCSPLIT paper [39] and left unchanged in a different recursive call on line 23.

Secondly, we also add a global parameter DEPTH, which denotes at what value of depth MCSPLIT re-encodes the remaining problem to a maximum clique instance. A special value of DEPTH $= -1$ makes FUSION essentially equal to MCSPLIT in all its behaviour. Similarly, DEPTH $= 0$ is similar to running the clique algorithm, but, unlike in the implementation of the clique algorithm, association graph's creation time is included as part of total running time.

Lastly, similarly to the ∇ operator from Definition 1.15, which creates the association graph from two graphs, we create a version of ∇ that works on the list of label classes called *future* in the MCSPLIT paper [39]. The set of vertices of the association graph is then defined as

$$\bigcup_{\langle G,H\rangle \in future} G \times H,$$

and the edges are defined in exactly the same way as in the original definition.

Given the two new variables depth and DEPTH, we only need to add the following lines of code between lines 5 and 6 of Algorithm 1 in the MCSPLIT paper [39] to transform MCSPLIT into FUSION:

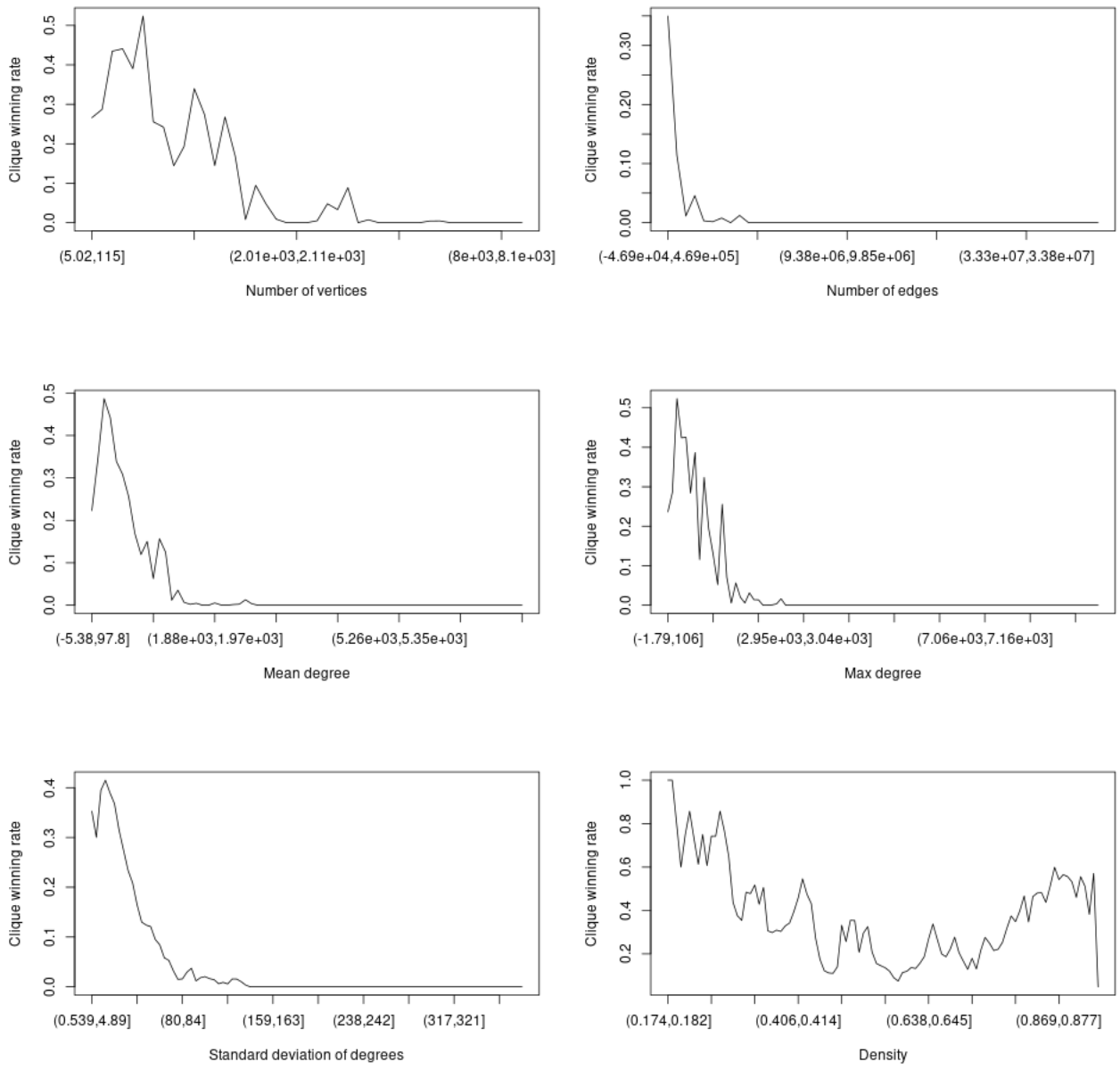


Figure 5.4: How often does the clique encoding outperform other algorithms at different values of properties of the association graph?

if DEPTH $\neq -1$ **and** depth $\geq$ DEPTH **then**
  $|incumbent_{\text{clique}}| \leftarrow |incumbent_{\text{McSplit}}| - |M|$;
  `search`$(\nabla(future), \emptyset, \emptyset, V_1)$;
  if $|M| + |incumbent_{clique}| > |incumbent_{\text{McSplit}}|$ **then**
    $incumbent_{\text{McSplit}} \leftarrow incumbent_{\text{McSplit}} \cup incumbent_{\text{clique}}$;
end

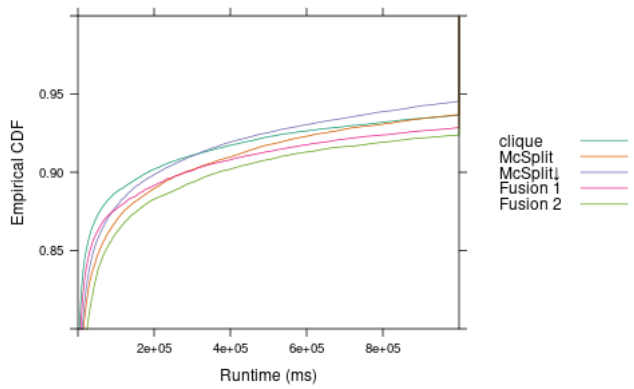


Figure 5.5: Cumulative plot of the main three MCS algorithms with two versions of FUSION: FUSION 1 and FUSION 2, that build the association graph after one and two branching decisions, respectively.

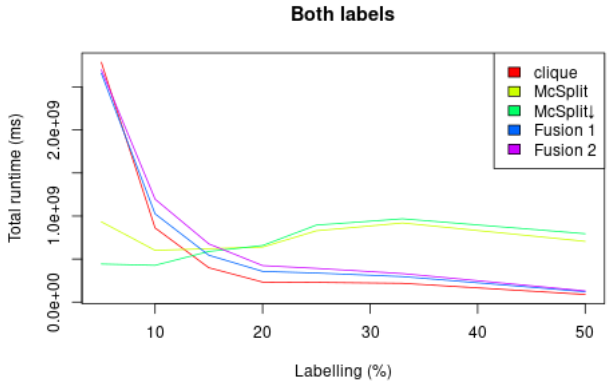


Figure 5.6: Total runtime of each algorithm, for each labelling percentage.

Here, $incumbent_{\text{MCSPLIT}}$ and $incumbent_{\text{clique}}$ denote the different variables with the same name in the two algorithms, M and *future* come from the MCSPLIT algorithm, V_1 is the set of vertices of one of the original graphs, used in the initialisation of the clique algorithm, and `search` is the main search function of the clique algorithm [36]. Note that due to how the incumbents are implemented in practice, we set the cardinality of $incumbent_{\text{clique}}$ without having to set the incumbent itself.

The algorithm was run on 10,000 randomly-selected instances with the labelling percentage varying between 5% and 50% (just like in Section 3.1) and the resulting ECDF plot in Figure 5.5 shows that:

- FUSION 2 is strictly worse than FUSION 1. Therefore we do not run experiments with versions of FUSION that switch algorithms after even more decisions.
- FUSION 1 is second best for very small runtimes, but gradually falls to the 4$^{\text{th}}$ place.
- At all points in the plot, FUSION 1 is worse than the original clique encoding.

Figure 5.6 helps to clarify the picture by showing differences in performance of the old and the new algorithms, for each labelling percentage. Keeping in mind that in this plot low total runtime values represent well-performing algorithms, we can see that the original clique algorithm is better than FUSION 1, which is better than FUSION 2, for all labelling percentages, except for small differences when all three algorithms' performance becomes terrible with 5% labelling. Another thing to note is that the differences between these three

algorithms is highest for average (10%–20%) labelling percentages, and becomes smaller when moving towards either extreme. This conclusively shows that switching between McSplit and the clique encoding at a certain depth of the search tree does not produce a competitive algorithm for solving MCS problem instances.

5.2.3 SmartFusion: Can We Switch Algorithms in a More Intelligent Manner?

Switching algorithms at a fixed depth of the search tree is one of the simplest possible strategies. In this section we explore an idea for how the decision whether to switch can be made using running time data of the two algorithms on various (unsolved) problem instances. For that purpose, we introduce a way to map partially solved instances inside McSplit into unsolved instances, and prove the method's correctness. For the rest of this section, consider an MCS problem instance between graphs $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, \mu_\mathcal{G}, \zeta_\mathcal{G})$ and $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H}, \mu_\mathcal{H}, \zeta_\mathcal{H})$ with $\mu_\mathcal{G} \colon V_\mathcal{G} \to \{0, \dots, N-1\}$ and $\mu_\mathcal{H} \colon V_\mathcal{H} \to \{0, \dots, N-1\}$. We start by formalising how each algorithm "sees" a problem instance (obviously, the definitions are heavily based on the two papers describing these algorithms [36, 39]).

Definition 5.3. A McSplit *instance* is a 5-tuple $(\textit{future}, M, \textit{incumbent}, \mathcal{G}, \mathcal{H})$, where:

- $\textit{future} = \{\langle G, H\rangle : G \subseteq V_\mathcal{G}, H \subseteq V_\mathcal{H}\}$ represents the fact that every vertex in G can only be matched to vertices in H and vice versa. Initially, $\textit{future} = \{\langle \mu_\mathcal{G}^{-1}(l), \mu_\mathcal{H}^{-1}(l)\rangle : l \in \mu_\mathcal{G}(V_\mathcal{G}) \cap \mu_\mathcal{H}(V_\mathcal{H})\}$[2].
- $M = \{(v, w) : v \in V_\mathcal{G}, w \in V_\mathcal{H}\}$ contains the pairs of vertices that have already been matched.
- $\textit{incumbent} = \{(v, w) : v \in V_\mathcal{G}, w \in V_\mathcal{H}\}$ is the maximum common subgraph known so far.
- $\mathcal{G}$ and $\mathcal{H}$ are the original graphs.

We will call this an *unsolved* McSplit *instance* if $M = \emptyset$ and a *partially solved instance* otherwise.

Remark 5.2. We include both graphs in the definition to make subsequent observations and statements both simpler and more useful. In reality, $\mathcal{G}$ and $\mathcal{H}$ are accessible from inside the McSplit algorithm, although the information about vertices and vertex labels is already encoded in the *future* variable, and the information about edges is stored in two two-dimensional arrays, that can be thought of as functions $A_\mathcal{G}, A_\mathcal{H}$, where $A_\mathcal{G} \colon V_\mathcal{G} \times V_\mathcal{G} \to \mathbb{M}(\{0, \dots, N-1\})$ encodes the multiset of labels on edges between any two vertices (and similarly for $A_\mathcal{H}$). We use the notation $\mathbb{M}(S)$ to denote the (infinite) set of multisets with elements from S, where each element of S can be used any number of times. Note that $\emptyset \in \mathbb{M}(S)$, for any set S. For any two $u, v \in V_\mathcal{G}$, $A_\mathcal{G}(u, v)$ is the multiset image of the edges between u and v under $\zeta_\mathcal{G}$.

Definition 5.4. A *clique instance* is a tuple $(G, \textit{incumbent})$, where $G = \mathcal{G} \nabla \mathcal{H} = (V, E)$ is the association graph, and $\textit{incumbent} \subseteq V$ is the maximum clique in the association graph known so far.

An *unsolved instance* is characterised by a pair of graphs and a non-negative integer $(\mathcal{G}, \mathcal{H}, k)$. Here, k is used to set up the initial size of the *incumbent*. An unsolved instance $(\mathcal{G}, \mathcal{H}, k)$ can be transformed to a clique instance $(\mathcal{G} \nabla \mathcal{H}, \textit{incumbent})$, and to a McSplit instance $(\textit{future}, \emptyset, \textit{incumbent}, \mathcal{G}, \mathcal{H})$, where *incumbent* is any set of cardinality k, and *future*, $\mathcal{G}$, $\mathcal{H}$ are all initialised as in Definition 5.3. It is also important to remember from the previous section that a McSplit instance $(\textit{future}, M, \textit{incumbent}, \mathcal{G}, \mathcal{H})$ can be transformed into a clique instance $(\nabla(\textit{future}), X)$, where X is any set of cardinality $|\textit{incumbent}| - |M|$.

[2]Technically, in order to account for loops, we would have to have a $\langle G, H\rangle$ pair for each unique combination of vertex label and multiset image of loops under the edge-labelling function (where $G \neq \emptyset \neq H$) (the same way the clique algorithm creates vertices of the association graph). In order to use more succint notation, and avoid long and convoluted sentences, we will stick with this simplification.

Definition 5.5. We call a partially solved instance $(\textit{future}, M, \textit{incumbent}, \mathcal{G}, \mathcal{H})$ *equivalent* to an unsolved instance $(\mathcal{G}', \mathcal{H}', k)$ if

- MCSPLIT takes the same path of execution on both instances (transforming an unsolved instance into an unsolved MCSPLIT instance as described earlier),
- and the clique algorithm takes the same path of execution on a clique instance constructed from $(\mathcal{G}', \mathcal{H}', k)$ as on $(\nabla(\textit{future}), X)$, where X is any set of cardinality $|\textit{incumbent}| - |M|$.

Proposition 5.2. *Let $(\textit{future}, M, \textit{incumbent}, \mathcal{G}, \mathcal{H})$ be an arbitrary partially solved instance. Order the elements of future in an arbitrary way: $\langle G_1, H_1\rangle, \ldots, \langle G_n, H_n\rangle$. Let $V_{\mathcal{G}'} = \bigcup_{i=1}^{n} G_i$. Let $E_{\mathcal{G}'}$ be a subset of $E_{\mathcal{G}}$, where we remove each edge that has one of its endpoints in $V_{\mathcal{G}} \setminus V_{\mathcal{G}'}$. Let $v \in V_{\mathcal{G}'}$ be arbitrary. Then it belongs to exactly one of $G_1, \ldots, G_n$, because all G_i's are pairwise disjoint [39]. Suppose it is G_j. Let $\mu_{\mathcal{G}'}(v) = j - 1$. Since $v \in V_{\mathcal{G}'}$ was arbitrary, this defines a function $\mu_{\mathcal{G}'} : V_{\mathcal{G}'} \to \{0, \ldots, n-1\}$. Finally, let $\zeta_{\mathcal{G}'} = \zeta_{\mathcal{G}}|_{E_{\mathcal{G}'}}$. Then the partially solved instance $(\textit{future}, M, \textit{incumbent}, \mathcal{G}, \mathcal{H})$ is equivalent to an unsolved instance $(\mathcal{G}', \mathcal{H}', |\textit{incumbent}| - |M|)$, where $\mathcal{G}' = (V_{\mathcal{G}'}, E_{\mathcal{G}'}, \mu_{\mathcal{G}'}, \zeta_{\mathcal{G}'})$ and $\mathcal{H}'$ is set up in a similar way.*

Proof. We begin by considering how MCSPLIT behaves on each of the two instances. The unsolved instance gets transformed into a MCSPLIT instance $(\textit{future}', \emptyset, \textit{incumbent}', \mathcal{G}', \mathcal{H}')$, where *incumbent'* is any set of cardinality $|\textit{incumbent}| - |M|$, and $\textit{future}' = \{\langle \mu_{\mathcal{G}'}^{-1}(l), \mu_{\mathcal{H}'}^{-1}(l)\rangle : l \in \mu_{\mathcal{G}'}(V_{\mathcal{G}'}) \cap \mu_{\mathcal{H}'}(V_{\mathcal{H}'})\}$. We will show that *future* $=$ *future'* in three steps:

1. First, we show that *future* and *future'* contain the same elements of $V_{\mathcal{G}}$. All vertices of $\mathcal{G}$ mentioned in *future'* can be expressed as
$$\bigcup_{l \in \mu_{\mathcal{G}'}(V_{\mathcal{G}'}) \cap \mu_{\mathcal{H}'}(V_{\mathcal{H}'})} \mu_{\mathcal{G}'}^{-1}(l). \tag{5.1}$$
Since
$$V_{\mathcal{G}'} = \bigcup_{i=1}^{n} G_i$$
and $\mu_{\mathcal{G}'}(G_i) = \{i-1\}$ by the definitions of $V_{\mathcal{G}'}$ and $\mu_{\mathcal{G}'}$,
$$\mu_{\mathcal{G}'}(V_{\mathcal{G}'}) = \mu_{\mathcal{G}'}\left(\bigcup_{i=1}^{n} G_i\right) = \{0, 1, \ldots, n-1\}.$$
Similarly, $\mu_{\mathcal{H}'}(V_{\mathcal{H}'}) = \{0, 1, \ldots, n-1\}$, and thus $\mu_{\mathcal{G}'}(V_{\mathcal{G}'}) \cap \mu_{\mathcal{H}'}(V_{\mathcal{H}'}) = \{0, 1, \ldots, n-1\}$. Then (5.1) can be rewritten as
$$\bigcup_{l=0}^{n-1} \mu_{\mathcal{G}'}^{-1}(l) = \bigcup_{l=0}^{n-1} G_{l+1} = \bigcup_{l=1}^{n} G_l,$$
which is exactly what we wanted to prove. A similar argument can show that *future* and *future'* contain the same elements of $V_{\mathcal{H}}$.

2. Next, we show that two vertices of $\mathcal{G}'$ are in the same G_i' if and only if they are in the same G_j. Indeed, for any $v, w \in V_{\mathcal{G}'}$,
$$v, w \in G_i' \iff \mu_{\mathcal{G}'}(v) = \mu_{\mathcal{G}'}(w) \iff v, w \in G_{\mu_{\mathcal{G}'}(v)+1}.$$
Replacing G with H and $\mathcal{G}$ with $\mathcal{H}$ also gives us an equivalent property for vertices of $\mathcal{H}'$.

3. Finally, we show that a similar structure is preserved between a pair of vertices belonging to different graphs. For any $v \in V_{\mathcal{G}'}$ and $w \in V_{\mathcal{H}'}$,

$$v \in G'_i, w \in H'_i \text{ for some } i \iff \mu_{\mathcal{G}'}(v) = \mu_{\mathcal{H}'}(w) \iff v \in G_{\mu_{\mathcal{G}'}(v)+1}, w \in H_{\mu_{\mathcal{G}'}(v)+1}.$$

This completes the proof that *future* = *future'*.

For two McSplit instances to be equivalent, it remains to show that the algorithm behaves exactly the same way with $(\emptyset, \textit{incumbent'})$, as it does with $(M, \textit{incumbent})$. Note that the only places in the algorithm, where these two variables can affect how the algorithm executes, are in lines 3–5 in the original paper [39], where we compare $|M|$ with $|\textit{incumbent}|$ and $|M| + \sum_{\langle G,H\rangle \in \textit{future}} \min(|G|, |H|)$ with $|\textit{incumbent}|$. When replacing $(M, \textit{incumbent})$ with $(\emptyset, \textit{incumbent'})$, we replace $(|M|, |\textit{incumbent}|)$ with $(0, |\textit{incumbent}| - |M|)$, so lines 3–5 test the same inequalities with $|M|$ subtracted from both sides. Therefore, McSplit executes the two instances in an identical way.

The clique algorithm accepts the partially solved instance as $(\nabla(\textit{future}), X)$, and the unsolved instance as $(\mathcal{G}' \nabla \mathcal{H}', Y)$, where X and Y are any two sets of cardinality $|\textit{incumbent}| - |M|$. Similarly to the McSplit situation, the elements in X and Y do not affect the algorithm, as long as their cardinalities match. Thus it remains to show that $\nabla(\textit{future}) = \mathcal{G}' \nabla \mathcal{H}'$. Let $\nabla(\textit{future}) = (V, E)$ and $\mathcal{G}' \nabla \mathcal{H}' = (V', E')$. Then

$$V = \bigcup_{i=1}^{n} G_i \times H_i,$$

and

$$\begin{aligned} V' &= \{(v, w) \in V_{\mathcal{G}'} \times V_{\mathcal{H}'} : \mu_{\mathcal{G}'}(v) = \mu_{\mathcal{H}'}(w)\} \\ &= \{(v, w) \in V_{\mathcal{G}'} \times V_{\mathcal{H}'} : v \in G_i, w \in H_i \text{ for some } i\}. \end{aligned}$$

Now since

$$V_{\mathcal{G}'} = \bigcup_{i=1}^{n} G_i, \quad V_{\mathcal{H}'} = \bigcup_{i=1}^{n} H_i,$$

we get that $V' = V$. The rules for adding edges to the association graphs are the same. They refer to different graphs, but $E_{\mathcal{G}'}, E_{\mathcal{H}'}, \zeta_{\mathcal{G}'}$, and $\zeta_{\mathcal{H}'}$ are only modified to accomodate the removal of vertices. Therefore, since $V' = V$ and they define what vertices are considered when adding edges to the association graph, $E' = E$, $\nabla(\textit{future}) = \mathcal{G}' \nabla \mathcal{H}'$, and the association graph receives equivalent input with the unsolved instance as it does with the partially solved instance. □

Chapter 6

Conclusion and Future Work

We took several competitive algorithms for the MCS problem, and explored how their performance changes with the number of labels on vertices and edges. Then we constructed an extensive list of features to represent each pair of graphs, trained three different ML models, and showed two of them to have near-optimal performance, significantly outperforming previous algorithms. During that process, we uncovered new information on how labels are typically implemented, the weaknesses of making generalisations about all labelled instances from a very limited sample, and the distributions and correlations of features in various benchmark datasets. ML models allowed us to gain further insight into which features are the most important, which algorithm prefers what values of each feature, and how confidently we can predict when each algorithm should be chosen. We were then able to build on that knowledge in order to extend one of the algorithms to vertex-labelled instances, and propose the idea of switching between two algorithms during runtime. We implemented a simple version, capable of switching algorithms according to a fixed rule, and laid the theoretical groundwork towards making the decision for when to switch in a more intelligent, ML-based manner. Although a great deal of work has been done, we would like to end with a few suggestions for constructing new algorithm portfolios, analysing the performance data of graph algorithms, and building the algorithms of tomorrow.

Firstly, the set of features used for the ML models is far from perfect. A better performance should be achieved by removing at least a few of the worst-performing features. Many things can be calculated from a graph, some of them are likely to perform well as new features. In particular, since the standard deviations of degrees performed so well, other measures of dispersion such as the coefficient of variation should be considered. As mentioned earlier in the dissertation, considering transformations of feature data might not bring massive advantages, but could be worthwhile nonetheless.

Secondly, in this project, once we fix the number of different labels in a graph, all labels occur with equal probabilities. This may not be very representative of real-world data. Hence, along with varying how many labels a graphs has, one could also consider some probability distribution over the set of possible labels, making some of them more common than others.

Finally, it would be interesting to see an implementation of SMARTFUSION based on new runtime data. Even though our proposed model builds two graphs every time it has to make a decision, faster ways are likely to exist, once the gathered data enables us to see which features are important and should be tracked. Similarly, instead of creating a new association graph every time, the same graph could be modified. One last question that our ML models were not particularly successful in answering is "when is MCSPLIT↓ better than MCSPLIT?"

Appendices

Appendix A

Plots of Distributions of Features

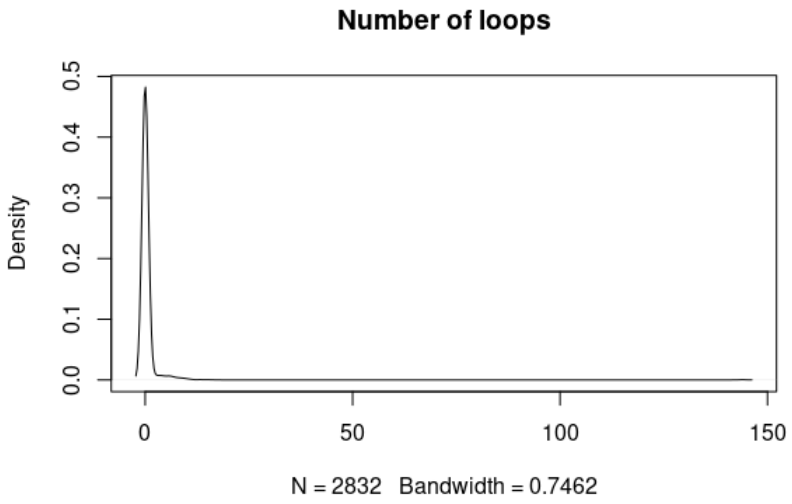


Figure A.1: Density plot of the number of loops in graphs from Section 2.2.

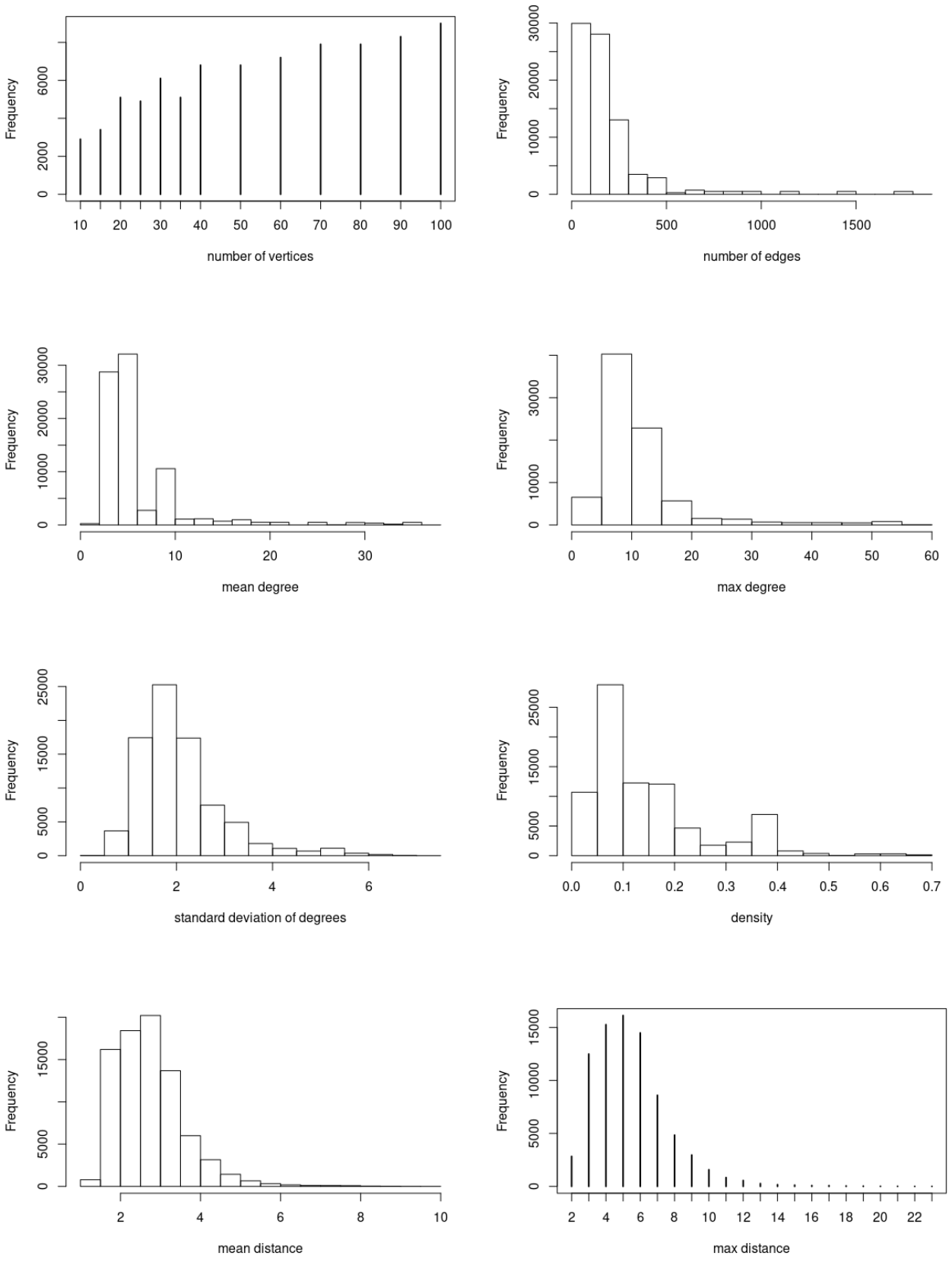


Figure A.2: Plots of how various features are distributed for graphs from Section 2.1.

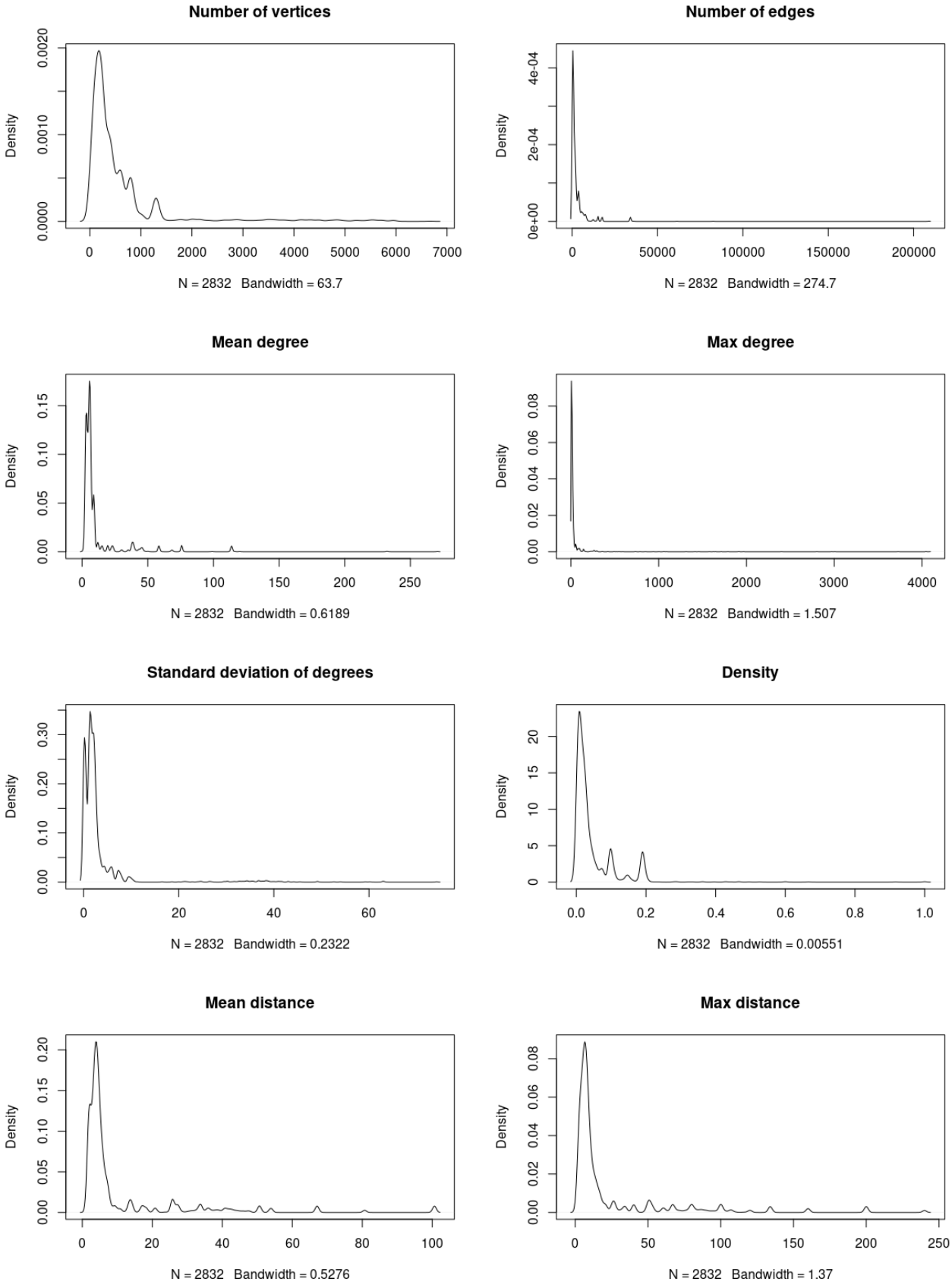


Figure A.3: Plots of how various features are distributed for graphs from Section 2.2.

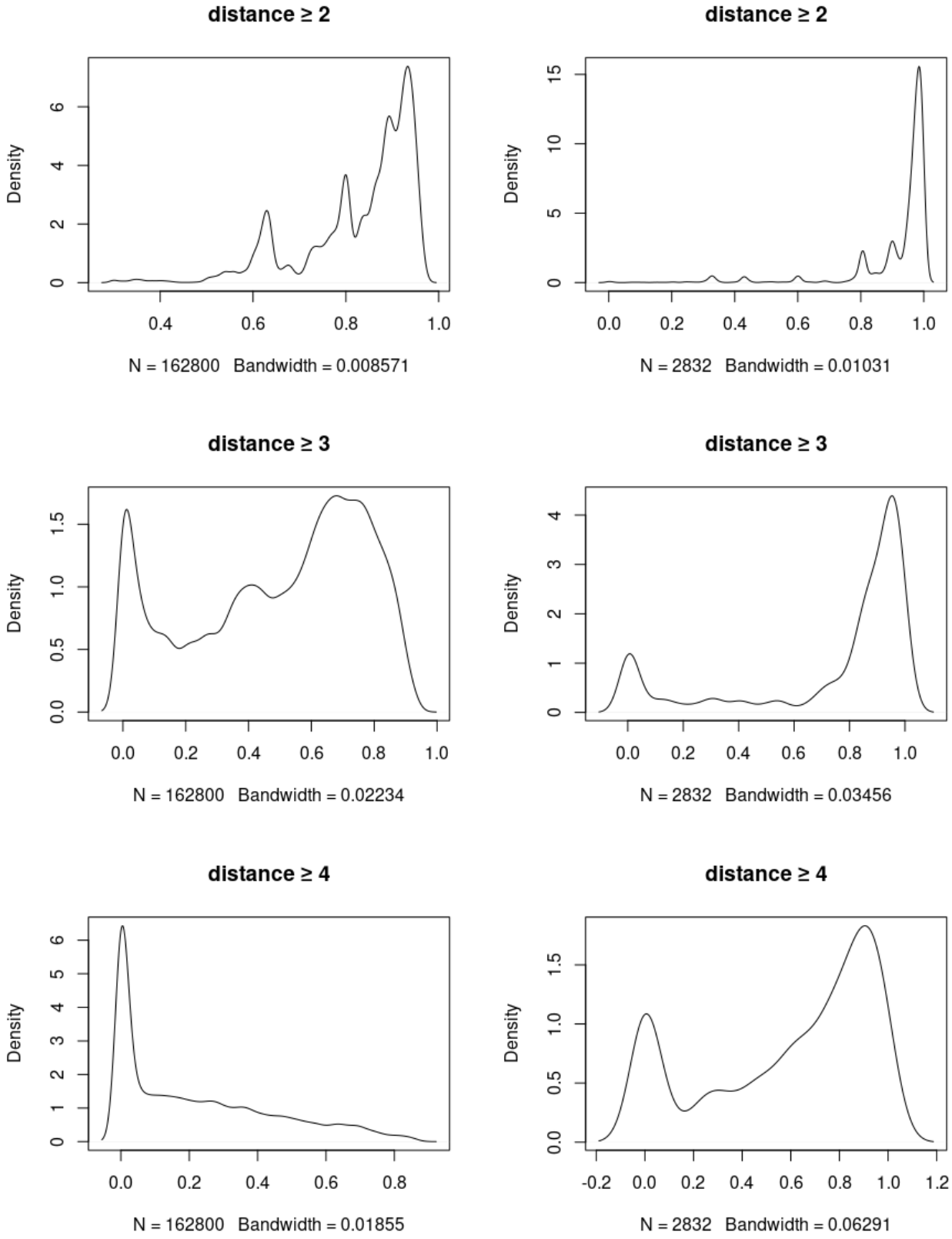


Figure A.4: Comparison of typical distances between pairs of vertices between the two graph databases, with graphs from Section 2.1 on the left, and graphs from Section 2.2 on the right.

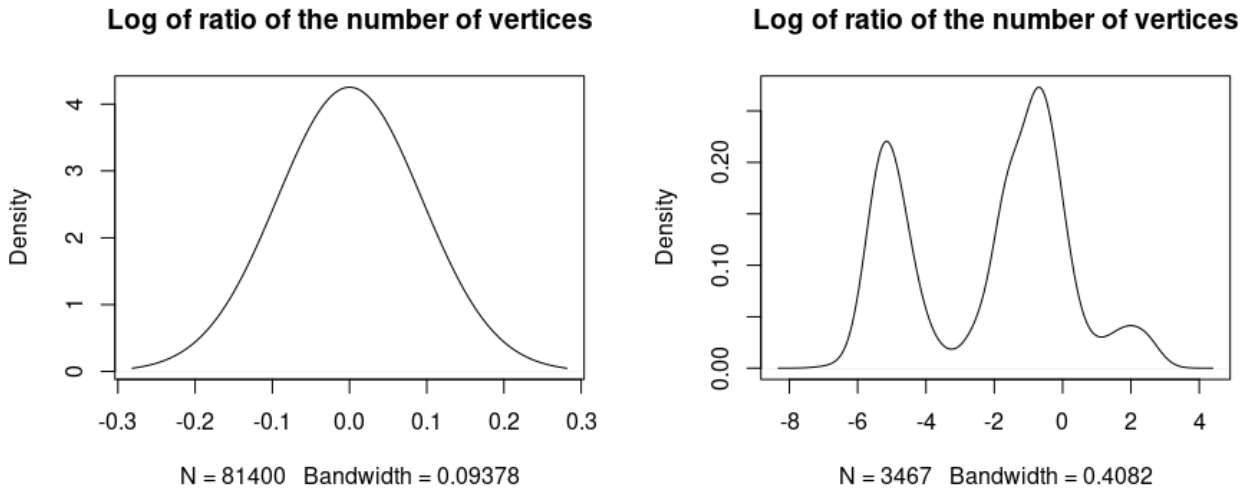


Figure A.5: The density plots of log-transformed ratio of the number of vertices between pattern and target graphs for both databases with graphs from Section 2.1 on the left and graphs from Section 2.2 on the right.

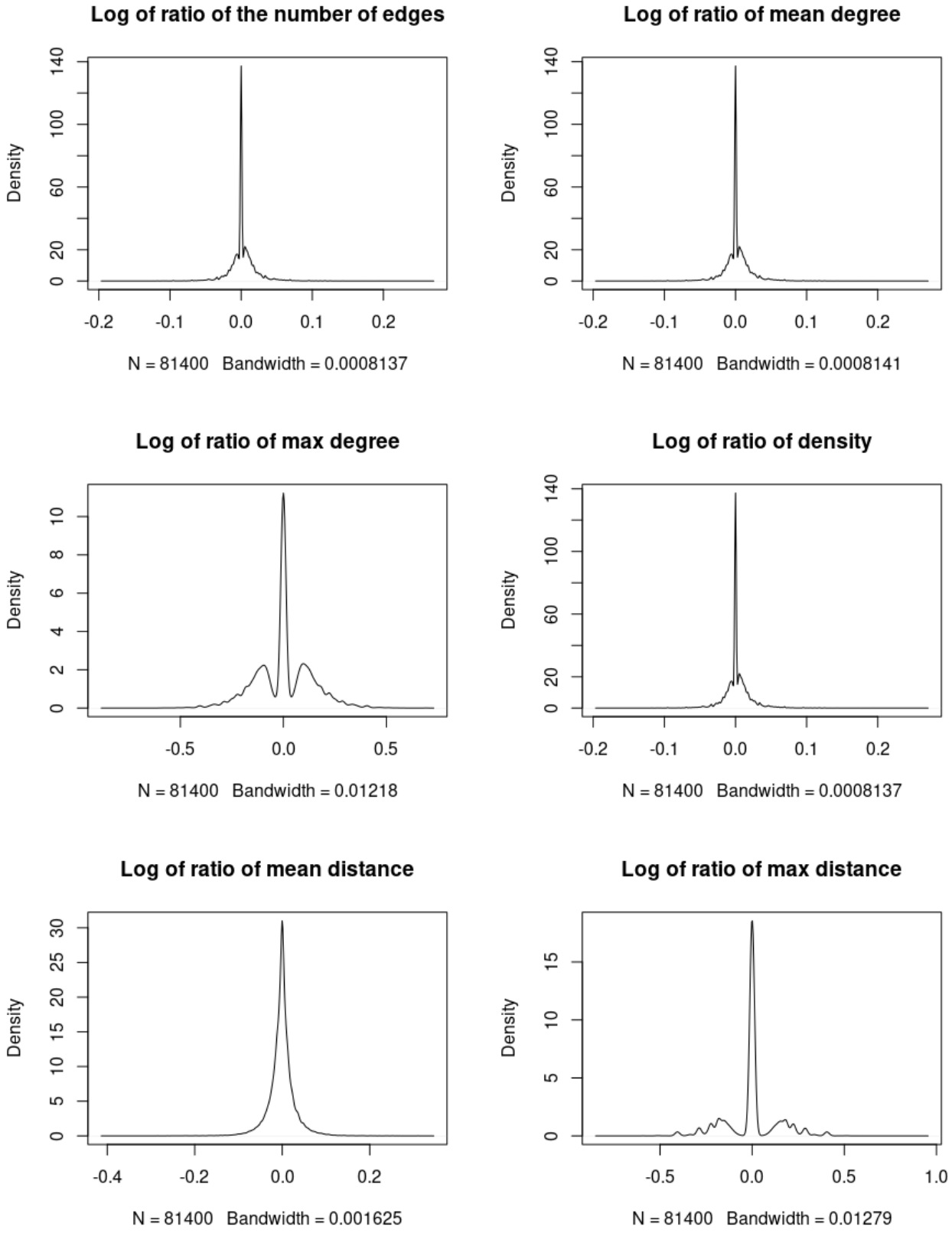


Figure A.6: The other density plots of the log-transformed ratio features for graphs from Section 2.1.

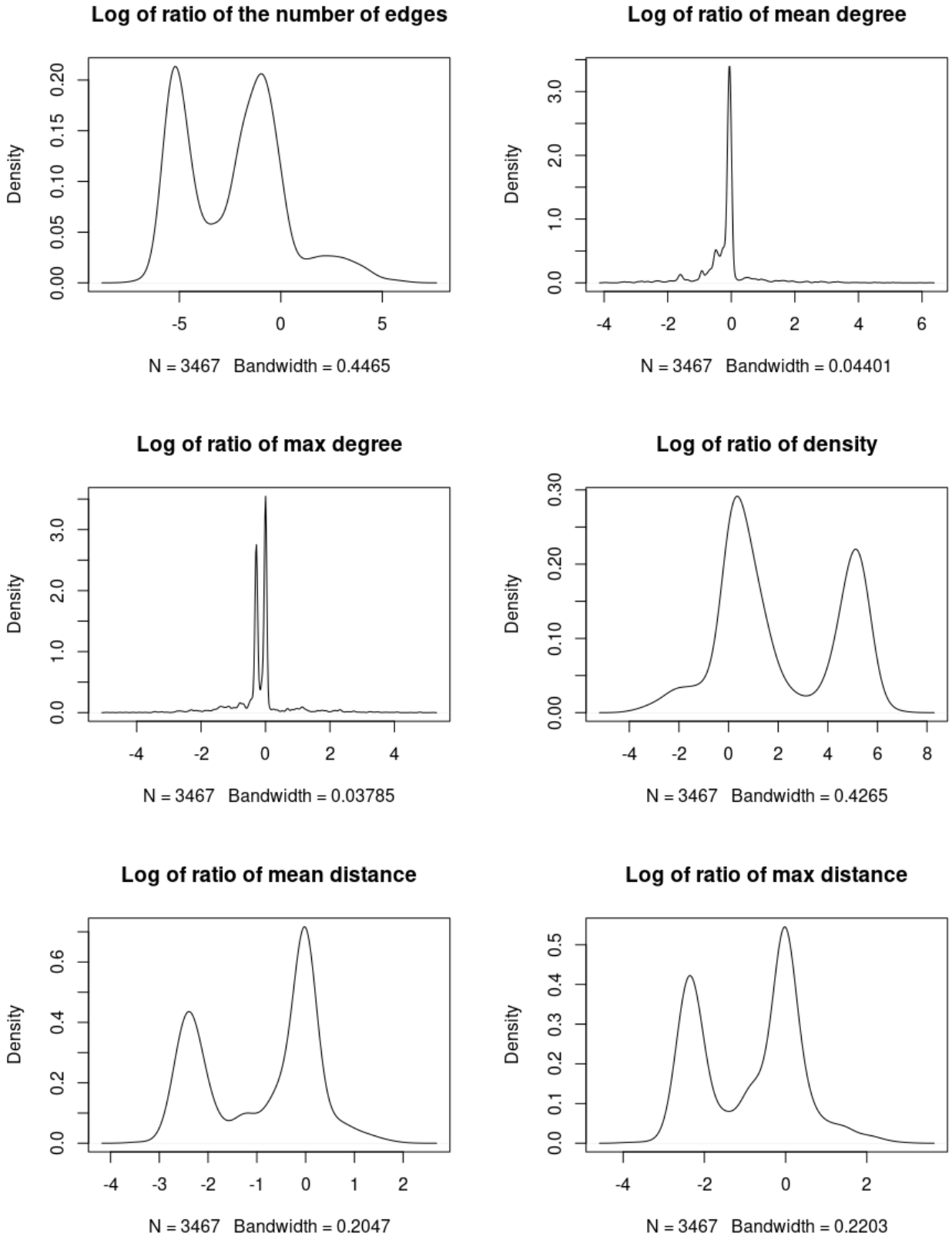


Figure A.7: The other density plots of the log-transformed ratio features for graphs from Section 2.2.

Bibliography

[1] Zeina Abu-Aisheh. *Anytime and Distributed Approaches for Graph Matching*. PhD thesis, Université François-Rabelais de Tours, 2016.

[2] László Babai, Paul Erdös, and Stanley M. Selkow. Random graph isomorphism. *SIAM J. Comput.*, 9(3):628–635, 1980.

[3] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Thomas Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger H. Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. ASlib: A benchmark library for algorithm selection. *Artif. Intell.*, 237:41–58, 2016.

[4] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] Horst Bunke, Pasquale Foggia, Corrado Guidobaldi, Carlo Sansone, and Mario Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In Terry Caelli, Adnan Amin, Robert P. W. Duin, Mohamed S. Kamel, and Dick de Ridder, 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, 2002.

[6] Edwin H. Connell. *Elements of Abstract and Linear Algebra*. University of Miami, December 2002.

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

[8] Donatello Conte, Corrado Guidobaldi, and Carlo Sansone. A comparison of three maximum common subgraph algorithms on a large database of labeled graphs. In Edwin R. Hancock and Mario Vento, editors, *Graph Based Representations in Pattern Recognition, 4th IAPR International Workshop, GbRPR 2003, York, UK, June 30 - July 2, 2003, Proceedings*, volume 2726 of *Lecture Notes in Computer Science*, pages 130–141. Springer, 2003.

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

[10] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[11] Guillaume Damiand, Christine Solnon, Colin de la Higuera, Jean-Christophe Janodet, and Émilie Samuel. Polynomial algorithms for subisomorphism of nD open combinatorial maps. *Computer Vision and Image Understanding*, 115(7):996–1010, 2011.

[12] Reinhard Diestel. *Graph Theory, 5th Edition*, volume 173 of *Graduate texts in mathematics*. Springer-Verlag, 2016.

[13] Surnjani Djoko, Diane J. Cook, and Lawrence B. Holder. An empirical study of domain knowledge and its benefits to substructure discovery. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):575–586, Jul 1997.

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

[15] Pasquale Foggia, Carlo Sansone, and Mario Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In *Proceedings of the 3rd IAPR TC-15 International Workshop on Graph-based Representations*, 2001.

[16] Jerome H. Friedman. Recent advances in predictive (machine) learning. *J. Classification*, 23(2):175–197, 2006.

[17] Tal Galili and Isaac Meilijson. Splitting matters: how monotone transformation of predictor variables may improve the predictions of decision tree models. *CoRR*, abs/1611.04561, 2016.

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

[19] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009.

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

[21] Xiuzhen Huang, Jing Lai, and Steven F. Jennings. Maximum common subgraph: some upper bound and lower bound results. *BMC Bioinformatics*, 7(S-4), 2006.

[22] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[23] Donald E. Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998.

[24] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1137–1145. Morgan Kaufmann, 1995.

[25] Lars Kotthoff. LLAMA: leveraging learning to automatically manage algorithms. Technical Report arXiv:1306.1031, arXiv, June 2013.

[26] Lars Kotthoff, Bernd Bischl, Barry Hurley, and Talal Rahwan. *Leveraging Learning to Automatically Manage Algorithms*. CRAN, December 2015.

[27] Lars Kotthoff, Pascal Kerschke, Holger Hoos, and Heike Trautmann. Improving the state of the art in inexact TSP solving using per-instance algorithm selection. In Clarisse Dhaenens, Laetitia Jourdan, and Marie-Eléonore Marmion, editors, *Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers*, volume 8994 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2015.

[28] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. Portfolios of subgraph isomorphism algorithms. In Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren, editors, *Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers*, volume 10079 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2016.

[29] Larry L. Havlicek and Nancy L. Peterson. Effect of the violation of assumptions upon significance levels of the pearson r. 84:373–377, 03 1977.

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

[31] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. A portfolio approach to algorithm selection. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, page 1542. Morgan Kaufmann, 2003.

[32] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.

[33] Andy Liaw and Matthew Wiener. *Breiman and Cutler's Random Forests for Classification and Regression*. CRAN, October 2015.

[34] Sharon L. Lohr. *Sampling: Design and Analysis*. Advanced (Cengage Learning). Cengage Learning, 2009.

[35] Helen M. Grindley, Peter J. Artymiuk, David W. Rice, and Peter Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. 229:707–21, 03 1993.

[36] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In Michel Rueher, 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, 2016.

[37] Ciaran McCreesh and Patrick Prosser. The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. *TOPC*, 2(1):8:1–8:27, 2015.

[38] Ciaran McCreesh, Patrick Prosser, and James Trimble. Heuristics and really hard instances for subgraph isomorphism problems. In Subbarao Kambhampati, 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, 2016.

[39] Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. In Carles Sierra, 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, 2017.

[40] Alexander M.F. Mood, Franklin A. Graybill, and Duane C. Boes. *Introduction to the Theory of Statistics*. International Student edition. McGraw-Hill, 1974.

[41] Samba Ndojh Ndiaye and Christine Solnon. CP models for maximum common subgraph problems. In Jimmy Ho-Man Lee, 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, 2011.

[42] Takao Nishizeki and Md. Saidur Rahman. *Planar Graph Drawing*, volume 12 of *Lecture Notes Series on Computing*. World Scientific, 2004.

[43] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[44] Panos M. Pardalos and Jue Xue. The maximum clique problem. *J. Global Optimization*, 4(3):301–328, 1994.

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

[46] David M. W. Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

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

[48] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[49] Lior Rokach and Oded Maimon. *Data Mining with Decision Trees - Theory and Applications. 2nd Edition*, volume 81 of *Series in Machine Perception and Artificial Intelligence*. WorldScientific, 2014.

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

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

[52] Christine Solnon, Guillaume Damiand, Colin de la Higuera, and Jean-Christophe Janodet. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognition*, 48(2):302–316, 2015.

[53] Robert R. Stoll. *Set Theory and Logic*. Dover books on advanced mathematics. Dover Publications, 1979.

[54] The R Core Team. *R: A Language and Environment for Statistical Computing: Reference Index*. R Foundation for Statistical Computing, 3.4.3 edition, November 2017.

[55] Roman Timofeev. Classification and regression trees (CART): Theory and applications. Master's thesis, CASE - Center of Applied Statistics and Economics, Humboldt University, Berlin, December 2004.

[56] Gabriel Valiente and Conrado Martnez. An algorithm for graph pattern-matching. In *In Proc. 4th South American Workshop on String Processing, volume 8 of Int. Informatics Series*, pages 180–197. Carleton University Press, 1997.

[57] Andrew R. Webb. *Statistical Pattern Recognition, 2nd Edition*. John Wiley & Sons, October 2002.

[58] Eric W. Weisstein. k-subset. *MathWorld – A Wolfram Web Resource*, April 2004.

[59] Soren H. Welling. Does Breiman's random forest use information gain or Gini index? Cross Validated, August 2015.

[60] Martin B. Wilk and Ramanathan Gnanadesikan. Probability plotting methods for the analysis of data. *Biometrika*, 55(1):1–17, 1968.

[61] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *CoRR*, abs/1111.2249, 2011.

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