Using Canonical Forms for Isomorphism Reduction in Graph-based Model Checking

Gijs Kant
kant@cs.utwente.nl

13 July 2010

Abstract

Graph isomorphism checking can be used in graph-based model checking to achieve symmetry reduction. Instead of one-to-one comparing the graph representations of states, canonical forms of state graphs can be computed. These canonical forms can be used to store and compare states. However, computing a canonical form for a graph is computationally expensive. Whether computing a canonical representation for states and reducing the state space is more efficient than using canonical hashcodes for states and comparing states one-to-one is not a priori clear. In this paper these approaches to isomorphism reduction are described and a preliminary comparison is presented for checking isomorphism of pairs of graphs. An existing algorithm that does not compute a canonical form performs better that tools that do for graphs that are used in graph-based model checking. Computing canonical forms seems to scale better for larger graphs.

Key words: Graph Isomorphism, Canonical Form, Graph-based Model Checking

Contents

1 Introduction

Formal methods in Software Engineering Software becomes larger and more complex. This makes also creating software more complex. In particular creating software that is correct, i.e., software that behaves according to its specifications, becomes harder as the size and complexity of systems increase. In software engineering several methods can be used to check whether a system behaves according to the specification, e.g., testing, simulation and formal verification. In formal verification of systems several methods can be used, such as formal proofs of correctness and model checking. Formal proofs of correctness can be very long and difficult to write and to understand. Often automatic reasoning is used to overcome part of this problem. Automatic theorem provers can be used to assist in proving correctness of a program. Another widely used technique is model checking. In model checking all possible states of the system and the transitions between the states are explored. For every

reachable state it is checked if certain formally defined safety properties hold. Also properties about the structure of the state space can be checked. The advantage of formal methods such as proofs and model checking over testing and simulation is that they can guarantee that every state of the system is considered (for as far as its behaviour is captured in the formal model), whereas testing and simulation are usually less complete. A major disadvantage of using automated reasoning (e.g., model checking) for validation is the conceptual gap between the informal understanding of what the system should do and the size of the formal proof or the state space that is required to check the correctness of the system. Therefore formal methods should always be used in companion with techniques for properly decomposing the system into understandable subsystems.

Model Checking In model checking (see e.g., [2]), systems are modelled as state transition systems, represented by Kripke structures, which consist of states, transitions between states, and a function that maps states to the propositions that hold in that state.

The states of the model that can be reached are defined by the possible (sequences of) transitions from a start state. Certain properties can be checked, such as safety properties (that some 'dangerous' state can never be reached) or liveness properties (that from each state eventually some 'progress' state can be or will be reached). Such properties are usually expressed in Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) formulae. For checking these properties all reachable states have to be considered. The result of model checking is the answer to the question if the specified system (or some state of that system) satisfies the specified properties. If some formula does not hold for the system, the model checker presents to the user a trace containing the transitions leading to an erroneous state.

In model checking we are faced with the problem of state space explosion, the problem that the number of states grows fast with the size of the system. Even for small systems the state space can be enormous. This limits the feasibility of model checking for large systems. Because of this problem a lot of effort has gone into reducing the state space. For this purpose often abstractions are used. It is, however, difficult to determine if the properties that hold in the more abstract model also hold in the modelled system. State space reduction can also be achieved by grouping similar states in the transition system, e.g., by bisimulation reduction, which preserves the behaviour of the original system.

Graph-based Model Checking In graph-based model checking, graphs are used for representing the states of a system. Graph transformation rules are used to define the transitions in the system. Properties of states are expressed as graph matchings. See [25, 10] for more on using graph rewriting systems for model checking.

A graph transition system is derived from a start graph and a set of graph transformation rules by recursively applying the rules to the set of states (initially only the start state) and adding the resulting states to the set until no new states are added. The set of all these derived state graphs form the set of reachable states and the names of the applied rules form the labels on the transitions in the transition system.

State space reduction by isomorphism checking Modelling the states as graphs enables to do on-the-fly state space reduction based on isomorphism of the states. States that are isomorphic are grouped, so that only one representative of the group is examined and stored instead of all individual states.

The idea behind the reduction is that the identities of the vertices in a state graph do not influence the behaviour of that state or the properties that hold in that state, because the matching of rules is not based on the identities of the vertices (this is exactly what makes graph matching a complex problem). So, two graphs being isomorphic implies that the same set of rules matches the graphs. Thus the resulting reduced transition system is bisimilar to the transition without reduction (this has been shown in [27]). From the bisimilarity it follows that the LTL and CTL properties that hold for the original system, also hold for the reduced system.

Experiments in [27] show that the number of states can be reduced dramatically for certain problems and that the time spent on isomorphism checking is not very much. This indicates that state space reduction by isomorphism checking is a promising technique for graph-based model checking.

We distinguish two types of isomorphism checking. The first is direct one-to-one isomorphism checking, i.e., algorithms that answer the question if two graphs are isomorphic with 'yes' or 'no'. The second type is based on the computation of canonical forms of the graphs. A canonical form of a graph G is a graph that is isomorphic to G, such that for a graph H, the canonical forms of G and H are equal if and only if G and H are isomorphic. By definition a set of isomorphic graphs has only one canonical form, which serves as unique representative of that set.

In model checking we do not want to check if two graphs are isomorphic, but we want to answer the question if a graph G is isomorphic to some graph in a set of graphs Q. With the first type of algorithms, $O(|Q|)$ isomorphism checks have to be done to answer that question. With the second type of isomorphism checking we can do better if we do not store the original graphs that we computed, but instead their canonical forms, i.e., Q is a set of canonical forms. Then we only have to compute the canonical form of G and check if that canonical form is in the set Q, for which we need $O(|Q|)$ equality checks. Testing for equality of two graphs can be done in $O(n+m)$ time for graphs with n vertices and m edges.

Algorithms for computing the canonical form of col-

oured graphs exist, such as NAUTY [17] and BLISS [12]. However, computing a canonical form for a graph is computationally complex. There is no algorithm known to solve the problem for arbitrary graphs within a polynomial time bound [13], [27]. Therefore, computing canonical forms is probably not the best method for comparing pairs of graphs, but it may improve performance in the case of comparing a graph to a large set of canonical forms of graphs.

In a model checking setting we usually do have large sets of states to consider. So to be faster in this setting, one-to-one isomorphism checking algorithms have to be a factor $|Q|$ faster for comparing a pair of graphs than a method based on canonical forms. Deciding if two graphs are isomorphic is a complex task, for which there is believed not to be a polynomial algorithm [31]. This means at least that determining which method is better in general is not trivial, but for large sets of graphs using canonical graphs may be a better choice.

GROOVE and hashing GROOVE [26] is a graph-based model checking tool. In GROOVE systems can be modelled by specifying states as edge labelled graphs and specifying graph transformation rules that describe the transitions between states. GROOVE has an editor for graphs and graph transformation rules. The tool can be used as a simulator and for model checking CTL and LTL formulas. It is implemented in Java.

Isomorphism reduction is done in GROOVE by a combination of using isomorphism checking for pairs of graphs and using an invariant hash function for reducing the number of graphs to be checked. States are stored in a hash map, using the invariant hash code as key for the set of graphs with that hash code. This way the number of state graphs that have to be checked for isomorphism is greatly reduced if the hash function is of high quality.

Problem statement Isomorphism checking is already being applied in model checking for symmetry reduction [27],[22],[28]. The goal of this paper is to compare several isomorphism reduction methods and find out which one offers the best performance in the case of graph-based model checking. In particular the performance of methods based on using canonical forms is compared to the existing algorithm in the tool GROOVE, which is a pairwise isomorphism checking algorithm that also uses hashing. Because of the complexity of both isomorphism checking and computing canonical forms it is not a priori clear which method leads to the best performance in isomorphism based symmetry reduction. This leads to the following questions.

1) Which isomorphism checking methods exist?

2) Does computing a canonical form of state graphs, instead of using pairwise isomorphism checking algorithms, improve the performance in graph-based model checking?

3) Does computing a canonical form of state graphs, instead of using hashing algorithms in combination with pairwise isomorphism checking algorithms, improve the performance in graph-based model checking?

This paper is written in the context of a research topics assignment as a preparation for a master's thesis. Therefore this is only a preliminary result. In this paper the theoretical background and related work are described, and preliminary experiments are presented with checking for isomorphism between pairs of graphs.

The paper is organised as follows. In the next section related work on isomorphism checking and isomorphism-based state space reduction is discussed. Section 3 lists definitions of concepts used throughout the paper. Section 4 treats the algorithms that we use for computing a canonical form of vertex coloured graphs. In Section 5 it is explained how the algorithms can be applied to edge labelled graphs by converting them to vertex coloured graphs. In Section 6 preliminary experimental results are presented that show that for most graphs used in graph-based model checking the existing algorithm in GROOVE, which does not compute canonical forms, performs better than tools that do compute a canonical form, but that canonical form computation scales better for larger graphs. In the final section conclusions and suggestions for future work are presented.

2 Related work

Complexity Subgraph matching is known to be a NP-complete problem (Problem GT48 in [9]). For isomorphism checking it is not known if the time complexity is in P or if the problem is NP-complete. It is believed not to be in P [18], [31].

One-to-one isomorphism checking Many algorithms exists that can check if two graphs are isomorphic. Ullmann [30] presented a search tree based algorithm for finding graph or subgraph isomorphisms between two graphs. Messner & Bunke [18, 19] made an optimised version for large graphs.

The graph matching algorithms by Cordella et al. [3], [4], [5] also aim at isomorphism checking for pairs of graphs. They use heuristics and efficient data structures that are optimed for matching large graphs.

Foggia, Sansone, and Vento compare four one-to-one isomorphism checking algorithms to NAUTY, a tool that computes canonical forms [8]. For many cases NAUTY performs comparable to these algorithms or better. For some cases NAUTY performs worse or is unable to find an answer, whereas some other algorithms are able to find an answer for all test cases.

Hsieh, Hsu & Hsu [11] do isomorphism checking for edge labelled graphs, but do not compute a canonical form. They compute vertex and graph signatures, which are used to partition the vertices. This partition is used

to limit the number of possible mappings between the vertices of the graphs that are compared. It is however unclear how the algorithm can be efficient for graphs for which the cells of the partition are large. It would make sense to update the vertex signatures based on the vertex signatures of neighbours, but such iterative partitioning is not done in the algorithm. The algorithm is only for undirected graphs, although it could be adapted to support directed graphs. No performance comparison with other algorithms has been given.

Invariant hash codes Rensink [27, 24] uses an isomorphism invariant hash code combined with one-to-one isomorphism checking for symmetry reduction in model checking. For state graphs an invariant hash code is computed. This hash code is used as key for the graph in the state store, which is implemented as a hash map. In order to check if a state has been visited before, the hash value is computed and only the graphs at the associated position in the hash map are compared to the newly encountered state. This reduces the number of graphs that have to be compared (by a factor that depends on the quality of the hash function). The hash code is based on a partition refinement algorithm where similar vertices are in the same cell. Each cell is distinguished by the number of incoming and outgoing edges and associated labels of vertices and those of the neighbouring vertices.

Canonical forms The NAUTY program by McKay [17] is able to produce a canonical form for directed coloured graphs, which can be used to test for isomorphism between graphs. The algorithm of McKay for computing the canonical form is described in [15] and will be explained in Section 4.

The complexity of the algorithm of McKay has been analysed by Miyazaki [20]. Miyazaki shows that NAUTY has an exponential worst case complexity. For some 3-regular graphs (for which canonical labellings can be computed in polynomial time, see [1]) NAUTY has an exponential lower bound. However, in practice the average computation time is much better.

Darga et al. [7] have optimised the NAUTY algorithm for symmetry detection in large and sparse graphs. The optimised algoritm is implemented in the tool SAUCY [6]. It does not, however, produce a canonical form of the graphs.

Optimisations of McKay's algorithm for large and sparse graphs have also been done by Junttila & Kaski [13] in the tool BLISS [12]. In experiments BLISS is shown to be faster than NAUTY for large and sparse graphs. It uses datatypes that allow more efficient storage and searching than the adjacency matrix that is used in NAUTY. Also other certificates for nodes in the search tree are used. Further, the heuristics for pruning certain subtrees of the search tree are optimised. The optimisations are further explained in Section 4 where the generation of the search tree is described. It would be interesting to see if e.g. BLISS has the same problems as NAUTY with computing canonical forms for the graphs used in [8].

Piperno proposed a new refinement algorithm with new graph invariants in order to reduce the search space [22]. The algorithm is implemented in the tool TRACES [23]. It uses multi-refinement, a combination of multiple refinement steps, as transitions in the search tree. Partitions of vertices are compared based on a 'quotient-graph', a graph where each vertex represents a cell of the partition and labelled edges represent the number of incident edges between vertices in the cells. However, TRACES only works for undirected graphs, which makes it unsuitable for our purposes.

The tool NAUTY has been used in model checking of systems specified in B by the tool ProB [29, 28]. The states of the B model are translated to edge labelled graph representations. These are again converted to a coloured graph representation and compared using NAUTY. In [29] a version of the NAUTY algorithm is used that is adapted to work for edge labelled graphs, but the search tree pruning optimisations of NAUTY are left out. In [28] on the contrary a conversion from edge labelled graphs to vertex coloured graphs is used in combination with the orignal NAUTY algorithm. In both approaches the states of the B model are converted to a graph representation and a canonical form for the state graph is computed in order to be able to store only the canonical forms. States in B models consist of sets containing abstract elements, nested sets and relations between elements. Transitions between states are inferred by operations on that data. Symmetries exist between the abstract elements of the sets. The elements do not have a concrete value, so if their relations to other elements are symmetric, they are interchangeable. Experimentation shows that the symmetry reduction results in faster model checking. However, this has only been tested with small numbers of vertices (< 100).

Subgraph matching algorithms There are algorithms that are efficient for generating all (frequent) subgraphs of a graph. The following two methods use canonical forms to distinguish the subgraphs and for efficient subgraph matching. They are tailored for undirected, connected graphs and they might not be very efficient for computing a canonical form for a graph, because they do not make use of the automorphisms in the graph. The complexity of both algorithms seems to be worse than that of NAUTY, so they are not interesting for us.

Kuramochi & Karypis [14] use a canonical labelling (for undirected graphs with edge labels and vertex labels). Canonical labellings are computed for all possible subgraphs in order to determine frequently occurring subgraphs in a large datasets of graphs. The algorithm can probably be used to compute canonical forms of graphs, but it is not tested for canonical labelling of graphs in general (whether it is faster than e.g. BLISS or NAUTY). It partitions vertices in a similar way as NAUTY

does and also uses iterative partitioning, but it does not prune the search tree by using automorphisms. It is probably easily adaptable to work for directed graphs.

Yan & Han [32] have a faster method for frequent subgraph discovery, which does also calculate canonical forms of subgraphs (called minimum DFS code, after the depth first search deployed in the algorithm). It builds a tree of codes for subgraphs, starting with all subgraphs consisting of one edge and iteratively adds edges to the subgraphs. For each subgraph a minimum code is computed, which is also used to prune parts of the tree.

3 Definitions

3.1 Graphs

We want to compare methods for isomorphism reduction for the tool GROOVE, where directed labelled graphs are used to represent states. In many tools (see e.g., Section 2), however,(directed) coloured graphs are used instead. Hence most algorithms in this paper will be presented in terms of coloured graphs. In this section we define both classes of graphs and isomorphism for these classes.

We assume a finite universe of labels Lab for edge labelled graphs and a finite universe of colours C. Throughout this paper it is assumed that Lab and C are fixed, totally ordered sets and that there is a hash function $hash : \mathsf{Lab} \cup \mathsf{C} \to \mathbb{N}$.

Definition 1 (Directed labelled graph)**.** A *directed labelled graph* G is a tuple $\langle V_G, E_G \rangle$ with a finite nonempty set of *nodes* (or *vertices*) V_G and a set of *edges* $E_G \subseteq V_G \times \mathsf{Lab} \times V_G$. The edges have associated source and target functions $src, tgt : E_G \to V_G$ and label function $lab : E_G \to \mathsf{Lab}$. The class of directed labelled graphs is denoted $\mathscr{G}_{\mathscr{L}}$.

Definition 2 (Isomorphism of directed labelled graphs)**.** Let $G = \langle V_G, E_G \rangle$ and $H = \langle V_H, E_H \rangle$ be two directed labelled graphs. A bijective function $f : V_G \to V_H$ is called an *isomorphism* if for all $v_1, v_2 \in V_G$ and $l \in \mathsf{Lab}$,

$$(v_1, l, v_2) \in E_G \iff (f(v_1), l, f(v_2)) \in E_H.$$

If such a function exists, G and H are called *isomorphic*, denoted $G \cong H$.

Definition 3 (Coloured graph)**.** A *directed graph* G is a tuple $\langle V_G, E_G \rangle$ with a finite nonempty set of *nodes* (or *vertices*) V_G and a set of *edges* $E_G \subseteq V_G \times V_G$. The edges have associated source and target functions $src, tgt : E_G \to V_G$. A directed graph G is called *coloured* if it has an associated function $c : V_G \to \mathsf{C}$. The class of coloured graphs is denoted $\mathscr{G}_{\mathscr{C}}$.

Definition 4 (Isomorphism of coloured graphs)**.** Let $G = \langle V_G, E_G, c_G \rangle$ and $H = \langle V_H, E_H, c_H \rangle$ be two coloured graphs. A bijective function $f : V_G \to V_H$ is called an *isomorphism* if for all $v \in V_G$ $c_G(v) = c_H(f(v))$ and for all $v_1, v_2 \in V_G$,

$$(v_1, v_2) \in E_G \iff (f(v_1), f(v_2)) \in E_H.$$

If such a function exists, G and H are called *isomorphic*, denoted $G \cong H$.

3.2 Transition systems

In model checking the behaviour of systems is modelled by transition systems.

Definition 5 (Transition system)**.** A *transition system* is a tuple $K = \langle Q, T, q_0 \rangle$, with a set of states Q, a set of transitions $T \subseteq Q \times Q$, and an initial state $q_0 \in Q$.

From a transition system a *Kripke structure* can be constructed, where each state is associated with a set of atomic propositions that hold in that state. In graph-based systems, usually the labels of self-edges in the transition system are used, i.e., the matching rules that do not change the state graph. These can be derived from the state graphs, so in graph transition systems the atomic properties are already present implicitly if the state graphs are stored.

In this paper we will consider graph transition systems defined as follows.

Definition 6 (Graph transition system)**.** A *graph transition system* is a transition system $K_G = \langle Q, T, q_0 \rangle$ where each state $q \in Q$ is a graph.

We write $\mathscr{G}$ for a class of graphs, e.g., directed labelled graphs, and $\mathscr{R}$ for the class of graph transformation rules that can be applied to graphs in $\mathscr{G}$.

In the following we assume the existence of a successor function that computes the set of successor state graphs for a given state graph for some state, based on a set of graph transformation rules:

Definition 7 (Successor function)**.** The successor function $succ : \mathscr{G} \times \mathscr{P}(\mathscr{R}) \to \mathscr{P}(\mathscr{G})$ computes the set of successor state graphs $succ(g, R)$ for a given state graph $g \in \mathscr{G}$, based on a set of graph transformation rules $R \subseteq \mathscr{R}$.

In Alg. 1 it is shown how a graph transition system can be derived from a start state and a set of rules. For a set of state graphs S (initially containing only the initial state) the successor states are considered. If a successor state has been visited before, only a transition to the state is added (line 9), otherwise a new state is added to the set of states Q and to S and a transition to this new state is added to the set of transitions (lines 10–13).

This algorithm produces a *graph transitions system modulo isomorphism*, i.e., isomorphic graphs are considered to represent the same state. This *isomorphism reduction* is achieved by checking for isomorphism instead of checking for equality in line 8. The reduced transitions

Algorithm 1 Compute reduced graph-based transition system for $q_0 \in \mathscr{G}$ and a set of rules $R \subseteq \mathscr{R}$.

```
1:  Q := {q_0}
2:  T := ∅
3:  S := {q_0}
4:  while S ≠ ∅ do
5:      Let q be some element of S
6:      S := S \ q
7:      for all s ∈ succ(q, R) do
8:          if ∃p ∈ Q such that s ≅ p then
9:              T := T ∪ {(q, p)}
10:         else
11:             Q := Q ∪ {s}
12:             T := T ∪ {(q, s)}
13:             S := S ∪ {s}
14:         end if
15:     end for
16: end while
17: return ⟨Q, T, q_0⟩;
```

system is *bisimilar* to the transition without isomorphism reduction (see [27]). Bisimulation equivalence implies that Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) formulae that hold in one transition system also hold in an equivalent system (see, e.g., [2] on bisimulation equivalence).

Instead of checking for isomorphism for each visited state separately, we would like to use canonical forms of states and store those. For this we need a canonical representation function that is defined as follows:

Definition 8 (Canonical representation and canonical form)**.** A *canonical representation* function $can : \mathscr{G} \to \mathscr{G}$ computes an isomorphism invariant graph representative for each graph, such that for every pair of graphs $G, H \in \mathscr{G}$, $can(G) = can(H)$ if and only if $G \cong H$. $can(G)$ is called the *canonical form* of G.

In Alg. 2 it is shown how such a canonical representation function can be used in generating a reduced transition system. In line 9 the canonical form is computed. In the next line an equal graph is looked up, instead of an isomorphic graph as in Alg. 1. The use of canonical forms may result in a system with different (but isomorphic) states than the system that is generated by Alg. 1. However, the two systems are isomorphic and the states that are mapped to each other by the isomorphism are isomorphic states. Isomorphism of transitions systems is an even stronger equivalence relation than bisimulation. Hence, the properties that hold for one system, also hold for an isomorphic one.

For optimisation reasons, hash values can be used as keys for storing the state graph in a hash set.

Definition 9 (Invariant hash function)**.** An *invariant hash function* is a function $hash : \mathscr{G} \to \mathbb{N}$ that associates an integer value, called *hash code*, with each graph such that for every pair of graphs $G, H \in \mathscr{G}$, $hash(G) = hash(H)$ if $G \cong H$.

Hash values can also be used in *bitstate hashing*, i.e., storing a set of hash values of visited states instead of

Algorithm 2 Compute the reduced graph-based transition system for $q_0 \in \mathscr{G}$ and a set of rules $R \subseteq \mathscr{R}$ using a canonical representation function *can*.

```
1:  r_0 := can(q_0)
2:  Q := {r_0}
3:  T := ∅
4:  S := {r_0}
5:  while S ≠ ∅ do
6:      Let q be some element of S
7:      S := S \ q
8:      for all s ∈ succ(q, R) do
9:          r := can(s)
10:         if r ∉ Q then
11:             Q := Q ∪ {r}
12:             S := S ∪ {r}
13:         end if
14:         T := T ∪ {(q, r)}
15:     end for
16: end while
17: return ⟨Q, T, q_0⟩;
```

Algorithm 3 Generate the (possibly incomplete) state space reachable from $q_0 \in \mathscr{G}$ and a set of rules $R \subseteq \mathscr{R}$ using an invariant hash function *hash*. Analysis of the state space is done on-the-fly.

```
1:  X := {hash(q_0)}
2:  S := {q_0}
3:  while S ≠ ∅ do
4:      Let q be some element of S
5:      S := S \ q
6:      for all s ∈ succ(q, R) do
7:          x := hash(s)
8:          if x ∉ X then
9:              X := X ∪ {x}
10:             S := S ∪ {s}
11:             Analyse s
12:         end if
13:     end for
14: end while
```

the complete states. This results in an incomplete state space, because of possible collisions of hash values, i.e., different states can have the same hash value. Alg. 3 shows how the (possibly incomplete) set of hash values representing reachable states can be computed. In line 7 the hash code is computed. In the following lines the hash code and the state graph are stored if the hash code was not yet in the set of 'visited' hash codes, otherwise that state graph is ignored. The algorithm can be used to approximate the set of reachable states and has a very low memory footprint. No states are generated that are not in the original transition system, but there is no guarantee that all states of the orginal system are reached. This can, however, be useful as an initial search for invalid states in a large state space. Because the state space is not guaranteed to be complete and different states can be merged into the same representation, temporal logic formulae can not be verified in the case of bitstate hashing. Instead, atomic properties can be checked on-the-fly (line 11).

3.3 Partitions and permutations

The canonical representation functions in NAUTY and BLISS are based on relabelling the vertices in the graph. Relabelling the vertices of the graph actually is performing a permutation on the vertex identities.

Definition 10 (Permutation)**.** A *permutation* of a set A is a bijective function $\alpha : A \to A$. The image of $a \in A$ under a permutation α is denoted $\alpha(a)$ or a^α. The set of all permutations for a set $\{1, 2, \ldots, n\}$ is denoted S_n.

A permutation can be represented as a matrix:

$$\alpha = \begin{bmatrix} 1 & 2 & \cdots & n \\ 1^\alpha & 2^\alpha & \cdots & n^\alpha \end{bmatrix} \in S_n.$$

Definition 11 (Graph permutation)**.** A *graph permutation* is a vertex permutation $\gamma : V \to V$ that associates with each directed coloured graph $G = \langle V, E, c\rangle$ a permuted graph $G^\gamma = \langle V^\gamma, E^\gamma, c^\gamma\rangle$, where

$$\begin{aligned} V^\gamma &= \{v^\gamma \mid v \in V\} = V, \\ E^\gamma &= \{(v_1^\gamma, v_2^\gamma) \mid (v_1, v_2) \in E\} \text{ and} \\ c^\gamma &= \{(v^\gamma, k) \mid (v, k) \in c\}. \end{aligned}$$

The set of all graph permutations for a set of vertices V is denoted S_V.

For all permutations $\gamma \in S_V$ it holds that $G^\gamma \cong G$. A special subset of S_V is the set of *automorphisms* of G, $\mathrm{Aut}(G) = \{\gamma \in S_V \mid G^\gamma = G\}$.

An important ingredient of the algorithm that will be described in the next section is partition refinement. Vertices of the graph are partitioned in equivalence classes. The initial partition of the vertices is based on the colours of the vertices. Then the partition is refined such that also the number of incoming and outgoing edges from the vertices is taken into account.

Definition 12 (Partition)**.** A *partition* π of a set of nodes V is a set $\{W_1, W_2, \ldots, W_r\}$ of nonempty disjoint *cells* $W_i \subseteq V$ whose union is V. A partition with only trivial cells, i.e., cells that contain only one element, is called a *discrete partition*. The partition that contains only one cell, the set V, is called the *unit partition*. The set of partitions of V is denoted $\Pi(V)$. An *ordered partition* of V is a sequence $(W_1, W_2, \ldots, W_r)$ such that the set $\{W_1, W_2, \ldots, W_r\}$ is a partition of V. The set of ordered partitions of V is denoted $\underset{\sim}{\Pi}(V)$.

The set of automorphisms for a graph G with vertex partition π is defined as $\mathrm{Aut}(G, \pi) = \{\gamma \in S_V \mid G^\gamma = G \wedge \pi^\gamma = \gamma\}$.

In the following we denote the vertices as natural numbers, i.e., the set of vertices V is the set of numbers $\{1, 2, \ldots, n\} \subseteq \mathbb{N}$ with $n = |V|$.

Definition 13 (Partition permutation)**.** If $\pi \in \underset{\sim}{\Pi}(V)$ is a discrete ordered partition, we define the *permuted graph* $G(\pi)$, isomorphic to G, by relabelling the vertices of G in the order that they appear in π: given $\pi = (\{i_1\}, \{i_2\}, \ldots, \{i_n\})$ with $\{i_1, i_2, \ldots, i_n\} = \{1, 2, \ldots, n\}$, the permuted graph, denoted $G(\pi)$, is defined as $(G)^\delta$, where the permutation δ is given by

$$\delta = \begin{bmatrix} i_1 & i_2 & \cdots & i_n \\ 1 & 2 & \cdots & n \end{bmatrix} \in S_V.$$

This permutation δ, associated with partition π, is also written as $\overline{\pi}$. This partition permutation provides a relabelling of vertices based on a generated partition of vertices.

Example 1. As an example, suppose an isomorphism $\gamma = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1\}$ that maps vertices of graph G to vertices of H, with

$$\begin{aligned} V_G &= V_H = \{1, 2, 3\}, \\ E_G &= \{(1, a, 2), (2, b, 3), (3, c, 1)\}, \text{ and} \\ E_H &= \{(2, a, 3), (3, b, 1), (1, c, 2)\}. \end{aligned}$$

If we use some ordered partition $\pi = (\{i_1\}, \{i_2\}, \{i_3\})$ as a permutation of G, then

$$(G)^{\overline{\pi}} = (H)^{\overline{\pi}^\gamma}.$$

For instance, let $\pi = (\{2\}, \{1\}, \{3\})$. Then

$$\begin{aligned} (E_G)^{\overline{\pi}} &= \{(2, a, 1), (1, b, 3), (3, c, 2)\}, \\ \overline{\pi}^\gamma &= (\{3\}, \{2\}, \{1\}), \\ (E_H)^{\overline{\pi}^\gamma} &= \{(2, a, 1), (1, b, 3), (3, c, 2)\} = (E_G)^{\overline{\pi}}. \end{aligned}$$

Definition 14 (Partition refinement)**.** Given partitions π_1, π_2 of some set, π_1 is called a *refinement* of π_2 or *finer* than π_2 (and π_2 is called *coarser* than π_1), denoted $\pi_1 \sqsubseteq \pi_2$, if for all cells $V_i \in \pi_1$ there exists a cell $W_j \in \pi_2$ such that $V_i \subseteq W_j$.

The partition refinement algorithm used in computing the canonical form, to be described in Section 4 computes the coarsest stable refinement of a partition. Stability of a partition is based on the numbers of adjacent elements of the members of the cells of the partition.

Definition 15 (Number of adjacent elements)**.** Given a directed graph $G = \langle V, E\rangle$ and a partition $\pi \in \Pi(V)$, for an element $v \in V$ and a cell $W \in \pi$, the number of elements of W which are adjacent in G to v is defined as:

$$d(v, W) = |\{w \in W \mid (v, w) \in E \vee (w, v) \in E\}| \qquad (1)$$

This definition considers edges in both directions. This differs from [17] where only one direction is used, which is related to the data structure used in NAUTY, which allows for easy comparison of rows of the matrix, whereas comparing columns is more expensive. In the case of undirected graphs this does not make a difference, but for directed graphs it does.

Definition 16 (Stable partition)**.** A partition π is called *stable* for a directed graph G if for every pair of cells $W_i, W_j \in \pi$ the number of adjacent elements in W_j is the same for each element in W_i, i.e., for all vertices $v_1, v_2 \in W_i$ it holds that $d(v_1, W_j) = d(v_2, W_j)$. The set of all stable partitions of a set V is denoted $\Pi_S(V)$.

The stable partition resulting from the partition refinement algorithm is not necessarily a discrete partition, so the result of partition refinement can not immediately be used as permutation of the vertices. Each discrete partition is also stable, but for n vertices there are $n!$ possible permutations and we want to find a unique partition that gives us a canonical relabelling. Therefore we need a search tree that we can search for candidate canonical permutations and we need a way to order the candidate permutations so that we can choose one. The generation of this search tree and the ordering of the permutations are discussed in Section 4.3 and 4.2.

The partition refinement algorithm consists of iteratively splitting cells of the partition based on the number of adjacent elements of members of cells, until the partition is stable. The splitting of the cells of a partition π is defined with respect to a set S (usually some cell of the partition) and denoted $split(\pi, S)$.

Definition 17 (Split)**.** For a partition $\pi \in \Pi(V)$ and a set $S \subseteq V$, a partition $\pi' = split(\pi, S)$ is a refinement of π for which for all $W \in \pi$, for all $W_i, W_j \subseteq W$ with $W_i, W_j \in \pi'$, it holds that

$$\forall v_1 \in W_i, v_2 \in W_j \cdot \big(d(v_1, S) = d(v_2, S) \iff W_i = W_j\big).$$

If $split(\pi, S) \neq \pi$, S is called a *splitter* of π.

4 Computing a canonical form for (directed) coloured graphs

In this section it is explained how a canonical form of a directed coloured graph can be computed. McKay published an algorithm for finding a unique vertex labelling for isomorphic graphs [15], which is implemented in the tool NAUTY [17]. Improvements have been done by Juntila & Kaski in the tool BLISS; the algorithm they describe is used in the remainder of this paper.

The idea is to generate for each graph a set of discrete partitions that can be used as permutation of the vertices of the graph, which results in a relabelled graph. If we have an ordering of the graphs, and if for isomorphic graphs the same set of relabelled graphs is generated, we can choose the minimum or maximum of the set as canonical form.

An easy but inefficient way of generating this set of graphs is generating all possible permutations of the vertices, which results in $|V|!$ permutations of the set of vertices V. An ordering of the graphs can be obtained by representing each graph by a string that is a concatenation of the vertex colours and of the rows of the adjacency matrix (which represents the incident edges in the graph), and use an ordering on the strings.

The tools NAUTY and BLISS use far more efficienct algorithms that do not generate all possible vertex permutations, but still result in an equal set of permuted graphs for isomorphic graphs. The algorithms mainly consist of the following two ingredients:

1) A partition refinement algorithm that computes the unique coarsest stable partition for a given graph and initial partition of vertices;

2) An algorithm that generates a search tree of stable partitions with discrete partitions as leaf nodes, of which one is chosen as the relabelling partition permutation leading to the canonical form.

The search tree is generated by first computing a stable partition (which is the root node of the tree) and then splitting one of the cells. For each of the members of the cell a subtree is added, where that member is put in a separate cell. Then each of the resulting partitions is stabilised again. This continues until all branches end in discrete partitions (the leaf nodes).

Because every intermediate partition is stabilised before it is split again, the number of nodes in the tree is reduced. The properties that are used in the partition refinement are isomorphism invariant, so the resulting set of permuted graphs stays equal for isomorphic graphs. This is required in order to be able to compute the canonical form.

In the next section the partition refinement algorithm is explained. In Sections 4.3 and 4.4 the generation and pruning of the search tree are described. An ordering of coloured graphs is given in Section 4.2.

4.1 Partition refinement algorithm

The vertices of a graph are partitioned into cells of vertices that are similar. Initially this partition is based on vertex colours, but this partition is refined based on the number of neighbours of vertices in other cells. The partition refinement algorithm and the result it produces are described in this section.

The algorithm computes the unique coarsest stable refinement of a partition. The stability of partitions is defined in terms of numbers of adjacent vertices in the cells of the partition. A partition is stable if for each pair of elements of a cell the number of adjacent vertices is equal for both elements in all of the cells of the partitions (see Definition 16).

Suppose we have two isomorphic graphs $G \cong H$, with $G^\alpha = H$. Then if π_1 is a partition of vertices in G and $\pi_2 = \pi_1^\alpha$ is a partition of vertices in H, equivalent to π_1, in the sense that $(G)^{\pi_1} = (H)^{\pi_2}$ (see Example 1). Then also the unique coarsest stable refinements of π_1 and π_2 are equivalent. This is the case, because stability of partitions is defined such that it is isomorphism invariant, i.e., does not depend on the particular identities

of vertices. It follows then from the uniqueness of the coarsest stable partition refinement and the isomorphism between G and H that the resulting stable partitions are equivalent (in the same sense of equivalence and by the same isomorphism α).

Unique coarsest stable refinement Here we prove that a unique coarsest stable refinement exists for each partition $\pi \in \Pi(V)$ of vertices V for a graph G. We assume that the set V is finite and hence also $\Pi(V)$, the set of all partitions of V, is finite. We start with proving that the set of all partitions of a set forms a lattice (both a least upper bound and a greatest lower bound exists for each set of partitions). Then we prove that the least upper bound preserves stability. From the fact that each discrete partition is stable we can conlude that for each partition there exists a stable refinement (i.e. the discrete partition). It then follows that for each partition there exists a unique coarsest stable refinement.

Definition 18 (Least upper bound of partitions)**.** For a set of partitions $\Pi \subseteq \Pi(V)$ and an ordering relation $\sqsubseteq$, an *upper bound* is an element $\pi \in \Pi(V)$ such that for all elements $\rho \in \Pi$, $\rho \sqsubseteq \pi$. The *least upper bound* of Π, denoted $\operatorname{lub}\Pi$, is the upper bound π such that for all other upper bounds ρ, $\pi \sqsubseteq \rho$.

The least upper bound of a pair of elements $\pi_1, \pi_2 \in \Pi(V)$ is also called *join*, denoted $\pi_1 \sqcup \pi_2$. For computing this least upper bound we need the following relation.

Every partition $\pi \in \Pi(V)$ can be considered as a binary equivalence relation where each pair reflects that two elements are in the same cell:

$$R = \{(s,t) \in V \times V \mid \exists W \in \pi \cdot s, t \in W\} \qquad (2)$$

An example of partitions represented by binary relations is shown in Figure 1. The other way around, a partition can be derived from a binary relation $R \subseteq V \times V$. The relation can be seen as a graph (an edge between two elements representing that a relation between the elements exists). By taking the maximal connected subgraphs (or components) and regarding the vertices in those subgraphs as the elements of a cell (so, one cell per subgraph) we have a partition of the elements.

Proposition 4.1. *Given a set of partitions* $\Pi = \{\pi_1, \pi_2, \ldots, \pi_r\} \subseteq \Pi(V)$ *and their associated binary relations* $R_1, R_2, \ldots, R_r$*, the partition* π' *formed by the sets of vertices of maximal connected subgraphs of the union* $R_1 \cup R_2 \cup \cdots \cup R_r$ *is the least upper bound of* Π.

Now we show that the least upper bound of two stable partitions is itself stable as well.

Theorem 4.2. *Given two stable partitions* $\pi_1, \pi_2 \in \Pi_S(V)$*, the least upper bound* $\operatorname{lub}\{\pi_1, \pi_2\}$ *is also stable.*

Proof. To prove: for all $\pi_1, \pi_2 \in \Pi_S$, $\pi_1 \sqcup \pi_2 \in \Pi_S$.

1) First we observe that because π_1 and π_2 are refinements of their least upper bound, the cells of $\pi_1 \sqcup \pi_2$ are unions of cells in π_1 and unions of cells in π_2.
2) Because the way the least upper bound is constructed there exists for each pair $v, w \in W$, $W \in \pi_1 \sqcup \pi_2$ a path
$$v = v_1, v_2, \ldots, v_r = w$$
such that for all pairs $v_i, v_{i+1} (1 \leq i < r)$, $\exists W' \in (\pi_1 \cup \pi_2) \cdot v_i, v_{i+1} \in W'$.
3) Because π_1 and π_2 are stable, each pair v_i, v_{i+1} has the same number of neighbouring elements for all cells of either π_1 or π_2, so certainly for unions of cells in π_1 or of cells in π_2.
4) Hence, by induction on r, the same holds for the pair v, w. So, $\pi_1 \sqcup \pi_2$ must also be stable. □

Definition 19 (Greatest lower bound of partitions)**.** Given a set of partitions $\Pi \subseteq \Pi(V)$ and an ordering relation $\sqsubseteq$, a *lower bound* is an element $\pi \in \Pi(V)$ such that for all elements $\rho \in \Pi$, $\pi \sqsubseteq \rho$. The *greatest lower bound* of Π, denoted $\operatorname{glb}\Pi$, is the upper bound π such that for all other lower bounds ρ, $\rho \sqsubseteq \pi$.

The greatest lower bound of a pair of elements $\pi_1, \pi_2 \in \Pi(V)$ is also called *meet*, denoted $\pi_1 \sqcap \pi_2$.

Proposition 4.3. *Given a set of stable partitions* $\Pi = \{\pi_1, \pi_2, \ldots, \pi_r\} \subseteq \Pi_S(V)$*, there exists a stable greatest lower bound* $\operatorname{glb}_S\{\pi_1, \pi_2, \ldots, \pi_r\} \in \Pi_S(V)$.

Proof. Let L be the set of stable lower bounds of Π:

$$L = \{\pi \in \Pi_S(V) \mid \forall \pi' \in \Pi \cdot \pi \sqsubseteq \pi'\}.$$

The least upper bound of this set of lower bounds, $\operatorname{lub} L$, is the stable greatest lower bound of Π, because

1) the least upper bound of a set of stable partitions is itself stable (Theorem 4.2);
2) $\operatorname{lub} L$ is a lower bound of Π, i.e., $\operatorname{lub} L \in L$;
3) $\operatorname{lub} L$ is an upper bound of the set L.

Hence, a stable greatest lower bound exists. □

Proposition 4.4. *Given a set of vertices* V*, the set of stable partitions* $\Pi_S(V)$ *forms a lattice under the refinement relation* $\sqsubseteq$*,* $\langle \Pi_S(V), \sqsubseteq \rangle$.

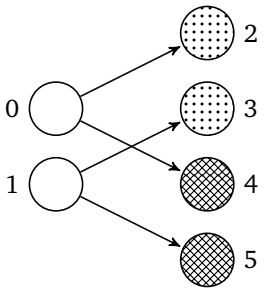
Proof. Both least upper bounds and greatest lower bounds exist, see Prop. 4.1 and 4.3 repectively. □

The existance of a greatest lower bound enables us to conclude the following.

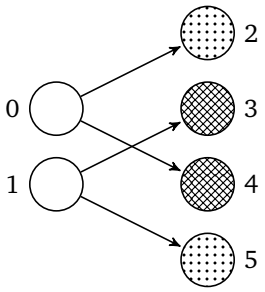
Theorem 4.5. *For a directed graph* G *and initial partition* $\pi \in \Pi(V)$*, there is a* unique coarsest stable refinement*, i.e., a stable partition* $\pi' \sqsubseteq \pi$*, such that for all other stable partitions* $\rho \sqsubseteq \pi$ *it holds that* $\rho \sqsubseteq \pi'$.

Proof. Two parts:

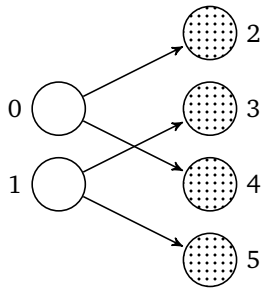
1) The discrete partition is stable, so there is always a stable partitions that is a refinement of π.
2) Of the refinements of π there is one which is the coarsest, this is the greatest lower bound of π in $\Pi_S(V)$, given by Prop. 4.3. □



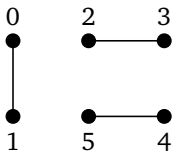
(a) Partition π_1.



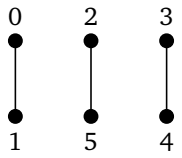
(b) Partition π_2.



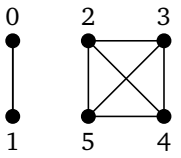
(c) The least upper bound of π_1 and π_2, $\pi_1 \sqcup \pi_2$.



(d) Relation R_1, reflecting the partition $\{\{0,1\},\{2,3\},\{4,5\}\}$.



(e) Relation R_2, reflecting the partition $\{\{0,1\},\{2,5\},\{3,4\}\}$.



(f) The transitive closure of $R_1 \cup R_2$, reflecting the partition $\{\{0,1\},\{2,3,4,5\}\}$.

Figure 1: The partitions π_1 and π_2, which are stable, and their least upper bound $\pi_1 \sqcup \pi_2$. The partitions are shown by the colours of the vertices. The partitions can be seen as relations, where elements in the same cell are related. Relation R_1 reflects partitions π_1, R_2 reflects partition π_2, and the transitive closure of R_1 and R_2 reflects the least upper bound $\pi_1 \sqcup \pi_2$.

McKay's partition refinement algorithm This unique coarsest stable partition can be computed by applying the partition refinement algorithm presented by McKay, which is shown in Algorithm 4. An example of the partition refinement is given in Figure 2. The algorithm iterates over the sequence of potential splitters $W \in \alpha$. It searches for cells V for which W is a splitter, i.e., there exists $v_1, v_2 \in V$ for which the number of adjacent elements in W is not equal: $d(v_1, W) \neq d(v_2, W)$. In line 12 the cell V is split into cells X_i, which are ordered by the number of adjacent elements in W. This can be easily done by building an ordered map where each element $v \in V$ is added to the entry with $d(v, W)$ as key. V is replaced by one of the largest cells, the others are added (with their ordering maintained) to the sequence of potential splitters α. An example of this step is shown in Figure 3.

Paige & Tarjan [21] published a similar algorithm, with some small differences:

1) McKay uses ordered partitions, i.e., sequences of disjoint cells that form a partition, and Paige & Tarjan use sets of cells for partitions;

2) After splitting a cell into subcells, the algorithm of Paige & Tarjan leafs out the largest subcell when adding subcells to the sequence of splitters. The algorithm of McKay instead replaces the original cell in the sequence of splitters by the largest subcell (if the original cell is still in the queue of splitters, otherwise the largest subcell is left out).

In effect, both algorithms implement a variant of the "process the smaller half" strategy of Hopcroft. Because of this the time complexity of the algorithms is $O(|E| \cdot \log|V|)$ (see [21]);

Proposition 4.6. *Given a graph G and a partition π of the vertices of G, refine(G, π, π) (Alg. 4) yields the coarsest stable partition of π for G.*

Proof. This has been proved in [15]. □

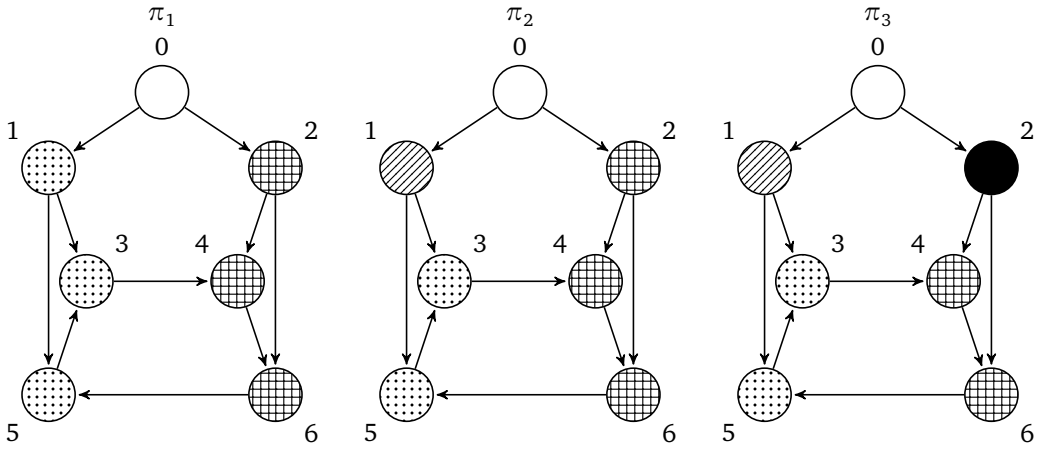
4.2 A total ordering on coloured graphs

To determine the minimum of a set of coloured graphs we need a total ordering on $\mathcal{G}_{\mathscr{C}}$. In this section we define an ordering based on the number of vertices, number of edges, the colours of the vertices, and the adjacency matrix, which represents the incident edges. We denote the vertices as natural numbers, i.e., the set of vertices V is the set of numbers $\{1, 2, \ldots, n\} \subseteq \mathbb{N}$ with $n = |V|$, and use the natural ordering of $\mathbb{N}$ as ordering of the vertices. Now we define the adjacency matrix for coloured directed graphs.

Definition 20 (Adjacency matrix)**.** For a coloured directed graph $G = \langle V, E, c\rangle$, the *adjacency matrix* $A(G)$ is a $n \times n$ matrix ($n = |V|$) with for all $i, j \in [1..n]$,

$$A(G)_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

The ordering of adjacency matrices is based on concatenating the rows of the matrix $(A(G)_i)$, for $1 \leq i \leq n$)



(a) The initial partitions with cells $\{0\}$, $\{1,3,5\}$, and $\{2,4,6\}$.

(b) First the first cell is used as a splitter ($m = 1$, $W = \{v_0\}$). The second cell ($k = 2$, $V_k = \{1,3,5\}$) is split into two cells, $\{3,5\}$ and $\{1\}$ (in that order), because v_1 has one incoming edge from 0 and the vertices 3 and 5 have no edges from or to 0.

(c) The cell $V_k = \{2,4,6\}$ ($k = 4$) is also split into two cells, $\{4,6\}$ and $\{2\}$ (in that order), because 2 has one incoming edge from 0 and the vertices 4 and 6 have no edges from or to 0. The resulting partition is stable.

Figure 2: An example of partition refinement by Algorithm 4. Graph (a) shows the initial partition of the vertices. In (b) and (c) the result of subsequent splitting of cells is shown. The split steps are explained in Figure 3.

$$
\begin{aligned}
\alpha' &= (\overbrace{\{0\}}^{W}, \{1,3,5\}, \{2,4,6\}) \\
\pi' &= (\{0\}, \underbrace{\{1,3,5\}}_{V_k}, \{2,4,6\}) \\
&\downarrow \\
X_1 &= \{3,5\}, \quad \forall v \in X_1 \cdot d(v,W) = 0 \\
X_2 &= \{1\}, \quad \forall v \in X_2 \cdot d(v,W) = 1 \\
&\downarrow \\
\alpha' &= (\overbrace{\{0\}}^{W}, \overbrace{\{3,5\}}^{X_1}, \{2,4,6\}, \overbrace{\{1\}}^{X_2}) \\
\pi' &= (\{0\}, \underbrace{\{3,5\}}_{X_1}, \underbrace{\{1\}}_{X_2}, \{2,4,6\})
\end{aligned}
$$

(a) First step.

$$
\begin{aligned}
\alpha' &= (\overbrace{\{0\}}^{W}, \{3,5\}, \{2,4,6\}, \{1\}) \\
\pi' &= (\{0\}, \{3,5\}, \{1\}, \underbrace{\{2,4,6\}}_{V_k}) \\
&\downarrow \\
X_1 &= \{4,6\}, \quad \forall v \in X_1 \cdot d(v,W) = 0 \\
X_2 &= \{2\}, \quad \forall v \in X_2 \cdot d(v,W) = 1 \\
&\downarrow \\
\alpha' &= (\overbrace{\{0\}}^{W}, \{3,5\}, \overbrace{\{4,6\}}^{X_1}, \{1\}, \overbrace{\{2\}}^{X_2}) \\
\pi' &= (\{0\}, \{3,5\}, \{1\}, \underbrace{\{4,6\}}_{X_1}, \underbrace{\{2\}}_{X_2})
\end{aligned}
$$

(b) Second step.

Figure 3: The split steps done by Algorithm 4 for the coloured graph in Figure 2. The step with $V_k = \{0\}$ is not shown, because a singleton cell cannot be split (the resulting partition is $\pi_k = (X_1)$, with $X_1 = V_k = \{0\}$). (a) shows the splitting that corresponds to the transition from π_1 to π_2. (b) shows the splitting that corresponds to the transition from the π_2 to π_3, which is a stable partition.

Algorithm 4 Compute the refinement $\mathit{refine}(G, \pi, \alpha)$ of an ordered partition $\pi \in \underset{\sim}{\prod}(V)$ for a directed graph $G = \langle V, E\rangle$, given α, a sequence of cells in π that are used as splitters. $\mathit{refine}(G, \pi, \pi)$ computes the coarsest stable partition of π for G.

1: $\pi' := \pi$
2: Let α' be a queue, initialised with the elements of α
3: **while** α' is not empty **do**
4: {Suppose $\alpha' = (W_1, W_2, \ldots, W_q)$ at this point.}
5: **if** π' is discrete **then**
6: **return** π';
7: **end if**
8: $W := W_1$
9: Remove W_1 from α'
10: {Suppose $\pi' = (V_1, V_2, \ldots, V_r)$ at this point.}
11: **for** $k := 1$; $k \le r$; $k{+}{+}$ **do**
12: Define $\pi_k = (X_1, X_2, \ldots, X_s) \in \underset{\sim}{\prod}(V_k)$ such that for all $v_1 \in X_i, v_2 \in X_j$, we have $d(v_1, W) < d(v_2, W) \iff i < j$.
13: **if** $s > 1$ **then**
14: $t := \min\{i \mid 1 \le i \le s \wedge |X_i| = \max\{|X_j| \mid X_j \in \pi_k\}\}$ {the smallest integer t such that $|X_t|$ is maximum (with $1 \le t \le s$)}
15: **if** $\exists j$ such that $W_j = V_k$ (with $1 \le j \le q$) **then**
16: $W_j := X_t$ {Replace W_j in α' by X_t, the largest subcell of W_j}
17: **end if**
18: **for** $1 \le i < t$ and $t < i \le s$ **do**
19: Add X_i to the end of α'
20: **end for**
21: Update π' by replacing the cell V_k with the cells $X_1, X_2, \ldots, X_s$ in that order (*in situ*).
22: **end if**
23: **end for**
24: **end while**
25: **return** π';

which results in a binary number. As ordering of these number the usual natural ordering of numbers is used.

The colours of the vertices are compared as follows. For coloured graphs with n vertices, the colours can be represented as a sequence $(c(1), c(2), \ldots, c(n))$. For comparing sequences of n colours we use lexicographical ordering, i.e., first the first elements are compared, if these are equal the second elements are compared, etc., until a difference in colour is found or all elements have been compared. In this section $c_{G,i}$ denotes the i-th elements of this sequence for G: $c_{G,i} = c_G(i)$ for $i \in V_G$. For two graphs $G = \langle V_G, E_G, c_G\rangle$ and $H = \langle V_H, E_H, c_H\rangle$ with $V_G = V_H = \{1, 2, \ldots, n\}$,

$$(c_{G,1}, c_{G,2}, \ldots, c_{G,n}) < (c_{H,1}, c_{H,2}, \ldots, c_{H,n}),$$
$$\text{if } c_{G,i} < c_{H,i} \text{ for the smallest } i \in [1..n]$$
$$\text{for which } c_{G,i} \neq c_{H,i}.$$

For coloured directed graphs we define an order relation $\le$ as follows.

Definition 21 (Order relation $\le$ on coloured graphs)**.** For all pairs of coloured graphs $G, H \in \mathscr{G}_{\mathscr{C}}$ with $G = \langle V_G, E_G, c_G\rangle$, $H = \langle V_H, E_H, c_H\rangle$, $V_G = \{1, 2, \ldots, n\}$, $V_H = \{1, 2, \ldots, m\}$

$$\begin{aligned} &G \le H, \\ &\quad \text{if } n < m \text{ or } \big(n = m \text{ and } \big(\\ &\quad\quad |E_G| < |E_H| \\ &\quad\quad\ \text{or } \big(|E_G| = |E_H| \text{ and } \big(\\ &\quad\quad\quad (c_{G,1}, c_{G,2}, \ldots, c_{G,n}) < (c_{H,1}, c_{H,2}, \ldots, c_{H,n}) \\ &\quad\quad\quad\ \text{or } \big(\forall_{1 \le i \le n}(c_{G,i} = c_{H,i}) \text{ and } A(G) \le A(H)\big) \\ &\quad\quad \big)\big) \\ &\quad \big)\big). \end{aligned}$$

Proposition 4.7. $(\mathscr{G}_{\mathscr{C}}, \le)$ *is totally ordered.*

Proof. The number of vertices and number of edges of graphs are totally ordered. If the number of vertices is equal for two graphs, then also the corresponding sequences of colours are totally ordered (the cartesian product of a totally ordered set is itself also totally ordered). Because the adjacency matrix can be expressed as a natural number the adjacency matrices are also totally ordered. And because from this information (number of vertices, number of edges, sequence of colours, and adjacency matrix) the graph can be reconstructed (in other words, the information captures all there is to know about the graph), the combination of this information as defined in Def. 21 is a total ordering, i.e., without further proof we can say that the relation $\le$ is

1) reflexive: $\forall G \in \mathscr{G}_{\mathscr{C}}$, $G \le G$;

2) antisymmetric: $\forall G, H \in \mathscr{G}_{\mathscr{C}}$,

$$G \le H \wedge H \le G \implies G = H;$$

3) transitive: $\forall G_1, G_2, G_3 \in \mathscr{G}_{\mathscr{C}}$,

$$G_1 \leq G_2 \wedge G_2 \leq G_3 \implies G_1 \leq G_3; \text{ and}$$

4) total: $\forall G, H \in \mathscr{G}_{\mathscr{C}}$, either $G \leq H$ or $H \leq G$. □

4.3 Generating a search tree for a canonical relabelling partition

The partition refinement is used in the generation of a search tree that is used to find the discrete partition that will be used for a canonical relabelling of the vertices. The search tree is generated in the following way. Given an initial (refined) partition, a non-trivial (non-singleton) cell W is selected, of which one vertex is chosen that is deleted from the cell and put in a separate (singleton) cell. This is done for each of the vertices in the cell, resulting in $|W|$ different partitions. The new partitions are then refined, and then the same procedure is followed for the resulting partitions. This is repeated until the partitions are discrete. The procedure is shown in Alg. 5.

Algorithm 5 Generate the search tree $T(G,\pi)$ for a directed graph $G = \langle V, E\rangle$ and partition $\pi \in \underset{\sim}{\Pi}(V)$, where the nodes are partitions of V. The root node of the tree is the coarsest stable refinement of π and the leaf nodes are discrete partitions of V. The result is a list of paths in the tree, which is an alternative representation of the tree itself.

```
1:  k := 1
2:  π_1 := refine(G, π, π)
3:  Let W_1 be the first non-trivial cell of π_1 of the smallest size.
4:  Let τ be a list with only the singleton path π_1 as an element.
5:  while k ≥ 1 do
6:      if π_k is discrete then
7:          k := k − 1
8:      end if
9:      if k ≥ 1 then
10:         if W_k ≠ ∅ then
11:             {The vertex identies are used to order the vertices.}
12:             v := min W_k
13:             W_k := W_k \ v
                {Suppose π_k = (V_1, V_2, ..., V_r) and v ∈ V_i at this point.}
14:             π_k' := (V_1, ..., V_{i−1}, {v}, V_i \ v, V_{i+1}, ..., V_r)
15:             π_{k+1} := refine(G, π_k', (v))
16:             k := k + 1
17:             Add the path (π_1, π_2, ... π_k) to τ.
18:             Let W_k be the first non-trivial cell of π_k of the smallest
                size.
19:         else
20:             k := k − 1
21:         end if
22:     end if
23: end while
24: return τ;
```

The result of the algorithm is a search tree of which the root node is the refinement of the initial partition, $\pi_1 = \mathit{refine}(G,\pi,\pi)$. The smallest non-trivial cell is selected and for each vertex v in that cell a subtree is added of which the root node is the refinement of the partition in which v is put in a separate singleton cell. An edge between the root node and the subtree is added, labelled v. The subtrees have the same structure. The nodes in the search tree are the intermediate refined partitions, resulting from individualising the non-trivial cells and refining the partitions. The edges of the search tree are labelled with the vertex that is isolated from its cell. The discrete partitions form the leaf nodes of the tree.

The search tree can be interpreted as sequences of traces from the root node to the discrete leaf nodes, defined as follows.

Definition 22 (Search tree)**.** A *search tree* $T(G,\pi)$ is the set of all paths $(\pi_1, \pi_2, \ldots, \pi_m)$ that are derived from the directed graph G, an ordered partition π, and a sequence $v_1, v_2, \ldots, v_{m-1}$ where, for $1 \leq i \leq m-1$, v_i is an element of the first non-trivial cell V_k of π_i which has the smallest size:

$$\forall V_j \in \pi_i \cdot k \neq j \implies |V_k| < |V_j| \vee (|V_k| = |V_j| \wedge k < j).$$

The derivation is established in the following way. π_1 is the coarsest stable refinement of π, i.e., $\pi_1 = \mathit{refine}(G,\pi,\pi)$. The successors are defined in terms of their predecessors. π_{i+1} is derived from π_i and v_i by partition refinement such that $\pi_{i+1} = \mathit{refine}(G, \pi_i \downarrow v_i, (v_i))$, where $\pi_i \downarrow v$ is defined for $\pi_i = (V_1, V_2, \ldots, V_r)$ and $v \in V_k \in \pi_i$ as

$$\pi_i \downarrow v = (V_1, \ldots, V_{k-1}, \{v\}, V_k \setminus v, V_{k+1}, \ldots, V_r).$$

The relation between the partitions π_i and the vertices v_i is represented by the following notation:

$$\pi_1 \xrightarrow{v_1} \pi_2 \xrightarrow{v_2} \cdots \xrightarrow{v_{m-1}} \pi_m.$$

Proposition 4.8. *Given a graph G, a stable partition $\pi \in \Pi(V)$ and an element $v \in V$, $\mathit{refine}(G, \pi \downarrow v, (v))$ yields a stable partition and $\mathit{refine}(G, \pi \downarrow v, (v)) \sqsubseteq \pi$.*

Proof. This has been proved in [15]. □

Because π_1 is stable and because of Prop 4.8, all partitions π_i in the search tree have to be stable. Note that for all sequences $(\pi_1, \pi_2, \ldots, \pi_m) \in T(G,\pi)$ it holds that

$$\pi_m \sqsubseteq \cdots \sqsubseteq \pi_2 \sqsubseteq \pi_1.$$

We write $X(G,\pi)$ for the set of all leaf nodes of $T(G,\pi)$, i.e., sequences of which the last element is a discrete partition. These ordered discrete partitions can be used as permutation of the vertices of the graph in the sense of Definition 13. We write π_λ for the discrete partition of leaf node λ and $\overline{\lambda}$ for the permutation associated with π_λ. For the set of all graphs resulting from the permutations that are generated by the search tree we write

$$P(G,\pi) = \{G^{\overline{\lambda}} \mid \lambda \in X(G,\pi)\}.$$

In [15, Theorem 2.14] it is stated that $T(G^\gamma, \pi^\gamma) = T(G,\pi)^\gamma$, in other words, for every sequence in $T(G,\pi)$ there is an equivalent sequence in $T(G^\gamma, \pi^\gamma)$. We reformulate and prove this property in the following lemma.

Lemma 4.9. *For two isomorphic coloured graphs G and G^γ ($\gamma \in S_V$),*

$$(\pi_1, \pi_2, \ldots, \pi_m) \in T(G,\pi) \iff (\pi_1^\gamma, \pi_2^\gamma, \ldots, \pi_m^\gamma) \in T(G^\gamma, \pi^\gamma).$$

Proof. First $(\pi_1, \pi_2, \ldots, \pi_m) \in T(G,\pi) \implies (\pi_1^\gamma, \pi_2^\gamma, \ldots, \pi_m^\gamma) \in T(G^\gamma, \pi^\gamma)$ (by induction). Consider the sequence

$$\pi_1 \xrightarrow{v_1} \pi_2 \xrightarrow{v_2} \cdots \xrightarrow{v_{m-1}} \pi_m.$$

1) Base step: $\pi_1 \in T(G,\pi) \implies \pi_1^\gamma \in T(G^\gamma, \pi^\gamma)$. For the initial partitions π and π^γ it holds that for all vertices $v \in V_G$, if v is in the i-th cell of π then v^γ is in the i-th cell of π^γ.

2) Induction step: If there exist $(\pi_1, \ldots, \pi_k, \ldots) \in T(G,\pi)$ and $(\pi'_1, \ldots, \pi'_k, \ldots) \in T(G^\gamma, \pi^\gamma)$ such that $\pi'_i = \pi_i^\gamma$ for $1 \le i \le k$, then π_k is discrete or there exist $(\pi_1, \ldots, \pi_k, \pi_{k+1}, \ldots) \in T(G,\pi)$ and $(\pi'_1, \ldots, \pi'_k, \pi'_{k+1}, \ldots) \in T(G^\gamma, \pi^\gamma)$ such that for $1 \le i \le k+1$, $\pi'_i = \pi_i^\gamma$.

   (a) Assume $(\pi_1, \ldots, \pi_k, \ldots) \in T(G,\pi)$ and $(\pi'_1, \ldots, \pi'_k, \ldots) \in T(G^\gamma, \pi^\gamma)$ such that $\pi'_i = \pi_i^\gamma$ for $1 \le i \le k$;

   (b) For the partitions π_k and π_k^γ it holds that for all vertices $v \in V_G$, if v is in the i-th cell of π_k then v^γ is in the i-th cell of π_k^γ;

   (c) This means that (if π_k is not discrete) the cell of π_k that is selected for the k-th iteration of generating $T(G,\pi)$ (the first smallest non-trivial cell), containing v_k, has an equivalent in π_k^γ that will be selected first in generating $T(G^\gamma, \pi^\gamma)$, which contains v_k^γ;

   (d) From this follows that $T(G^\gamma, \pi^\gamma)$ contains a branch starting with $\pi_k^\gamma \xrightarrow{v_k^\gamma}$;

   (e) The position of the cells of the resulting partition $\pi_k \downarrow v_k$ will still be equivalent to $\pi_k^\gamma \downarrow v_k^\gamma$;

   (f) If for the partitions $\pi_k \downarrow v_k$ and $\pi_k^\gamma \downarrow v_k^\gamma$ the partitions are equivalent, then also the stable partitions $\pi_{k+1} = \mathit{refine}(G, \pi_k \downarrow v_k)$ and $\pi_{k+1}^\gamma = \mathit{refine}(G^\gamma, \pi_k^\gamma \downarrow v_k^\gamma)$ are equivalent, i.e., it holds that for all vertices $v \in V_G$, if v is in the i-th cell of π_{k+1} then v^γ is in the i-th cell of π_{k+1}^γ, because partition refinement preserves isomorphism of isomorphic partitions;

   (g) Hence, either π_k and π_k^γ are discrete or there exist $(\pi_1, \ldots, \pi_k, \pi_{k+1}, \ldots) \in T(G,\pi)$ and $(\pi'_1, \ldots, \pi'_k, \pi'_{k+1}, \ldots) \in T(G^\gamma, \pi^\gamma)$ such that for $1 \le i \le k+1$, $\pi'_i = \pi_i^\gamma$.

Similar for the symmetric case: $(\pi_1, \pi_2, \ldots, \pi_m) \in T(G,\pi) \impliedby (\pi_1^\gamma, \pi_2^\gamma, \ldots, \pi_m^\gamma) \in T(G^\gamma, \pi^\gamma)$. □

A consequence of the this lemma is that for isomorphic graphs with equivalent initial partitions equivalent leaf nodes are generated.

Lemma 4.10. *For two isomorphic coloured graphs G and G^γ ($\gamma \in S_V$),*

$$\pi_X \in X(G,\pi) \iff \pi_X^\gamma \in X(G^\gamma, \pi^\gamma).$$

Proof. $T(G^\gamma, \pi^\gamma) = T(G,\pi)^\gamma$ (Lemma 4.9) implies $X(G^\gamma, \pi^\gamma) = X(G,\pi)^\gamma$. □

The equivalence of the sets of leaf nodes means equivalence of the associated discrete partition, which results in equal graphs when used as permutation of the vertices.

Proposition 4.11. *If two graphs $G = \langle V, E_G, c_G\rangle$ and $H = \langle V, E_H, c_H\rangle$ are isomorphic, i.e., a function $\gamma \in S_V$ exists that maps vertices in G to vertices in H such that $G^\gamma = H$, then the sets of graphs resulting from permutations generated by the search tree contain exactly the same graphs, i.e., $P(G,\pi) = P(G^\gamma, \pi^\gamma)$.*

Proof. By Lemma 4.10. □

So, given a total ordering of graphs (see Section 4.2), we can choose the minimum of the set $P(G,\pi)$ as the canonical form of G.

4.4 Pruning the search tree

The number of leaf nodes in the search tree grows very large if the graph has a large automorphism group. The worst case complexity of the search tree generation is $O(|V|!)$. Therefore several heuristics need to be used to prune parts of the tree.

By choosing well the parts that we prune, we want to reduce the number of candidate graphs without losing the property of Prop. 4.11, i.e. that two isomorphic graphs will result in the same set of candidate graphs such that one graph is the minimum of both sets (which is the canonical form). There exist several methods to prune (large) parts of the search tree without losing the ability to compute a canonical form. The methods presented in [15] and [13] are based on finding automorphisms and on using leaf certificates and node invariants for nodes of the search tree.

Definition 23 (Leaf certificate)**.** For a leaf node $\lambda \in X(G,\pi)$, a *leaf certificate* $C(G,\pi,\lambda)$ is a certificate that maps leaf nodes to some value such that for all leaf nodes $\lambda_1, \lambda_2 \in X(G,\pi)$, and their associated partitions π_{λ_1} and π_{λ_2},

$$C(G,\pi,\lambda_1) = C(G,\pi,\lambda_2) \iff \exists \gamma \in \mathrm{Aut}(G,\pi) \text{ such that } \pi_{\lambda_1}^\gamma = \pi_{\lambda_2}.$$

A leaf certificate for which this holds is the combination of the permuted graph and the permuted initial partition: $C(G,\pi,\lambda) = \langle G^{\overline{\lambda}}, \pi^{\overline{\lambda}}\rangle$. The definition implies that if for two leaf nodes λ_1 and λ_2 the leaf certificates are equal, also the associated graphs are equal because there exists an automorphism $\gamma \in \mathrm{Aut}(G,\pi)$, such that

$$G^{\overline{\lambda_2}} = G^{\overline{\lambda_1}^{\gamma}} = (G^{\overline{\lambda_1}})^{\gamma} = G^{\overline{\lambda_1}}.$$

The automorphism implied by the equality of the certificates is determined by the leaf node partitions. If two leaf nodes $\lambda_1, \lambda_2 \in X(G,\pi)$ give rise to the same certificate, there exists an automorphism $\gamma = \overline{\lambda_1}\,\overline{\lambda_2^{-1}} \in \mathrm{Aut}(G,\pi)$. This will be used for detecting automorphisms in the graph during the generation of the search tree.

Definition 24 (Node invariant)**.** Given a node $\nu \in T(G,\pi)$ and the associated stable partition π_ν, a *node invariant* $I(G,\pi,\pi_\nu)$ is an invariant such that for all graph permutations $\gamma \in S_V$,

$$I(G,\pi,\pi_\nu) = I(G^\gamma, \pi^\gamma, \pi_\nu^\gamma).$$

An example of such an invariant is an integer value based on the number of vertices in the cells of the partition π_ν, e.g.,

$$I(G,\pi,\pi_\nu) = \prod_{W\in\pi_\nu} |W|.$$

Based on such a node invariant I also a new invariant can be defined that combines the invariant values of the nodes in a path of the search tree $\nu = (\pi_1, \pi_2, \ldots, \pi_l)$:

$$\vec{I}(G,\pi,\nu) = (I_1, I_2, \ldots, I_l),$$

where $I_i = I(G,\pi,\pi_i)$. This invariant will be used to select which paths are taken in the traversal of the search tree.

4.4.1 Pruning using automorphisms found

During the generation of the search tree, leaf nodes are encountered with associated discrete partitions. If we store the leaf nodes and the leaf certificates for these nodes, we can compute all automorphisms by comparing new certificates with the certificates stored. If for a graph $G = \langle V,E,c\rangle$ and initial partition $\pi \in \Pi(V)$, we discover a leaf node λ_1 with the same certificate as a stored leaf node λ_2, i.e., $C(G,\pi,\lambda_1) = C(G,\pi,\lambda_2)$, then there is an automorphism $\gamma = \overline{\lambda_1}\,\overline{\lambda_2^{-1}} \in \mathrm{Aut}(G,\pi)$. This automorphism induces an orbit partition, i.e., a partition $\{\{v\} \cup \{v^\gamma\} \mid v \in V\}$, which can be computed as follows. Let the two associated discrete partitions be $\pi_{\lambda_1} = (\{v_{1,1}\},\{v_{1,2}\},\ldots,\{v_{1,n}\})$ and $\pi_{\lambda_2} = (\{v_{2,1}\},\{v_{2,2}\},\ldots,\{v_{2,n}\})$. Then there is binary equivalence relation $\{(v_{1,1},v_{2,1}),(v_{1,2},v_{2,2}),\ldots,(v_{1,n},v_{2,n})\}$ with $v_{1,i}^\gamma = v_{2,i}$ for all $1 \le i \le n$. The transitive closure of this relation can be represented as a partition, which we call the *orbit partition* Φ.

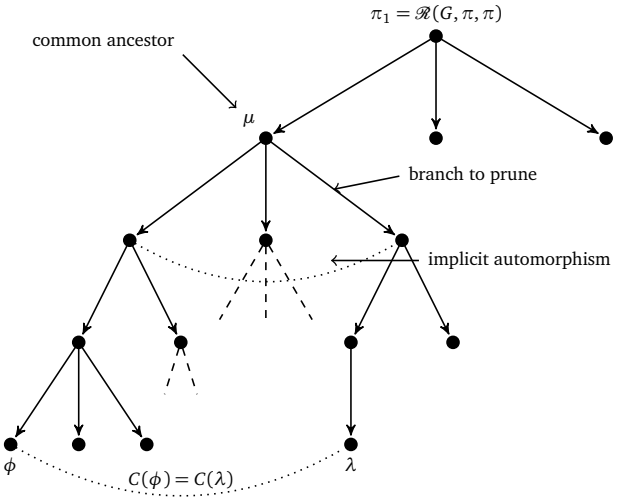


Figure 4: Pruning the search tree. The certificate values of the leaf nodes ϕ and λ are equal, so the branch of λ can be pruned from their common ancestor μ.

The orbit partition Φ is stored to be used in pruning parts of the search tree. The stored orbit partition is updated with new orbit partitions by computing the least upper bound of both (see Prop. 4.1 on how to determine the least upper bound of two partitions).

This orbit partition implies that in individualising a cell in the generation of the search tree (line 12 in Alg. 5), only one of the members of a cell of the orbit partition has to be considered. If we just have visited a branch in the search tree corresponding to vertex v, then we can skip branches from the same node if they correspond to a vertex that is in the same cell as v in orbit partition Φ.

Moreover, we can prune the whole subtree in which we found λ_1 up to its common ancestor with λ_2, the leaf node we stored earlier. This is visualised in Figure 4. There ϕ is the leaf node that is discovered before and λ is the newly discoved leaf node with the same leaf certificate as ϕ. The two leaf nodes having the same certificate implies that the two branches from their common ancestor in the search tree, which is μ in the figure, result from two individualised vertices that are automorphic. We can now prune the current subtree under the common ancestor, because the two branches result in the same values. This means we can backtrack to μ.

Because there can be a lot of leaf nodes, we do not want to store all leaf nodes and all leaf certificates. In Bliss two leaf nodes are stored [13]: the first leaf node we discovered, which is called ϕ, and the leaf node that leads to the 'best' candidate sofar, which is called ψ. The orbit partition Φ is stored for automorphisms with the first leaf node ϕ, and another one, Ψ for automorphisms with ψ. In Bliss also the m most recently found automorphisms are stored, where m can be set to some

convenient value.

4.4.2 Pruning using leaf certificates and node invariants

Using the discrete partitions associated with the leaf nodes we can permute the vertices of the original graph (as discussed in the previous section), i.e., the partition $\pi_{\overline{\lambda}}$ associated with leaf node λ is used to generate graph $G^{\overline{\lambda}}$. Doing this for all leaf nodes results in the set of graphs $P(G,\pi)$ of which the minimum can be chosen as the canonical form of G.

However, the leaf certificates and node invariants defined earlier enable to consider a smaller set in the following way. The set of graphs to consider can be limited to the graphs that result from leaf nodes with a minimum leaf certificate:

$$P_C(G,\pi) = \{G^{\overline{\lambda}} \mid \lambda \in X(G,\pi) \wedge C(G,\pi,\lambda) = \min\{C(G,\pi,\nu) \mid \nu \in X(G,\pi)\}\}$$

This can be taken a step further by reducing the set to only include leaf nodes with a minimum node invariant:

$$P_I(G,\pi) = \{G^{\overline{\lambda}} \mid \lambda \in X(G,\pi) \wedge C(G,\pi,\lambda) = \min\{C(G,\pi,\nu) \mid \nu \in X(G,\pi)\} \wedge \vec{I}(G,\pi,\lambda) = \min\{\vec{I}(G,\pi,\nu) \mid \nu \in X(G,\pi)\}\}$$

Because of the iterative nature of the vector $\vec{I}(G,\pi,\lambda)$, parts of the search tree that do not have a minimum node invariant can be pruned early in the search tree. If $\vec{I}(\lambda_1) < \vec{I}(\lambda_2)$ for some nodes λ_1 and λ_2, then also for every descendant λ_1' of λ_1 and descendant λ_2' of λ_2 it holds that $\vec{I}(\lambda_1') < \vec{I}(\lambda_2')$. So, if λ_1 and λ_2 are descendants of the same node ν, only child λ_1 has to be considered, because the subtree from λ_2 will not lead to leaf nodes with a minimum node invariant. Note that the ordering of nodes by the node invariant need not to be complete, so multiple children of a node in the search tree can have the same node invariant.

The better the node invariant function is in discriminating nodes, the larger the part of the search tree that is pruned. Usually this results in a trade-off between time spent on calculating the invariant and the reduction of the search tree that is achieved by using the invariant.

5 Conversion of edge labelled graphs to vertex coloured graphs and vice versa

In the algorithm explained in the previous section, vertex coloured graphs are used, while Groove uses edge labelled graphs (even node labels are labelled (self-)edges). In order to be able to use the existing algorithms, the edge labelled graphs have to be converted to vertex coloured graphs. How this can be done is described in Section 5.1.

Another conversion is needed to be able to use existing datasets of *undirected* coloured graphs in our experiments. This conversion is described in Section 5.2.

5.1 Conversion function from edge labelled graphs to vertex coloured graphs

We want to have a conversion function that preserves isomorphism of graphs, because then checking for isomorphism of the converted graphs yields the same result as checking for isomorphism of the original graphs. This enables the use of existing isomorphism checking algorithms that are built for vertex coloured graphs also for edge labelled graphs.

First we formally define the properties of such functions. Then we present two conversions that have these properties: a layered conversion function τ_1 inspired by [16] (and used in [28]) and a function τ_2 that converts each edge label into a distinct coloured vertex. In both methods a mapping between vertex colours and edge labels is maintained. An example for both methods is shown in Figure 6.

In Section 5.1.4 the difference in size of the resulting graphs from the two conversion functions is discussed.

5.1.1 Definition

Definition 25 (Isomorphism preserving conversion function τ_L)**.** A function $\tau_L : \mathscr{G}_{\mathscr{L}} \to \mathscr{G}_{\mathscr{C}}$ is called an *isomorphism preserving conversion function* from edge labelled graphs to vertex coloured graphs if for all $G, H \in \mathscr{G}_{\mathscr{L}}$,

$$G \cong H \iff \tau_L(G) \cong \tau_L(H).$$

If we have such an isomorphism preserving conversion function τ_L and we want to check if two edge labelled graphs G and H are isomorphic, then it suffices to check if $\tau_L(G)$ and $\tau_L(H)$ are isomorphic. Furthermore, if we want to store canonical forms to check if we have seen an isomorphic graph before, we can store the canonical form of the converted graphs. If *can* is a canonical representation function for coloured graphs, then

$$G \cong H \iff \tau_L(G) \cong \tau_L(H) \iff can(\tau_L(G)) = can(\tau_L(H)).$$

From now on we assume there to be a fixed, ordered set of labels $\mathscr{L} = \{l_1, l_2, \ldots, l_k\}$ that is the same for the different graphs that are compared.

5.1.2 Layered conversion τ_1

The idea of the layered conversion is to create a layer of copies of vertices for each distinct edge label. Edges can then be added in the layer that corresponds to its label. The layers are distinguished by the colours that are given to the vertices. The vertices that are a copy of the same orginal are linked to each other by a chain of edges, i.e.,

the copy of v in layer n has an outgoing edge to the copy of v in layer $n+1$. Moreover, the self-edges of vertices in an edge labelled graph are regarded as *vertex labels*, so the self-edges of a vertex can be encoded in the vertex colour and need not to be represented by edges in the resulting graph. This means that the colour of vertices in the converted graph is based on a combination of a) the edge label that is represented by the layer in which the vertex is placed, and b) the set of vertex labels (labels of self-edges) of the corresponding vertex in the original graph.

So, in the conversion a mapping is created from combinations of a set of vertex labels and an edge label to vertex colours. This mapping is shown in Figure 6b. The result of the conversion if in Figure 6d. In this conversion we assume the set of vertex labels (used in self-edges) and the set of edge labels (used in edges between vertices) to be disjoint. The set of edge labels is totally ordered, so we can define a successor relation $succ_G$.

For an ordered set of edge labels $L_E = \{l_1, l_2, \ldots, l_k\}$ (with $l_i < l_j \iff i < j$),

$$succ_G(l_i) = l_{i+1}.$$

Definition 26 ('Layered' conversion function τ_1)**.** Given a directed labelled graph $G = \langle V, E\rangle$ with its associated map from vertices to sets of vertex labels $L_V : V \to \mathscr{P}(\mathscr{L})$ and set of edge labels $L_E \subseteq \mathscr{L}$ with

$$\begin{aligned} L_V &= \{(v, \{l \mid (v,l,v) \in E\}) \mid v \in V\} \text{ and} \\ L_E &= \{l \mid (v_1, l, v_2) \in E \wedge v_1 \neq v_2\}, \end{aligned}$$

the converted graph is a coloured graph $\tau_1(G) = \langle V', E', c\rangle$ with

$$\begin{aligned} V' &= \{(v, (l_e, L_V(v))) \mid v \in V \wedge l_e \in L_E\}, \\ E' &= \{\big((v_1, (l_e, L_V(v_1))), (v_2, (l_e, L_V(v_2)))\big) \in V' \times V' \\ &\qquad \mid (v_1, l_e, v_2) \in E\} \\ &\quad \cup \{((v, (l_{e,1}, L_V(v))), (v, (l_{e,2}, L_V(v))) \in V' \times V' \\ &\qquad \mid l_{e,2} = succ_G(l_{e,1})\} \text{ and} \\ c &= \{(v', (l_e, l_v)) \mid v' = (v, (l_e, l_v)) \in V'\}. \end{aligned}$$

For a graph $G \in \mathscr{G}_{\mathscr{L}}$ and its converted form $G' = \tau_1(G)$, as a result of the conversion there are mappings $m_G : V_G \times L_E \to V_{G'}$ from vertices and edge labels in G to vertices in G', $o_G : V_{G'} \to V_G$ that maps vertices in the G' to the vertex in G from which it originates, and a label function $\lambda_G : V_{G'} \to L_E$ that maps vertices in G' to the edge label they represent (in other words, the layer of the vertex). They are given by

$$\begin{aligned} m_G &= \{\big((v, l_e), (v, (l_e, L_V(v)))\big) \mid l_e \in L_E \wedge v \in V_G\} \\ o_G &= \{\big((v, (l_e, l_v)), v\big) \in V_{G'} \times V_G\} \\ \lambda_G &= \{\big((v, (l_e, l_v)), l_e\big) \in V_{G'} \times L_E\}. \end{aligned}$$

From each vertex $v \in V_G$ there is also a sequence of vertices in G':

$$\begin{aligned} s_G = \{\big(v, \big((v, (l_1, l_v)), (v, (l_2, l_v)), \ldots, (v, (l_k, l_v))\big)\big) \\ \mid v \in V_G \wedge l_v = L_V(v) \\ \wedge l_1, l_2, \ldots, l_k \in L_E, l_i < l_j \iff i < j\}. \end{aligned}$$

Given a function $f : V_G \to V_H$, the *pointwise extension* of f to sequences, i.e., the function applied to a sequence of vertices resulting in the sequence of images of these vertices, is denoted by the same symbol $f : V_G^* \to V_H^*$ and is defined as:

$$f\big((v_1, v_2, \ldots, v_k)\big) = \big(f(v_1), f(v_2), \ldots, f(v_k)\big).$$

These functions and this notation will be used in the following proofs.

Proposition 5.1. *For all $G, H \in \mathscr{G}_{\mathscr{L}}$, $G \cong H \iff \tau_1(G) \cong \tau_1(H)$, i.e., the conversion function τ_1 is isomorphism preserving.*

Proof. Steps:

1) $G \cong H \implies \tau_1(G) \cong \tau_1(H)$.

   Assume $G \cong H$, i.e., a bijective function $f : V_G \to V_H$ exists that is an isomorphism between G and H. $G \cong H$ implies that the set of edge labels L_E is the same for both graphs.

   To prove: $\tau_1(G) \cong \tau_1(H)$, i.e. that a bijective function $g : V_{\tau_1(G)} \to V_{\tau_1(H)}$ exists that is an ismorphism between $\tau_1(G)$ and $\tau_1(H)$. Such a function is given by:

$$\begin{aligned} g = \{(v, v') \in V_{\tau_1(G)} \times V_{\tau_1(H)} \\ \mid v' = m_H\big(f(o_G(v)), \lambda_G(v)\big)\}. \end{aligned}$$

   It is easy to see that g is an isomorphism, i.e., for all $v \in V_{\tau_1(G)}$, $c(v) = c(g(v))$ and for all $v_1, v_2 \in V_{\tau_1(G)}$ and $l \in \mathscr{L}$,

$$(v_1, l, v_2) \in E_{\tau_1(G)} \iff (g(v_1), l, g(v_2)) \in E_{\tau_1(H)}.$$

2) $\tau_1(G) \cong \tau_1(H) \implies G \cong H$

   Assume $\tau_1(G) \cong \tau_1(H)$, i.e., there exists a bijective function $g : V_{\tau_1(G)} \to V_{\tau_1(H)}$ that is an isomorphism between $\tau_1(G)$ and $\tau_1(H)$.

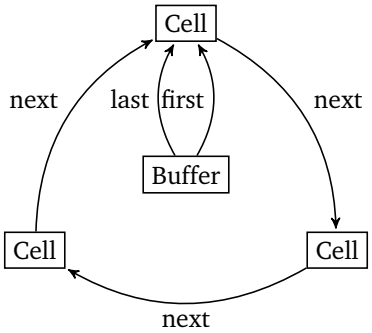
   Then there exists a bijective function $f : V_G \to V_H$ given by

$$f = \{(v, v') \in V_G \times V_H \mid s_{\tau_1(H)}(v') = g(s_{\tau_1(G)}(v))\}.$$

   It is easy to see that f is an isomorphism between G and H, i.e., for all $v_1, v_2 \in V_G$ and $l \in \mathscr{L}$,

$$(v_1, l, v_2) \in E_G \iff (f(v_1), l, h(v_2)) \in E_H.$$

   So we have to conclude that $G \cong H$, given $\tau_1(G) \cong \tau_1(H)$. □



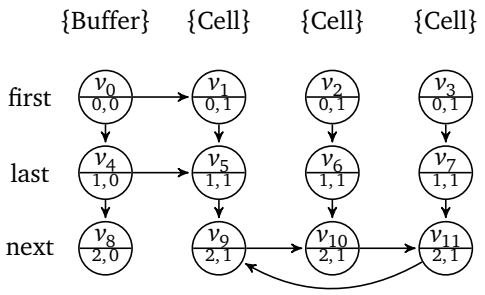
(a) An edge labelled graph G representing a bounded buffer. In the center is a 'Buffer' vertex, which points to the first and last cells, the vertices labelled 'Cell', that are occupied. The cells of the buffer are connected by 'next' edges.

	{Buffer} 0	{Cell} 1
first ↦ 0	0, 0	0, 1
last ↦ 1	1, 0	1, 1
next ↦ 2	2, 0	2, 1

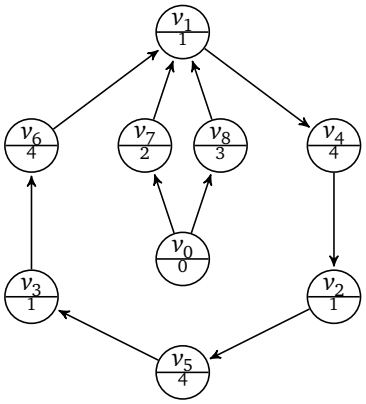
(b) The mapping between labels in the edge labelled graph and colours in the layered graph, used by τ_1. Each combination of set of vertex labels and row number in the coloured graph is mapped to a distinct colour. The rows in the coloured graph represent the different labels of edges between vertices in the labelled graph (which form the set of edge labels L_E).

{Buffer}	↦	0
{Cell}	↦	1
first	↦	2
last	↦	3
next	↦	4

(c) The mapping between labels in the edge labelled graph and colours in the graph with vertices for edge labels, used by τ_2. Each edge label that is used in the graph is mapped to a distinct colour. Also each set of vertex labels is mapped to a distinct colour.



(d) $\tau_1(G)$. For each vertex in the original graph there are $|L_E|$ copies in the graph, where L_E is the set of edge labels (not vertex labels) that are used in the original graph. This way $|L_E|$ layers of vertices are created that correspond to the different labels. The subsequent copies of a vertex are connected to each other by an edge. The vertex colours are according to the mapping in (b). For each edge in the original graph an edge is created between corresponding source and target vertices at the layer that corresponds to the edge label.



(e) $\tau_2(G)$. For each vertex in the original graph there is a copy in this graph. The vertex colours are according to the mapping in (c). For each edge a label vertex is created with the colour according to the mapping from edge label to colours. The copies of the original vertices are connected with the label vertices with unlabelled directed edges.

Figure 6: A bounded buffer represented by an edge labelled graph and its converted versions.

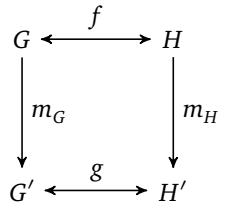


Figure 5: Overview of the proof of Prop. 5.1 and Prop. 5.2.

5.1.3 Label to vertex conversion τ_2

The 'label vertex' conversion creates a label vertex for each edge in the original graph with the colour corresponding to the label of the edge. Also for each vertex in the original graph a vertex is created in the resulting graph and edges are added between these vertices and the label vertices to connect the copies of the source and target vertices to the edges. For this conversion a mapping is maintained from edge labels to vertex colours. One special class of colours is reserved to designate vertices that represent the vertices in the original graph. These colours represent a set of vertex labels, i.e., the labels of the self-edges of the vertex in the original graph. Here again we assume the set of vertex labels and the set of edge labels to be disjoint. See Figure 6c for an example of the mapping and Figure 6e for the resulting converted graph.

Definition 27 ('Label vertex' conversion function τ_2)**.** $\tau_2 : \mathscr{G}_{\mathscr{L}} \to \mathscr{G}_{\mathscr{C}}$ is a conversion function that maps each directed labelled graph $G = \langle V, E\rangle$ to a coloured graph $\tau_2(G) = G' = \langle V', E', c\rangle$ with

$$\begin{aligned} V' &= V \cup \{(v_1, l, v_2) \in E \mid v_1 \neq v_2\},\\ E' &= \{(v_1, (v_1, l, v_2)) \mid (v_1, l, v_2) \in E\}\\ &\quad \cup \{((v_1, l, v_2), v_2) \mid (v_1, l, v_2) \in E\},\\ c &= \{(v, (\mathsf{v}, \lambda_V(v))) \mid v \in V\}\\ &\quad \cup \{((v_1, l, v_2), (\mathsf{e}, l)) \mid (v_1, l, v_2) \in E, v_1 \neq v_2\}, \end{aligned}$$

where $\lambda_V : V \to \mathscr{P}(\mathscr{L})$ is the map from vertices to their associated set of vertex labels

$$\lambda_V = \{(v, \{l \mid (v, l, v) \in E\}) \mid v \in V\}.$$

Here, v is used to mark a colour as belonging to a vertex in the original graph and e to mark a colour as representing the label of an edge in the original graph. This is needed to be able to distinguish 'edge' vertices from 'vertex' vertices in the resulting graph.

If $\tau_2(G) = G'$ then there exists mappings $m_{V_G} : V_G \to V_{G'}$ and $m_{E_G} : E_G \to V_{G'}$ resulting from the conversion that respectively map vertices and edges in G to vertices in G', given by

$$\begin{aligned} m_{V_G} &= \{(v, v) \in V_G \times V_{G'}\},\\ m_{E_G} &= \{(e, e) \in E_G \times V_{G'}\}. \end{aligned}$$

Proposition 5.2. *For all $G, H \in \mathscr{G}_{\mathscr{L}}$, $G \cong H \iff \tau_2(G) \cong \tau_2(H)$, i.e., the conversion function τ_2 is isomorphism preserving.*

Proof. Steps:

1) $G \cong H \implies \tau_2(G) \cong \tau_2(H)$.

   Assume $G \cong H$, i.e., a bijective function $f : V_G \to V_H$ exists that is an isomorphism between G and H.

   To prove: $\tau_2(G) \cong \tau_2(H)$, i.e. that a bijective function $g : V_{\tau_2(G)} \to V_{\tau_2(H)}$ exists that is an isomorphism between $\tau_2(G)$ and $\tau_2(H)$.

   Such a function is given by:

$$\begin{aligned} g = \{&(v, v') \in V_{\tau_2(G)} \times V_{\tau_2(H)} \mid v' = m_{V_H}(f(m_{V_G}^{-1}(v)))\}\\ &\cup \{(e, e') \in V_{\tau_2(G)} \times V_{\tau_2(H)}\\ &\quad \mid (v_1, l, v_2) = m_{E_G}^{-1}(e)\\ &\quad \wedge (f(v_1), l, f(v_2)) = m_{E_G}^{-1}(e')\}. \end{aligned}$$

   It is easy to see that g is an isomorphism, i.e.,

   (a) for all $v \in V_{\tau_2(G)}$, $c(v) = c(g(v))$, because the colour of v is based:
   - either on the self-edges of $m_{V_G}^{-1}(v)$ in G and the colour of $g(v)$ is by the same mapping based on the self-edges of $m_{V_H}^{-1}(g(v)) = f(m_{V_G}^{-1}(v))$ and f being an isomorphism implies that for every self-edge of $m_{V_G}^{-1}(v)$ labelled l there also exists a self-edge labelled l on $f(m_{V_G}^{-1}(v))$;
   - or on the label l of the edge $(v_1, l, v_2) = m_{E_G}^{-1}(v)$ and then $f(m_{E_G}^{-1}(v)) = m_{E_H}^{-1}(g(v))$ has the same label, so $g(v)$ must have the same colour as v;

   (b) for all $v_1, v_2 \in V_{\tau_2(G)}$ and $l \in \mathscr{L}$,

$$(v_1, v_2) \in E_{\tau_2(G)} \iff (g(v_1), g(v_2)) \in E_{\tau_2(H)}.$$

   Following from the definition, either v_1 or v_2 results from a vertex in G, the other results from an edge in G. Assume v_1 results from a vertex, then $v_1' = m_{V_G}^{-1}$ is that vertex. Then $(v_1', l, w) = m_{V_E}^{-1}(v_2)$ is the edge from which v_2 originates, with $c(v_2) = (\mathsf{e}, l)$. Because f is an isomorphism, there also has to be a vertex $f(w) \in V_H$ such that there exists an edge $e = (f(v_1'), l, f(w)) \in E_H$. Then there also has to be a vertex $m_{E_H}(e) = g(v_2) \in V_{\tau_2(H)}$ and from the definition of the conversion it follows that there is an edge between $m_{V_H}(f(v_1)) = g(v_1)$ and $g(v_2)$, so $(g(v_1), g(v_2)) \in E_{\tau_2(H)}$. A similar argument can be given for the case that v_2 results from a vertex and v_1 from an edge. So, $(v_1, v_2) \in E_{\tau_2(G)} \implies (g(v_1), g(v_2)) \in E_{\tau_2(H)}$. The same argument also holds for the symmetric case: $(g(v_1), g(v_2)) \in E_{\tau_2(H)} \implies (v_1, v_2) \in E_{\tau_2(G)}$.

2) $\tau_2(G) \cong \tau_2(H) \implies G \cong H$.

Assume $\tau_2(G) \cong \tau_2(H)$, i.e., there exists a bijective function $g : V_{\tau_2(G)} \to V_{\tau_2(H)}$ that is an isomorphism between $\tau_2(G)$ and $\tau_2(H)$. Then there exists a bijective function $f : V_G \to V_H$ given by

$$f = \{(v, v') \in V_G \times V_H \mid m_{V_G}(v') = g(m_{V_G}(v))\}.$$

It is easy to see that f is an isomorphism between G and H, i.e., for all $v_1, v_2 \in V_G$ and $l \in \mathscr{L}$,

$$(v_1, l, v_2) \in E_G \iff (f(v_1), l, f(v_2)) \in E_H.$$

So we have to conclude that $G \cong H$, given $\tau_2(G) \cong \tau_2(H)$. □

5.1.4 Size of the converted graphs

For an edge labelled graph $G = \langle V, E\rangle \in \mathscr{G}_{\mathscr{L}}$, number of vertices $n = |V|$, number of non-self-edges $m = |E \setminus E_S|$ (with E_S being the set of self-edges of G), and number of edge labels $k = |L_E|$, we can say the following about the size of the converted graphs. The layered conversion τ_1 results in $n \cdot k$ vertices and $n \cdot (k-1) + m$ edges. The label vertex conversion τ_2 results in $n + m$ vertices and $2 \cdot m$ edges. It depends on the number of labels and the density of the graph which of the methods is more efficient. With both methods the number of vertices and the number of edges usually increase by the conversion. This is not the case when there are a lot of self-edges.

5.2 Conversion function from undirected coloured graphs to edge labelled graphs

In this section we discuss the conversion from *undirected* coloured graphs to edge labelled graphs. We need this kind of conversion to use existing datasets of undirected coloured graphs for our experiments, in which we compare methods for checking isomorphism of edge labelled graphs.

Undirected coloured graphs can be converted to edge labelled graphs such that isomorphism of two undirected coloured graphs implies isomorphism of their converted forms. We first define this kind of conversion functions with this property and then give a conversion function that has this property.

5.2.1 Definition

Definition 28 (Isomorphism preserving conversion function τ_C)**.** A function $\tau_C : \mathscr{G}_{\mathscr{C}} \to \mathscr{G}_{\mathscr{L}}$ is called an *isomorphism preserving conversion function* from undirected coloured graphs to edge labelled graphs if for all $G, H \in \mathscr{G}_{\mathscr{C}}$,

$$G \cong H \iff \tau_C(G) \cong \tau_C(H).$$

In the following we use C as the set of colours and $\mathscr{L} = \mathsf{C} \cup \{\mathsf{e}\}$ as the set of edge labels (with $\mathsf{e} \notin \mathsf{C}$). The colours in C are used to represent vertex colours, e is a special label used to denote edges.

5.2.2 Conversion function τ_3

An isomorphism preserving conversion can be done in the following way. Each vertex v of an undirected coloured graph G_V is converted into a vertex v' in the edge labelled result graph G_E and a self-edge is attached to v' with the label that corresponds to the colour of v. For every (undirected) edge (v_1, v_2) in G_V, edges, labelled with e, are added in both directions between the vertices in G_E that represent v_1 and v_2. This is expressed in the following definition.

Definition 29 (Edge-vertex conversion function)**.** $\tau_3 : \mathscr{G}_{\mathscr{C}} \to \mathscr{G}_{\mathscr{L}}$ is a conversion function that maps each undirected coloured graph $G = \langle V, E, c\rangle$ to a directed labelled graph $\tau_3(G) = \langle V', E'\rangle$ with

$$\begin{aligned} V' &= V, \\ E' &= \{(v_1, \mathsf{e}, v_2) \in V' \times \mathscr{L} \times V' \\ &\qquad \mid (v_1, v_2) \in E \vee (v_2, v_1) \in E\} \\ &\quad \cup \{(v, l, v) \in V' \times \mathscr{L} \times V' \mid l = c(v)\}. \end{aligned}$$

An example of this conversion is in Figure 7. We will show that this definition meets the criterion in Definition 28, i.e., that the conversion function τ_3 is isomorphism preserving. For this we need the following notation.

For a graph $G \in \mathscr{G}_{\mathscr{C}}$ and its converted form $G' = \tau_3(G)$, as a result of the conversion there is a mapping $m_{V_G} : V_G \to V_{G'}$ from vertices in G to vertices in G' and a mapping $m_{E_G} : E_G \to E_{G'}$ from edges in G to edges in G' given by

$$\begin{aligned} m_{V_G} &= \{(v, v) \in V_G \times V_{G'}\}, \\ m_{E_G} &= \{(e, e) \in E_G \times E_{G'}\}. \end{aligned}$$

Proposition 5.3. *For all $G, H \in \mathscr{G}_{\mathscr{C}}$, $G \cong H \iff \tau_3(G) \cong \tau_3(H)$, i.e., the conversion function τ_3 is isomorphism preserving.*

Proof. Steps:

1) $G \cong H \implies \tau_3(G) \cong \tau_3(H)$.

Assume $G \cong H$, i.e., a bijective function $f : V_G \to V_H$ exists that is an isomorphism between G and H.

To prove: $\tau_3(G) \cong \tau_3(H)$, i.e., that a bijective function $g : V_{\tau_3(G)} \to V_{\tau_3(H)}$ exists that is an isomorphism between $\tau_3(G)$ and $\tau_3(H)$. Such a function is given by:

$$g = \{(v, v') \in V_{\tau_3(G)} \times V_{\tau_3(H)} \mid v' = m_{V_H}(f(m_{V_G}^{-1}(v)))\}.$$

g is an isomorphism, i.e., for all $v_1, v_2 \in V_{\tau_3(G)}$ and $l \in \mathsf{C}$,

$$(v_1, l, v_2) \in E_{\tau_3(G)} \iff (g(v_1), l, g(v_2)) \in E_{\tau_3(H)} :$$

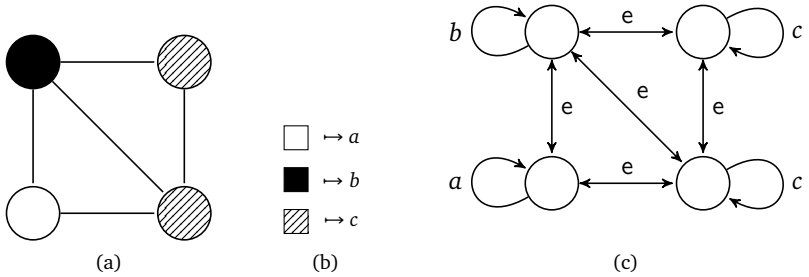


Figure 7: An example of the conversion of an undirected coloured graph to an edge labelled directed graph by conversion function τ_3. (a) shows the undirected coloured graph G. (b) shows the mapping between colours in G to labels in the labelled graph $\tau_3(G)$. (c) shows the resulting edge labelled graph $\tau_3(G)$.

(a) For the self-edges (v,l,v) in $E_{\tau_3(G)}$ there is a vertex $v' = m_{V_G}^{-1}(v)$ in G with colour l, so there is a vertex $f(v')$ in H with the same colour, hence also a vertex $m_{V_H}(f(v')) = g(v)$ in $\tau_3(H)$ with a self-edge labelled l: $(g(v),l,g(v)) \in E_{\tau_3(H)}$.
The same holds for the symmetric case: $(g(v),l,g(v)) \in E_{\tau_3(H)} \implies (v,l,v) \in E_{\tau_3(G)}$.

(b) For the edges $e = (v_1, \mathsf{e}, v_2)$ in $E_{\tau_3(G)}$ there exists an edge $e' = m_{E_G}^{-1}(e)$ in G between vertices $v_1' = m_{V_G}^{-1}(v_1)$ and $v_2' = m_{V_G}^{-1}(v_2)$ in G, so also between $f(v_1')$ and $f(v_2')$ in H, hence also, because of the conversion, between $m_{V_H}(f(v_1')) = g(v_1)$ and $m_{V_H}(f(v_2'))g(v_2)$ in $\tau_3(H)$: $(g(v_1),\mathsf{e},g(v_2)) \in E_{\tau_3(H)}$.
The same holds for the symmetric case: $(g(v_1),\mathsf{e},g(v_2)) \in E_{\tau_3(H)} \implies (v_1,\mathsf{e},v_2) \in E_{\tau_3(G)}$.

2) $\tau_3(G) \cong \tau_3(H) \implies G \cong H$.

Assume $\tau_3(G) \cong \tau_3(H)$, i.e., there exists a bijective function $g : V_{\tau_3(G)} \to V_{\tau_3(H)}$ that is an isomorphism between $\tau_3(G)$ and $\tau_3(H)$. Then there exists a bijective function $f : V_G \to V_H$ given by

$$f = \{(v,v') \in V_G \times V_H \mid v' = m_{V_H}^{-1}(g(m_{V_G}(v)))\}.$$

f is an isomorphism between G and H, i.e.,

(a) for all $v \in V_G$, $c(v) = c(h(v))$, and

(b) for all $v_1, v_2 \in V_G$,

$$(v_1,v_2) \in E_G \iff (h(v_1),h(v_2)) \in E_H.$$

Because if $(v_1,v_2) \in E_G$, then there also exists $(m_G(v_1),\mathsf{e},m_G(v_2)) \in E_{\tau_3(G)}$ and hence also an edge $(g(m_G(v_1)),\mathsf{e},g(m_G(v_2))) \in E_{\tau_3(H)}$, which has to originate from an edge $(h(v_1),h(v_2))$ in H, so

$$(v_1,v_2) \in E_G \implies (h(v_1),h(v_2)) \in E_H.$$

The same holds for the symmetric case: $(h(v_1),h(v_2)) \in E_H \implies (v_1,v_2) \in E_G$.

So we have to conclude that $G \cong H$, given $\tau_3(G) \cong \tau_3(H)$. □

The complexity of this conversion is linear in the size of the graph: $\Theta(|V_G| + |E_G|)$.

6 Experiments

6.1 Experiment setup

6.1.1 Combinations of tools and conversions.

The combinations of isomorphism checking tools and conversion methods included in the experiments are:

Groove The algorithm that is already implemented in Groove, which checks if two edge-labelled graphs are isomorphic;

Bliss-layered Conversion function τ_1 combined with the tool Bliss.

Nauty-layered Conversion function τ_1 combined with the tool Nauty.

Bliss-labelvertex Conversion function τ_2 combined with the tool Bliss.

Nauty-labelvertex Conversion function τ_2 combined with the tool Nauty.

6.1.2 Graphs.

Directed, edge labelled graphs. We chose a set of 13 graphs of various sizes that are used as state graphs in model checking. For each of these graphs, we have an isomorphic variant and a non-isomorphic but otherwise similar one. Some of the graphs are in Fig. 8.

no-hops-* States in the model of an ad-hoc network connectivity protocol that is described and used in [27]. The graphs used here have four or seven vertices in the network and a designated scheduler node. The graphs are quite symmetric.

circ-buffer-3 State in a model of a circular buffer with three cells.

din-phil-* States in a model of the philosophers problem with three philosophers.

checkers-s* States of the checkers boardgame. The graphs are not very symmetric, because several different labels are used to distinguish places on the board.

seq3-flow2-* A large and mildly dense graph modelling a sequence of actions in a program. The graph is not very symmetric. The graphs with suffixes -copy2, -copy3 and -copy4 contain two, three, respectively four copies of the original graph, with added edges to make the graphs connected.

no-hops-lts Labelled Transition System (LTS) for the model of an ad-hoc network connectivity protocol (see above) with five network nodes. The LTS does contain cycles.

Undirected coloured graphs. We also selected a collection of undirected coloured graphs, available from the `bliss` website [12], in order to be able to compare the performance of GROOVE with the other tools in the case of larger, undirected graphs, for which those tools are tailored. These graphs are known to be hard; the graphs have been used to compare the efficiency of graph isomorphism checking tools. Although graphs in BLISS and NAUTY are coloured, these graphs do not use colours, i.e., all vertices have the same colour. Because of this we expect the layered conversion to perform better than the label-vertex conversion, because conversion τ_2 adds a vertex for each edge in the original graph, even if edge labels are not used. The layered conversion will use one layer without unnecessary overhead.

ag2-* Affine geometries.

cfi-* Cai-Fürer-Immerman graphs.

k-* Complete graphs K_n with n vertices and $\binom{n}{2}$ undirected edges.

rnd-3-reg-* Random 3-regular graphs, varying from 1,000 to 50,000 vertices.

6.1.3 Experiment environment

The experiments with coloured graphs were performed on a system with two Quad Core Xeon 1.86GHz CPU's and 8GB of memory running openSUSE 10.2 with Linux kernel 2.6.27 (64 bit). GROOVE version 3.3.1 was used with a Java 1.6.0 VM with a maximum of 3640 MB of memory. NAUTY version 2.4b7 [17] and BLISS version 0.50 [12] were employed.

The experiments with edge-labelled graphs were performed on a system with 4 dual Intel E5520 CPUs and 24GB RAM. GROOVE 4.0.1 has been used with a Sun Java 1.6.0 64-bit VM with a maximum of 20GB of memory for each core. The same versions of NAUTY and BLISS were used as for the coloured graphs.

The experiments involved pair-wise comparison of the edge-labelled graphs given above for isomorphism. When using a graph conversion, the edge-labelled graphs are first converted to a node-coloured graph and their canonical forms are computed, then these canonical forms are checked for equality. GROOVE checks for isomorphism of two edge-labelled graphs directly. The graphs in the collection of undirected coloured graphs are first converted to directed, edge labelled graphs by converions τ_3.

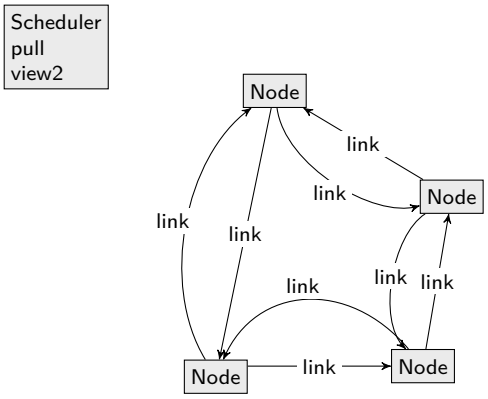
Each experiment was performed 10 times and we give the average execution time. The execution times are recorded by using the `System.nanoTime()` timer in Java. The `bliss` and `nauty` executables are called from Java and communication between the Java program and these external programs is done through `stdin` and `stdout` streams and writing and reading to and from files. Since most tools finish within seconds and we are interested in algorithms that are fast, experiments lasting longer than 5 minutes were aborted.

6.2 Results

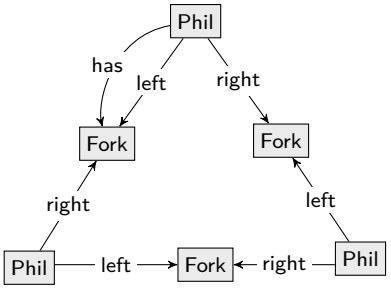
Directed, edge labelled graphs. Table 1 presents information about the size of the graphs used. Lines marked with $|V|$ and $|E|$ indicate the number of vertices and edges of a graph, respectively. Lines marked with $|L|$ and $|L_E|$ indicate the number of distinct (edge) labels in the graph. For each conversion method we give the conversion time in *ms*. From the table we see that the layered conversion yields smaller graphs only when the original graph is very small (< 10 vertices). On the other cases the label-vertex conversion is clearly better.

The running times for the tools are in Table 2. We see that there is not much difference in execution time between isomorphic and non-isomorphic pairs of graphs for NAUTY and BLISS. GROOVE, however, performs much worse for non-isomorphic pairs of graphs than for isomorphic pairs. From the results we see that NAUTY does well for some small graphs, but is unable to give an answer within 5 minutes for the larger graphs. BLISS with the label-vertex conversion (τ_2) is the method that perform closest to GROOVE, in the case of non-isomorphic pairs of graphs, but still is usually two times slower, except for the largest graph (no-hops-lts). Performance is clearly related to the size of the graph after conversion. For most graphs the label-vertex conversion (τ_2) results in smaller converted graphs and in smaller execution times. Fig. 9 gives plots of the values in Table 2. The y-axis has a logarithmic scale.

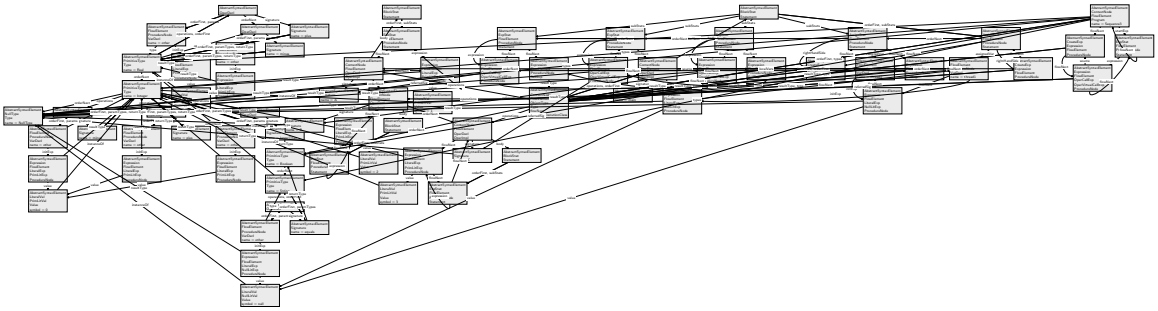
Undirected coloured graphs. The tables in Appendix A show the size of the coloured graphs used (the columns $|V|$ and $|E|$ indicate the number of vertices and edges of a graph, respectively) and average running



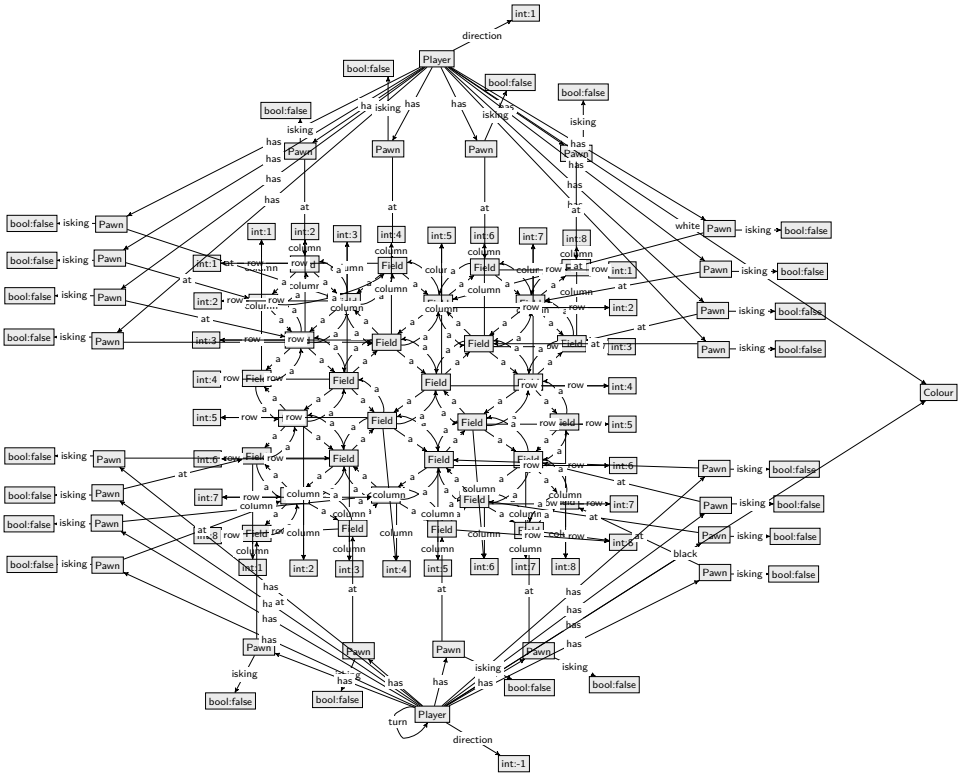
(a) no-hops-4 (5 vertices, 15 edges). State in the model of an ad-hoc network connectivity protocol.



(b) din-phil-1 (6 vertices, 13 edges). State in a model of the dining philosophers problem with three philosophers. René Descartes has already picked up the fork on his left.



(c) seq3-flow2 (75 vertices, 549 edges). A sequence of actions in a program.



(d) checkers-s1 (117 vertices, 357 edges). Start state of the checkers boardgame.

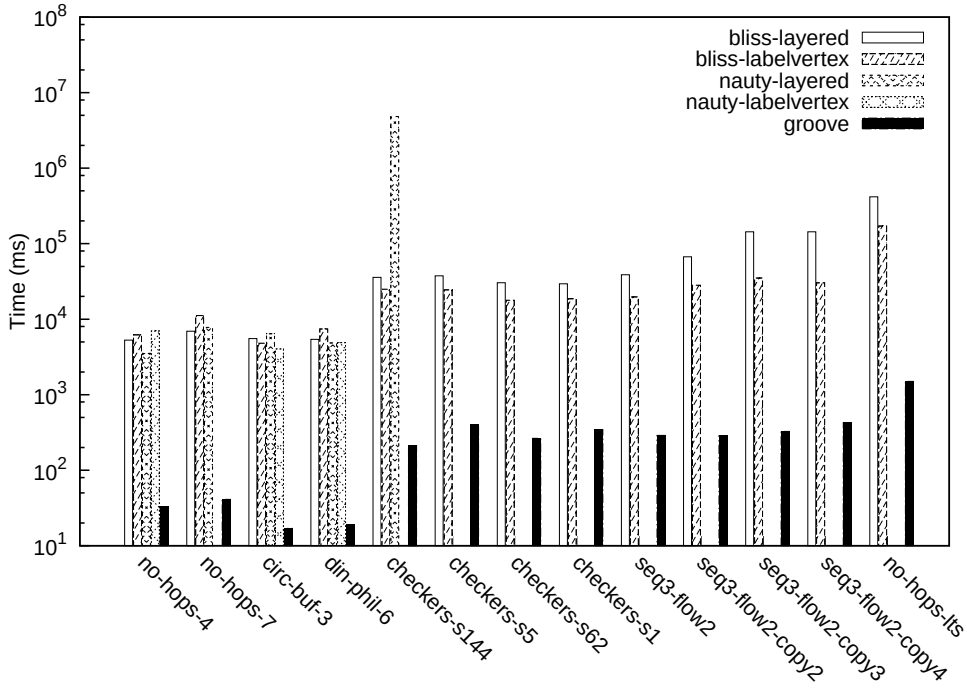
Figure 8: Some of the graphs used in the experiments.

Table 1: Results for the conversion methods. Time is in *ms*.

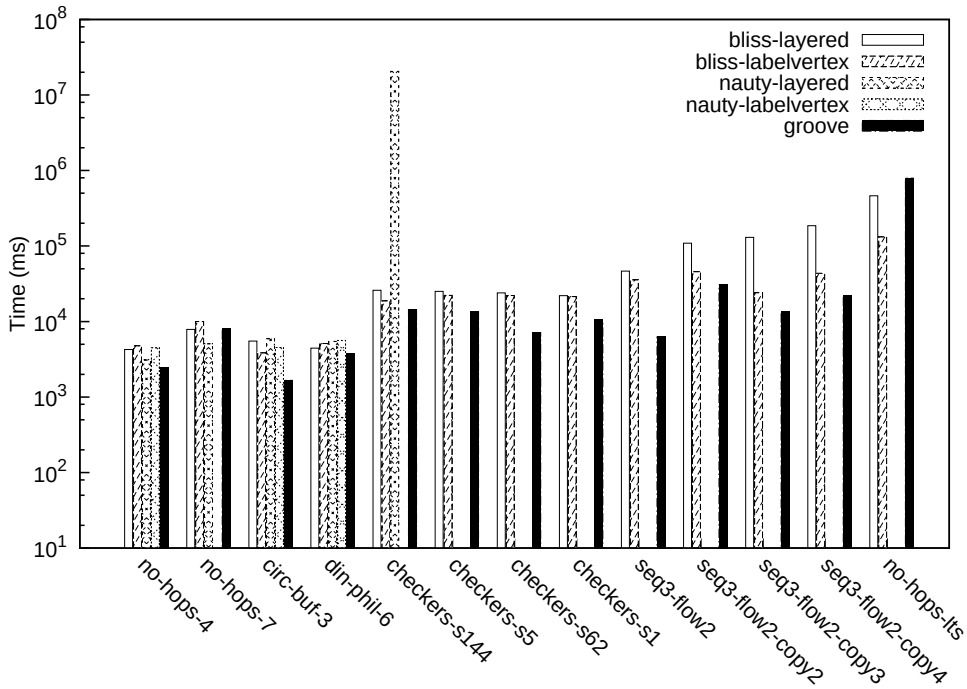
		no-hops-4	no-hops-7	circ-buf-3	din-phil-6	checkers-s144	checkers-s5	checkers-s62	checkers-s1	seq3-flow2	seq3-flow2-copy2	seq3-flow2-copy3	seq3-flow2-copy4	no-hops-lts
Original	$\|V\|$	5	8	4	6	70	69	70	117	75	150	225	300	2,143
graph	$\|E\|$	15	45	9	12	294	309	303	357	549	1,099	1,646	2,193	5,617
labels	$\|L\|$	5	5	5	4	26	24	25	24	75	76	76	76	31
edge labels	$\|L_E\|$	1	1	3	2	9	9	9	9	23	24	24	24	7
Layered	$\|V\|$	5	8	12	12	630	621	630	1,053	1,725	3,600	5,400	7,200	15,001
conversion	$\|E\|$	8	35	13	12	782	791	792	1,175	1,858	3,867	5,798	7,729	16,578
τ_1	time	1.2	2.0	1.2	1.2	10.3	16.9	7.5	8.3	10.2	14.6	34.9	32.4	100.6
Label-vertex	$\|V\|$	13	43	9	12	292	308	302	356	283	567	848	1,129	5,863
conversion	$\|E\|$	16	70	10	12	444	478	464	478	416	834	1,246	1,658	7,440
τ_2	time	1.3	2.3	0.8	1.7	6.7	7.4	4.9	6.3	5.9	8.1	7.8	6.2	43.1

Table 2: Results for the experiments performed. Time is in *ms*.

		no-hops-4	no-hops-7	circ-buf-3	din-phil-6	checkers-s144	checkers-s5	checkers-s62	checkers-s1	seq3-flow2	seq3-flow2-copy2	seq3-flow2-copy3	seq3-flow2-copy4	no-hops-lts
Iso	GROOVE	0.0	0.0	0.0	0.0	0.2	0.4	0.3	0.3	0.3	0.3	0.3	0.4	1.5
	BLISS-layered	5.3	6.9	5.5	5.4	35.8	37.4	30.4	29.4	38.8	67.0	143.4	143.8	416.8
	NAUTY-layered	3.5	7.8	6.4	4.8	4,832.6	–	–	–	–	–	–	–	–
	BLISS-labelvertex	6.2	11.1	4.8	7.4	24.9	24.4	17.8	18.6	19.7	28.2	34.9	30.2	170.7
	NAUTY-labelvertex	7.0	–	4.0	4.9	–	–	–	–	–	–	–	–	–
Non-iso	GROOVE	2.4	8.0	1.7	3.7	14.5	13.5	7.1	10.5	6.2	30.7	13.3	22.1	790.7
	BLISS-layered	4.3	7.9	5.5	4.4	25.9	25.1	23.9	22.0	46.6	108.9	130.0	185.4	461.6
	NAUTY-layered	3.1	5.1	5.9	5.5	20,522.2	–	–	–	–	–	–	–	–
	BLISS-labelvertex	4.8	10.0	3.8	5.1	18.8	22.2	22.1	21.4	35.7	45.6	24.0	43.6	131.8
	NAUTY-labelvertex	4.5	–	4.5	5.6	–	–	–	–	–	–	–	–	–

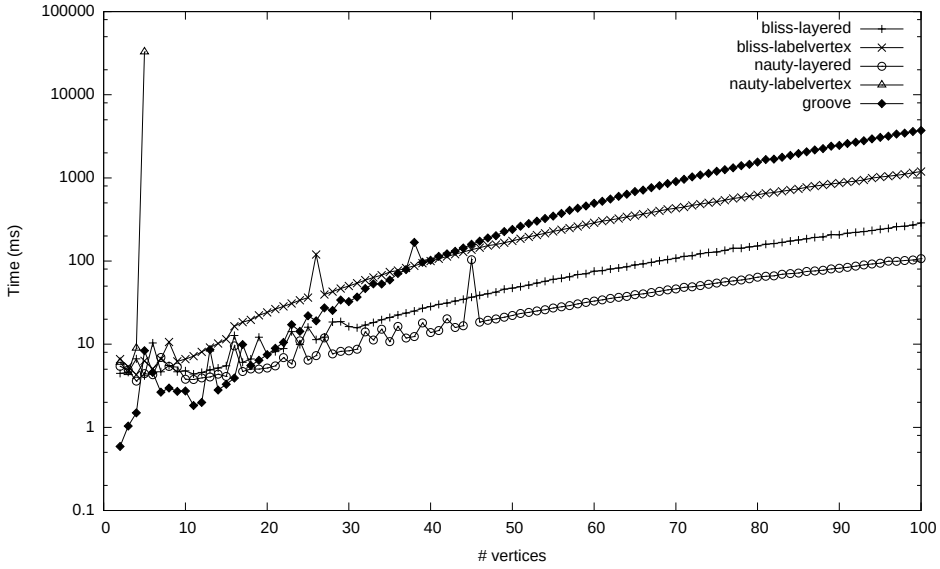


(a) Isomorphic pairs of graphs.

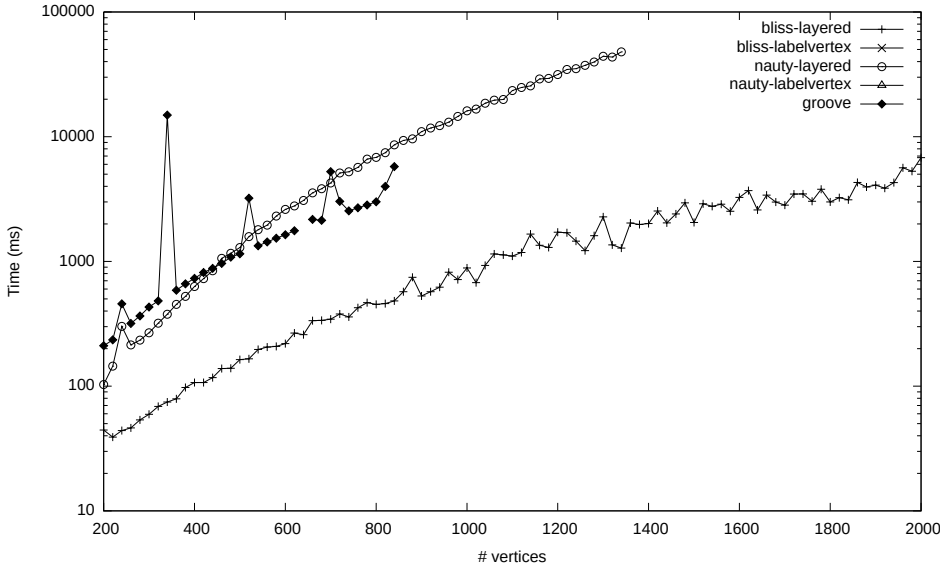


(b) Non-isomorphic pairs of graphs.

Figure 9: Plots for the execution time for pairs of edge-labelled graphs.

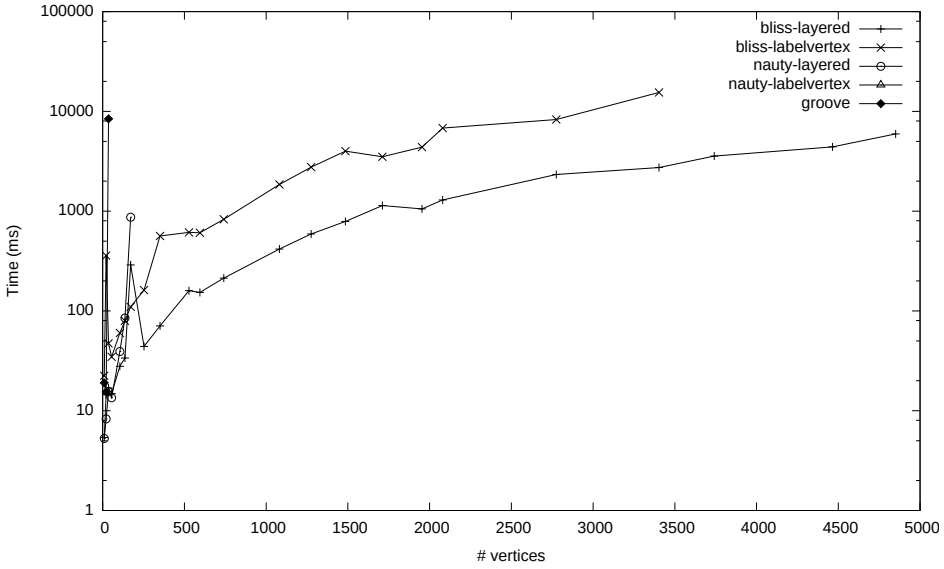


(a) `k-*`.

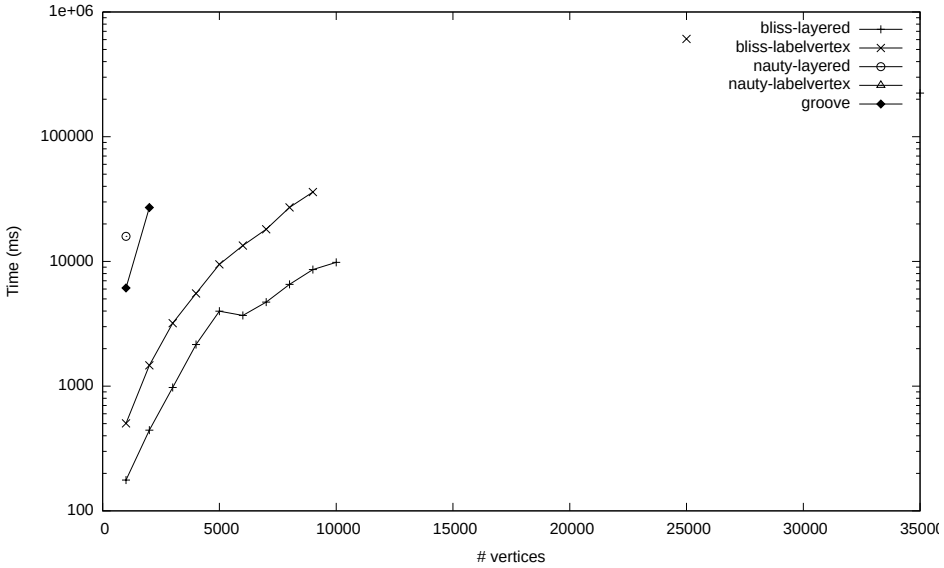


(b) `cfi-*`.

Figure 10: Plots for the execution time for undirected, coloured graphs.



(c) `ag2-*`.



(d) `rnd-reg-*`.

Figure 10: Plots for the execution time for undirected, coloured graphs.

times for the different tools in *ms*. Fig 10 shows the average execution times for each of the sets of undirected coloured graphs. The y-axis has a logarithmic scale.

First, we see that the layered conversion is, as we expected, much better than the label-vertex conversion for these particular series of graphs, due to the absence of distinguishing vertex colours. So we ignore the results for the tools with label-vertex conversion and continue with the results of the tools GROOVE, BLISS-layered and NAUTY-layered. The performance of these tools varies a lot for the different series of graphs. For series `cfi`, `ag2` and `rnd-3-reg` clearly BLISS is the best tool. For `ag2` and `rnd-3-reg` the tools NAUTY and GROOVE are soon unable to give an answer within 5 minutes. For the complete graphs (K_n), `nauty` is the fastest tool when combined with the layered conversion. For small graphs the three tools perform comparably, but for larger graphs `bliss` is about half as fast as NAUTY and the execution times of GROOVE grow more than a constant factor above those of NAUTY and BLISS.

7 Conclusions and Future Work

Isomorphism checking in graph-based model checking is useful for achieving symmetry reduction. Two methods for isomorphism checking are described in this paper: comparing two graphs directly in a one-to-one fashion, and computing canonical forms of the graphs, which enables checking sets of graphs for isomorphism. The BLISS and NAUTY tools are able to compute canonical forms of vertex coloured graphs. Using a conversion function these algorithms can also be applied to edge labelled graphs, the type of graphs that is used in the graph-based model checker GROOVE. In GROOVE an algorithm exists that computes canonical hash codes for edge labelled graphs and an algorithm that checks isomorphism of two graphs based on computing graph certificates (but without computing a canonical form).

The results of our experiments have shown that, contrary to expectations, the state-or-the-art isomorphism checking tools NAUTY and BLISS do not do better than our own ad-hoc implementation in GROOVE for graphs that are used in graph-based model checking. On the other hand, BLISS does appear to scale better to larger graphs, at least in the label-vertex conversion. The better performance of BLISS for large graphs is confirmed by experiments with a collection of large, but undirected, coloured graphs. This is not the kind of graphs we use in model checking, but the experiments do show that there are limits to GROOVE with respect to the size of the graphs that can be dealt with efficiently.

More experimentation and profiling has to be carried out to determine if this is really the case, and if so, to explain the phenomenon: for graphs with little or no actual symmetry, the computational complexity of the GROOVE algorithm should instead be better than that of BLISS.

As future work we have identified the following actions:

- So far we have only compared graphs pairwise, whereas in the context of model checking we are actually interested in finding an isomorphic representative in a set of previously generated graphs. Experiments should be set up to compare the performance of GROOVE and BLISS also in that context.
- We believe that the performance loss in BLISS is mainly due to the need for conversion, which increases the size of the graphs. The canonical form algorithm beneath NAUTY and BLISS, however, can in principle easily be adapted to cope with edge-labelled graphs directly. We intend to carry out this re-implementation to get the best of both worlds.

References

[1] L. Babai and E.M. Luks. Canonical Labeling of Graphs. In *Proc. of the 15th Annual ACM Symposium on Theory of Computing*, pages 171–183. ACM, 1983.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, Mass., 2008.

[3] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance Evaluation of the VF Graph Matching Algortithm. In *Proc. of the International Conference on Image Analysis and Processing*, pages 1172–1177, 1999.

[4] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[5] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[6] P.T. Darga, M. Liffiton, K. Sakallah, and I. Markov. Saucy2: Fast Symmetry Discovery. Online, 2008. `http://vlsicad.eecs.umich.edu/BK/SAUCY/`.

[7] P.T. Darga, K.A. Sakallah, and I.L. Markov. Faster Symmetry Discovery using Sparsity of Symmetries. In *Proc. of the 45th Design Automation Conference*, pages 149–154, 2008.

[8] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[9] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[10] A.H. Ghamarian, M.J. de Mol, A. Rensink, Eduardo Zambon, and M.V. Zimakova. Modelling and Analysis Using GROOVE. Technical Report TR-CTIT-10-18, CTIT, University of Twente, Enschede, 2010.

[11] S. Hsieh, C. Hsu, and L. Hsu. Efficient Method to Perform Isomorphism Testing of Labeled Graphs. *Lecture Notes in Computer Science*, 3984:422–431, 2006.

[12] T. Junttila. bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs. Online, 2008. `http://www.tcs.hut.fi/Software/bliss/`.

[13] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proc. of the 9th Workshop on Algorithm Engineering and Experiments*, pages 135–149. SIAM, 2007.

[14] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1038–1051, 2004.

[15] B.D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[16] B.D. McKay. nauty User's Guide (Version 2.4). Online, 2007. `http://cs.anu.edu.au/~bdm/nauty/`.

[17] B.D. McKay. nauty 2.4. Online, 2008. `http://cs.anu.edu.au/~bdm/nauty/`.

[18] B.T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.

[19] B.T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):307–323, 2000.

[20] T. Miyazaki. The complexity of McKay's canonical labeling algorithm. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 28:239–256, 1997.

[21] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[22] A. Piperno. Search Space Contraction in Canonical Labeling of Graphs. Preliminary version. `http://arxiv.org/abs/0804.4881v1`, 2008.

[23] A. Piperno. Traces 0.01. Online, 2008. `http://www.dsi.uniroma1.it/~piperno/pers/Traces.html`.

[24] A. Rensink. Isomorphism Checking in GROOVE. In *Proc. of the Third International Workshop on Graph Based Tools (GraBaTs 2006)*, volume 1 of *Electronic Communications of the EASST*, 2007.

[25] A. Rensink. Explicit State Model Checking for Graph Grammars. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132, 2008.

[26] A. Rensink. Groove 4.0.1. Online, 2010. `http://groove.sourceforge.net`.

[27] A. Rensink. Isomorphism Checking for Symmetry Reduction. Technical Report TR-CTIT-??-??, CTIT, University of Twente, Enschede, In preparation.

[28] C. Spermann and M. Leuschel. ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In *Proc. of the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'08)*, pages 15–22, 2008.

[29] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry Reduced Model Checking for B. In *Proc. of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)*, pages 25–34, 2007.

[30] J.R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

[31] E.W. Weisstein. Isomorphic Graphs. *MathWorld – A Wolfram Web Resource*, 2009. `http://mathworld.wolfram.com/IsomorphicGraphs.html`.

[32] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proc. of the 2nd IEEE International Conference on Data Mining (ICDM'02)*, 2002.

A Results

Table 3: Performance results for the coloured graphs `k-*`. Time is in *ms*.

Graph	$\|V\|$	$\|E\|$	Bliss layered	Nauty layered	Bliss labelvertex	Nauty labelvertex	Groove
k-2	2	4	4.5	5.4	6.6	6.1	0.6
k-3	3	9	4.6	4.9	5.4	4.6	1.0
k-4	4	16	6.7	3.6	4.3	9.0	1.5
k-5	5	25	4.2	4.4	6.3	32,773.8	8.4
k-6	6	36	10.3	4.3	5.1	–	4.6
k-7	7	49	4.7	6.9	6.6	–	2.6
k-8	8	64	5.5	5.4	10.6	–	3.0
k-9	9	81	4.7	5.3	6.2	–	2.7
k-10	10	100	4.8	3.8	6.6	–	2.7
k-11	11	121	4.4	3.8	7.2	–	1.8
k-12	12	144	4.6	3.9	8.1	–	2.0
k-13	13	169	4.9	4.0	9.1	–	8.6
k-14	14	196	5.2	4.4	10.2	–	2.8
k-15	15	225	5.5	4.1	11.5	–	3.3
k-16	16	256	12.7	9.5	16.4	–	3.9
k-17	17	289	6.1	4.7	18.4	–	9.9
k-18	18	324	6.6	5.0	19.6	–	5.6
k-19	19	361	12.1	5.0	22.2	–	6.4
k-20	20	400	7.5	5.2	24.1	–	7.5
k-21	21	441	8.2	5.5	26.4	–	8.9
k-22	22	484	8.9	6.9	28.5	–	10.5
k-23	23	529	14.2	5.8	30.8	–	17.2
k-24	24	576	9.9	11.0	33.8	–	14.3
k-25	25	625	16.0	6.4	36.4	–	22.0
k-26	26	676	11.4	7.3	119.8	–	19.1
k-27	27	729	12.2	11.9	39.6	–	27.4
k-28	28	784	18.5	7.6	43.0	–	25.5
k-29	29	841	18.7	8.2	46.7	–	34.0
k-30	30	900	16.3	8.3	50.1	–	32.6
k-31	31	961	15.8	8.7	54.2	–	36.9
k-32	32	1,024	17.0	14.1	58.9	–	46.7
k-33	33	1,089	18.2	11.2	63.6	–	53.1
k-34	34	1,156	19.7	15.2	67.6	–	53.0
k-35	35	1,225	20.9	10.7	73.9	–	59.1
k-36	36	1,296	22.5	16.4	76.8	–	70.8
k-37	37	1,369	23.6	11.9	82.4	–	78.8
k-38	38	1,444	25.0	12.4	87.4	–	167.5
k-39	39	1,521	27.1	18.0	94.1	–	96.7
k-40	40	1,600	28.4	13.8	100.1	–	102.4
k-41	41	1,681	30.1	14.6	106.4	–	113.7
k-42	42	1,764	31.3	20.2	113.5	–	122.3
k-43	43	1,849	33.1	15.9	120.9	–	131.0
k-44	44	1,936	34.7	16.7	128.5	–	143.5
k-45	45	2,025	36.8	103.6	137.1	–	159.2
k-46	46	2,116	38.7	18.4	144.7	–	173.3
k-47	47	2,209	40.5	19.4	153.7	–	189.4
k-48	48	2,304	42.3	20.2	158.2	–	201.5
k-49	49	2,401	45.8	21.1	165.6	–	226.4
k-50	50	2,500	47.3	22.2	175.0	–	239.7
k-51	51	2,601	49.2	23.4	185.0	–	261.9
k-52	52	2,704	51.4	24.1	195.1	–	282.3

continued on next page

Graph	$\|V\|$	$\|E\|$	Bliss layered	Nauty layered	Bliss labelvertex	Nauty labelvertex	Groove
k-53	53	2,809	54.3	25.1	205.8	–	301.7
k-54	54	2,916	57.0	26.0	215.9	–	325.5
k-55	55	3,025	60.3	27.3	227.4	–	347.4
k-56	56	3,136	62.2	28.2	238.5	–	373.5
k-57	57	3,249	64.8	29.2	248.6	–	406.8
k-58	58	3,364	68.8	30.6	260.0	–	432.4
k-59	59	3,481	70.3	31.8	273.7	–	459.7
k-60	60	3,600	75.4	33.0	288.6	–	495.5
k-61	61	3,721	76.7	34.2	301.9	–	523.0
k-62	62	3,844	80.0	35.6	312.4	–	556.6
k-63	63	3,969	82.6	36.7	325.3	–	599.3
k-64	64	4,096	85.5	37.6	340.5	–	635.8
k-65	65	4,225	89.9	39.5	352.6	–	684.7
k-66	66	4,356	92.7	40.3	366.6	–	714.7
k-67	67	4,489	96.2	41.9	383.1	–	763.4
k-68	68	4,624	100.8	43.3	400.5	–	805.6
k-69	69	4,761	104.2	45.1	418.5	–	856.5
k-70	70	4,900	108.0	46.2	431.3	–	903.1
k-71	71	5,041	113.8	48.2	446.9	–	960.8
k-72	72	5,184	115.9	49.0	466.7	–	1,027.3
k-73	73	5,329	122.3	50.6	484.9	–	1,082.5
k-74	74	5,476	126.7	52.7	502.0	–	1,134.1
k-75	75	5,625	128.3	54.4	517.2	–	1,200.0
k-76	76	5,776	134.5	56.2	538.4	–	1,254.2
k-77	77	5,929	142.8	57.8	566.2	–	1,321.4
k-78	78	6,084	142.8	59.3	582.5	–	1,401.3
k-79	79	6,241	147.9	61.1	601.9	–	1,455.6
k-80	80	6,400	151.9	63.9	626.8	–	1,549.3
k-81	81	6,561	159.4	65.6	652.9	–	1,660.5
k-82	82	6,724	162.1	66.4	667.1	–	1,682.2
k-83	83	6,889	167.9	69.4	689.8	–	1,771.8
k-84	84	7,056	173.8	70.7	715.7	–	1,867.4
k-85	85	7,225	179.6	71.7	738.0	–	1,963.6
k-86	86	7,396	185.2	74.9	771.6	–	2,058.8
k-87	87	7,569	192.2	76.1	798.0	–	2,165.6
k-88	88	7,744	195.4	77.5	821.6	–	2,250.2
k-89	89	7,921	207.4	80.5	844.8	–	2,404.3
k-90	90	8,100	207.0	81.8	868.1	–	2,464.2
k-91	91	8,281	216.2	84.2	893.2	–	2,591.4
k-92	92	8,464	221.0	87.0	914.8	–	2,691.4
k-93	93	8,649	226.0	89.9	946.7	–	2,803.8
k-94	94	8,836	233.3	92.1	994.1	–	2,942.6
k-95	95	9,025	241.1	94.4	1,025.7	–	3,067.5
k-96	96	9,216	246.9	99.8	1,049.4	–	3,168.5
k-97	97	9,409	259.3	99.4	1,077.0	–	3,365.1
k-98	98	9,604	262.2	101.5	1,109.0	–	3,451.6
k-99	99	9,801	271.3	103.1	1,155.6	–	3,606.7
k-100	100	10,000	287.7	106.8	1,192.5	–	3,718.9

Table 4: Performance results for the coloured graphs `cfi-*`. Time is in *ms*.

Graph	$\|V\|$	$\|E\|$	Bliss layered	Nauty layered	Bliss labelvertex	Nauty labelvertex	Groove
cfi-20	200	800	44.5	103.1	–	–	210.9
cfi-22	220	880	39.1	144.7	–	–	234.9
cfi-24	240	960	44.0	301.2	–	–	457.0
cfi-26	260	1,040	46.3	214.3	–	–	318.2
cfi-28	280	1,120	53.7	233.9	–	–	365.1
cfi-30	300	1,200	59.5	267.6	–	–	430.5
cfi-32	320	1,280	68.7	319.8	–	–	482.8
cfi-34	340	1,360	74.6	376.8	–	–	14,913.4
cfi-36	360	1,440	79.1	452.8	–	–	587.7
cfi-38	380	1,520	97.6	524.7	–	–	660.5
cfi-40	400	1,600	107.0	629.3	–	–	731.0
cfi-42	420	1,680	106.8	728.5	–	–	814.5
cfi-44	440	1,760	117.0	842.2	–	–	879.9
cfi-46	460	1,840	138.6	1,058.8	–	–	962.6
cfi-48	480	1,920	139.1	1,161.5	–	–	1,082.4
cfi-50	500	2,000	163.4	1,290.7	–	–	1,152.1
cfi-52	520	2,080	165.8	1,581.5	–	–	3,215.8
cfi-54	540	2,160	197.0	1,798.9	–	–	1,332.3
cfi-56	560	2,240	205.8	1,955.5	–	–	1,434.3
cfi-58	580	2,320	208.5	2,307.3	–	–	1,534.5
cfi-60	600	2,400	219.4	2,611.0	–	–	1,636.8
cfi-62	620	2,480	266.7	2,787.8	–	–	1,760.3
cfi-64	640	2,560	258.6	3,081.3	–	–	–
cfi-66	660	2,640	334.7	3,546.2	–	–	2,171.7
cfi-68	680	2,720	336.9	3,828.8	–	–	2,135.0
cfi-70	700	2,800	344.6	4,252.5	–	–	5,250.3
cfi-72	720	2,880	379.2	5,113.9	–	–	3,030.2
cfi-74	740	2,960	359.3	5,230.6	–	–	2,541.0
cfi-76	760	3,040	425.3	5,678.4	–	–	2,692.1
cfi-78	780	3,120	467.5	6,614.5	–	–	2,837.3
cfi-80	800	3,200	453.0	6,822.6	–	–	3,010.1
cfi-82	820	3,280	459.8	7,448.5	–	–	3,990.3
cfi-84	840	3,360	482.8	8,599.0	–	–	5,764.2
cfi-86	860	3,440	571.1	9,330.3	–	–	–
cfi-88	880	3,520	746.5	9,615.2	–	–	–
cfi-90	900	3,600	528.0	10,984.8	–	–	–
cfi-92	920	3,680	571.2	11,727.4	–	–	–
cfi-94	940	3,760	621.8	12,271.8	–	–	–
cfi-96	960	3,840	819.1	13,081.3	–	–	–
cfi-98	980	3,920	714.7	14,544.0	–	–	–
cfi-100	1,000	4,000	885.8	16,132.4	–	–	–
cfi-102	1,020	4,080	676.2	16,663.9	–	–	–
cfi-104	1,040	4,160	929.4	18,574.4	–	–	–
cfi-106	1,060	4,240	1,146.5	19,645.1	–	–	–
cfi-108	1,080	4,320	1,129.4	19,879.8	–	–	–
cfi-110	1,100	4,400	1,103.3	23,418.5	–	–	–
cfi-112	1,120	4,480	1,179.4	24,830.6	–	–	–
cfi-114	1,140	4,560	1,658.2	25,528.1	–	–	–
cfi-116	1,160	4,640	1,345.7	29,071.6	–	–	–
cfi-118	1,180	4,720	1,294.1	29,302.4	–	–	–
cfi-120	1,200	4,800	1,713.6	31,473.7	–	–	–
cfi-122	1,220	4,880	1,699.3	34,623.9	–	–	–
cfi-124	1,240	4,960	1,455.4	35,097.1	–	–	–

continued on next page

Graph	$\|V\|$	$\|E\|$	Bliss layered	Nauty layered	Bliss labelvertex	Nauty labelvertex	Groove
cfi-126	1,260	5,040	1,220.8	37,299.6	–	–	–
cfi-128	1,280	5,120	1,609.1	39,705.8	–	–	–
cfi-130	1,300	5,200	2,274.1	44,109.0	–	–	–
cfi-132	1,320	5,280	1,357.6	43,487.8	–	–	–
cfi-134	1,340	5,360	1,278.5	47,840.2	–	–	–
cfi-136	1,360	5,440	2,033.6	–	–	–	–
cfi-138	1,380	5,520	1,979.3	–	–	–	–
cfi-140	1,400	5,600	2,013.8	–	–	–	–
cfi-142	1,420	5,680	2,539.5	–	–	–	–
cfi-144	1,440	5,760	2,036.2	–	–	–	–
cfi-146	1,460	5,840	2,403.5	–	–	–	–
cfi-148	1,480	5,920	2,947.1	–	–	–	–
cfi-150	1,500	6,000	2,051.8	–	–	–	–
cfi-152	1,520	6,080	2,900.9	–	–	–	–
cfi-154	1,540	6,160	2,764.6	–	–	–	–
cfi-156	1,560	6,240	2,875.1	–	–	–	–
cfi-158	1,580	6,320	2,521.5	–	–	–	–
cfi-160	1,600	6,400	3,264.9	–	–	–	–
cfi-162	1,620	6,480	3,695.7	–	–	–	–
cfi-164	1,640	6,560	2,584.6	–	–	–	–
cfi-166	1,660	6,640	3,404.7	–	–	–	–
cfi-168	1,680	6,720	2,990.3	–	–	–	–
cfi-170	1,700	6,800	2,824.0	–	–	–	–
cfi-172	1,720	6,880	3,471.0	–	–	–	–
cfi-174	1,740	6,960	3,479.5	–	–	–	–
cfi-176	1,760	7,040	3,034.7	–	–	–	–
cfi-178	1,780	7,120	3,792.7	–	–	–	–
cfi-180	1,800	7,200	2,998.2	–	–	–	–
cfi-182	1,820	7,280	3,246.2	–	–	–	–
cfi-184	1,840	7,360	3,120.6	–	–	–	–
cfi-186	1,860	7,440	4,294.1	–	–	–	–
cfi-188	1,880	7,520	3,954.4	–	–	–	–
cfi-190	1,900	7,600	4,094.3	–	–	–	–
cfi-192	1,920	7,680	3,861.5	–	–	–	–
cfi-194	1,940	7,760	4,286.4	–	–	–	–
cfi-196	1,960	7,840	5,635.6	–	–	–	–
cfi-198	1,980	7,920	5,278.2	–	–	–	–
cfi-200	2,000	8,000	6,797.4	–	–	–	–

Table 5: Performance results for the coloured graphs `ag2-*`. Time is in *ms*.

Graph	$\|V\|$	$\|E\|$	Bliss layered	Nauty layered	Bliss labelvertex	Nauty labelvertex	Groove
ag2-2	10	34	5.4	5.3	22.4	–	19.0
ag2-3	21	93	14.5	8.3	358.2	–	15.4
ag2-4	36	196	16.9	15.6	47.3	–	8,423.0
ag2-5	55	355	14.9	13.5	34.7	–	–
ag2-7	105	889	27.9	39.3	60.3	–	–
ag2-8	136	1,288	33.7	84.8	79.5	–	–
ag2-9	171	1,791	289.2	870.8	110.2	–	–
ag2-11	253	3,157	44.0	–	161.7	–	–
ag2-13	351	5,083	70.9	–	563.5	–	–
ag2-16	528	9,232	159.8	–	613.8	–	–
ag2-17	595	10,999	153.5	–	607.0	–	–
ag2-19	741	15,181	213.4	–	828.8	–	–
ag2-23	1,081	26,473	417.4	–	1,850.3	–	–
ag2-25	1,275	33,775	590.2	–	2,760.8	–	–
ag2-27	1,485	42,309	789.0	–	4,001.8	–	–
ag2-29	1,711	52,171	1,139.4	–	3,511.2	–	–
ag2-31	1,953	63,457	1,052.6	–	4,389.0	–	–
ag2-32	2,080	69,664	1,294.1	–	6,804.3	–	–
ag2-37	2,775	106,819	2,331.4	–	8,277.4	–	–
ag2-41	3,403	144,607	2,738.6	–	15,478.9	–	–
ag2-43	3,741	166,453	3,562.3	–	–	–	–
ag2-47	4,465	216,529	4,411.6	–	–	–	–
ag2-49	4,851	244,951	5,952.9	–	–	–	–

Table 6: Performance results for the coloured graphs `rnd-3-reg-*`. Time is in *ms*.

Graph	$\|V\|$	$\|E\|$	Bliss layered	Nauty layered	Bliss labelvertex	Nauty labelvertex	Groove
rnd-3-reg-1000-1	1,000	4,000	176.8	15,906.0	503.1	–	6,130.9
rnd-3-reg-2000-1	2,000	8,000	443.2	–	1,467.7	–	27,047.0
rnd-3-reg-3000-1	3,000	12,000	974.8	–	3,200.6	–	–
rnd-3-reg-4000-1	4,000	16,000	2,156.4	–	5,534.1	–	–
rnd-3-reg-5000-1	5,000	20,000	3,989.3	–	9,464.3	–	–
rnd-3-reg-6000-1	6,000	24,000	3,685.2	–	13,395.1	–	–
rnd-3-reg-7000-1	7,000	28,000	4,713.9	–	18,094.9	–	–
rnd-3-reg-8000-1	8,000	32,000	6,545.6	–	27,143.1	–	–
rnd-3-reg-9000-1	9,000	36,000	8,603.8	–	35,988.2	–	–
rnd-3-reg-10000-1	10,000	40,000	9,824.1	–	–	–	–
rnd-3-reg-25000-1	25,000	100,000	–	–	608,412.1	–	–
rnd-3-reg-35000-1	35,000	140,000	223,799.3	–	–	–	–