CPQ Keys: a survey of graph canonization algorithms

Towards a CPQ based graph database index

Roan Hofland
Osaka University, Japan
Eindhoven University of Technology, The Netherlands
r.w.p.hofland@student.tue.nl

17th of July, 2022

Supervisors
George Fletcher (g.h.l.fletcher@tue.nl)
Yuya Sasaki (sasaki@ist.osaka-u.ac.jp)
Seiji Maekawa (maekawa.seiji@ist.osaka-u.ac.jp)

2IMC10 - Internship

Contents

1 Introduction **4**
1.1 Canonization . . . 4
1.2 Algorithms . . . 4

2 Preliminaries **5**
2.1 Conjunctive Path Queries . . . 5
2.2 CPQ Query Graphs . . . 5
2.3 CPQs & Treewidth . . . 6
2.4 Graph Transforms . . . 7
2.4.1 From edge labelled graph to edge unlabelled graph . . . 7
2.4.1.1 Using labelled vertices . . . 7
2.4.1.2 By duplicating the graph . . . 8
2.4.1.3 By duplicating the graph for each colour . . . 8
2.4.2 From directed graph to undirected graph . . . 9
2.4.2.1 Using labelled vertices . . . 9
2.4.2.2 By building arrows on edges . . . 9
2.4.2.3 By constructing a tri-coloured undirected graph . . . 9
2.4.2.4 By duplicating the graph . . . 10
2.4.2.5 Special cases . . . 10
2.4.3 Other Transforms . . . 10
2.4.3.1 Self Loops . . . 10
2.4.3.2 Parallel Edges . . . 11
2.4.3.3 Generalised Solution . . . 11
2.5 gMark Extensions . . . 11
2.5.1 Plain CPQ generation . . . 11
2.5.2 Other Utilities . . . 12
2.6 Evaluation Framework . . . 12
2.6.1 Algorithm Implementations . . . 12
2.6.2 Evaluation Logic . . . 13
2.6.3 Utilities . . . 13

3 Canonization Algorithms **14**
3.1 Nauty . . . 14
3.2 Bliss . . . 14
3.3 Nishe . . . 15
3.4 Scott . . . 15
3.5 Traces . . . 15
3.6 Without existing implementation . . . 16
3.6.1 Graph Rewriting . . . 16
3.6.2 Using Trees . . . 16
3.6.3 Group-theoretic Work . . . 16

4 Performance Evaluation **17**
4.1 Methodology . . . 17
4.2 Datasets . . . 17
4.3 Results . . . 18

5 Concluding Remarks **21**
5.1 Future Work . . . 21
5.1.1 Using only isomorphism testing . . . 21
5.1.2 Output handling . . . 21
5.1.3 Specialised setup logic . . . 22
5.1.4 Using specialised algorithms . . . 22
5.1.5 Custom canonization algorithm . . . 22
5.1.6 Generalised Evaluation Framework . . . 22
5.1.7 Language-aware Index . . . 22

A Evaluation Data **27**
A.1 Bliss 27
A.2 Nauty (dense) 27
A.3 Nauty (sparse) 28
A.4 Nishe 28
A.5 Scott 28
A.6 Traces 29

B Evaluation Framework Setup **29**
B.1 Getting Started 29
B.1.1 Docker 29
B.1.2 From Source 29
B.2 Development 30
B.2.1 Adding Algorithms 30

Version History

Version	Date	Sections	Comment
v1.0	2022-07-12	1, 2, 3, 4, 5 & A	Initial version
v1.1	2022-07-17	1, 2, 3, 4, 5, A & B	Proof read, apply feedback from Mireille & George

1 Introduction

Conjunctive path queries (CPQ) are one of the most frequently used queries for complex graph analysis. However, tailored support for these queries in various domains is lacking. One area where optimisations targeted specifically at CPQs could help is database querying. To accomplish this we can make use of a language-aware graph database index. At its core the idea of an index is that we can use a key to retrieve specific information that can be used to accelerate the evaluation of a query. In the context of a database index the retrieved information will be the evaluation result of a (partial) query and the key(s) used will be derived from the query the user wants to evaluate.

What makes an index language-aware is the fact that it is designed for a specific query language. For this project we will focus on the language of CPQs, which will be formally introduced in Section 2.1. A key insight is that for a given language paths through a graph may exist that cannot be distinguished by any query. A language-aware index will capitalise on this and optimise what is stored in the index. As mentioned, a key is used to access data from the index. The focus of this report lies on part of the process required to compute such an index key, namely, canonization.

This report presents the results of a survey into existing algorithms for canonization and is aimed at finding the most suitable algorithm to use in a language-aware graph database index based on CPQs. The codebase used to evaluate the tested algorithms will be made available as open source[1] and various extensions have been added to the existing open source gMark software[2]. Part of this report is also available as a website[3].

1.1 Canonization

Canonization is the process of finding a representative form of some input graph that is the same for all input graphs that are isomorphically equivalent. Therefore, canonization is closely related to isomorphism testing. Two graphs G_1 and G_2 are isomorphically equivalent when a function $p : G_1 \to G_2$ exists that permutes the vertices of G_2 in such a way that for any two vertices $u, v \in G_1$ it holds that they are adjacent in G_1 if and only if $p(u)$ and $p(v)$ are adjacent in G_2. If this holds true, then the graphs G_1 and G_2 are isomorphically equivalent, denoted by $G_1 \simeq G_2$. In addition we define the concept of an automorphism, which is an isomorphism from a graph to itself.

In general it is still unknown how hard exactly the graph isomorphism problem is [56]. However, it is considered unlikely that the problem is NP-complete, as this would collapse the polynomial-time hierarchy [25]. Unfortunately, this does not imply that the problem is easy. For more than three decades the fastest proven running time for graph isomorphism was $e^{O(\sqrt{n \log n})}$ by Babai et al. [5], where n denotes the number of vertices in the graph. However, recently in 2015 Lászió Babai managed to improve upon this bound and show that the problem can be solved in quasipolynomial time $e^{(\log n)^{O(1)}}$ [4], where n is again the number of vertices.

What canonization allows us to do, is to compute a canonical representative that is equivalent for all isomorphically equivalent graphs. Essentially, this canonical form is a sort of checkpoint that removes the need to directly compare all input graphs. Instead the canonical form of an input graph is computed and this canonical form can be efficiently compared with the canonical forms computed from other graphs. If two canonical forms are then found to be equal, that means that the graphs they were computed from are isomorphically equivalent without having to explicitly check this. However, this does come at a cost, as computing a canonical form is typically slower than directly testing if two graphs are isomorphically equivalent. However, comparing already computed canonical forms, is a very cheap process.

1.2 Algorithms

Since canonization has many applications wealth of literature exists on the topic. For this report a focus will be put on canonization algorithms with an existing implementation. These algorithms will then be evaluated on their suitability to generate canonical representations of CPQ query graphs. The algorithms that will be evaluated are Bliss [39], Nauty [53], Nishe [75], Scott [10], and Traces [61]. These algorithms will be discussed in more detail in Section 3. Some promising theoretical algorithms without existing implementation will also be discussed in this section, but these are not considered for the evaluation.

[1] `https://github.com/RoanH/CPQKeys`
[2] `https://github.com/RoanH/gMark`
[3] `https://cpqkeys.roanh.dev`

2 Preliminaries

In this section the core theory required for the rest of the report will be explained.

2.1 Conjunctive Path Queries

As stated in Sasaki et al. [68] conjunctive path queries (CPQ) are a basic graph query language that covers more than 99% of query shapes that appear in practice. Hence optimisations specifically targeted at CPQs will benefit a large majority of real world use cases. To illustrate how CPQs work we will show how to construct a query to find people who know someone who knows them. However, before that we have to formalise the exact definition of a CPQ.

A conjunctive path query can be recursively constructed from the operations of identity 'id', edge label 'l', inverse edge label 'l^-', join '$\circ$' and conjunction '$\cap$'. This results in the following grammar:

$$CPQ ::= id \mid l \mid l^- \mid CPQ \circ CPQ \mid CPQ \cap CPQ \mid (CPQ)$$

The exact definition of these operations when evaluated on a graph $\mathcal{G}$ with vertex set $\mathcal{V}$, edge label set $\mathcal{L}$ and edge set $\mathcal{E} = \mathcal{V} \times \mathcal{V} \times \mathcal{L}$ is then given as follows:

$$\begin{aligned}
[\![id]\!]_{\mathcal{G}} &= \{(v, v) \mid v \in \mathcal{V}\} \\
[\![l]\!]_{\mathcal{G}} &= \{(v, u) \mid (v, u, l) \in \mathcal{E}\} \\
[\![l^-]\!]_{\mathcal{G}} &= \{(u, v) \mid (v, u, l) \in \mathcal{E}\} \\
[\![q_1 \circ q_2]\!]_{\mathcal{G}} &= \{(v, u) \mid \exists m \in \mathcal{V} : (v, m) \in [\![q_1]\!]_{\mathcal{G}} \wedge (m, u) \in [\![q_2]\!]_{\mathcal{G}}\} \\
[\![q_1 \cap q_2]\!]_{\mathcal{G}} &= \{(v, u) \mid (v, u) \in [\![q_1]\!]_{\mathcal{G}} \wedge (v, u) \in [\![q_2]\!]_{\mathcal{G}}\} \\
[\![(q_1)]\!]_{\mathcal{G}} &= [\![q_1]\!]_{\mathcal{G}}
\end{aligned}$$

This means that the result of evaluating a CPQ is a set of vertex pairs. More information about CPQs and their properties can be found in the paper by Sasaki et al. [68].

Given the formal definitions we now show how to answer our original question. Note that finding people you know who also know you is equivalent to finding a path consisting of 2 knows edges that starts and ends at the same node. We can write the following CPQ for this $(\texttt{knows} \circ \texttt{knows}) \cap id$, where we add the conjunction with identity to ensure we only return results that end and start at the same node. To make the example more concrete we show a simple social network in Figure 1. Evaluating the suggested CPQ then gives the following results $[\![(\texttt{knows} \circ \texttt{knows}) \cap id]\!]_{\mathcal{G}} = \{(\text{Alice}, \text{Alice}), (\text{Bob}, \text{Bob})\}$.

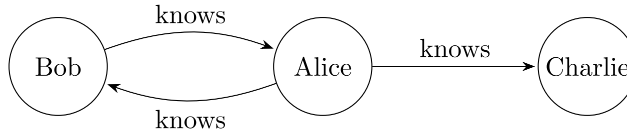


Figure 1: A simple social network graph $\mathcal{G}$.

2.2 CPQ Query Graphs

Since CPQ is a graph querying language, we can visualise the graph structure matched by a CPQ as a graph. We call this graph the query graph of a CPQ. A query graph for a CPQ q is defined as $G_q = (V_q, E_q, L, v_s, v_t)$. Here V_q and E_q respectively represent the vertices and the edges in the query graph. The set L represents the edge labels appearing in the CPQ, and v_s and v_t are the source and target vertex of the CPQ.

Next we will show how to recursively construct the query graph of a CPQ using the equations below. The method shown here is an updated version of an approach originally proposed by Seiji Maekawa in internal notes. First we extend the definition of a query graph with an extra set F_{id} that keeps track of distinct vertices that should eventually be merged to become the same vertex. The query graph construction works in three main steps. First we have 3 base case definitions of $f_{q2graph}$ that define a query graph for identity, edge label and inverse edge label. Second we have 3 recursive definitions for the join, conjunction and bracket operations that compute the query graph of smaller parts of the input query and then merge the results. Finally, we have a recursive merge step f_{merge} that cleans up the final graph by merging marked pairs of identity vertices in F_{id}. After all identity pairs have been merged the set F_{id} is no longer required.

$$
\begin{aligned}
f_{q2graph}(id, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) &= (V_f, E_f, L, v_s, v_t, F_{id} \cup \{(v, u)\}) \\
f_{q2graph}(l, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) &= (V_f, E_f \cup \{v, u, l\}, L, v_s, v_t, F_{id}) \\
f_{q2graph}(l^-, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) &= (V_f, E_f \cup \{u, v, l\}, L, v_s, v_t, F_{id}) \\
f_{q2graph}(q_1 \circ q_2, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) &= f_{q2graph}(q_1, v, m, (V_f \cup \{m\}, E_f, L, v_s, v_t, F_{id})) \\
&\quad \cup_G f_{q2graph}(q_2, m, u, (V_f \cup \{m\}, E_f, L, v_s, v_t, F_{id})) \\
f_{q2graph}(q_1 \cap q_2, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) &= f_{q2graph}(q_1, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) \\
&\quad \cup_G f_{q2graph}(q_2, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) \\
f_{q2graph}((q_1), v, u, (V_f, E_f, L, v_s, v_t, F_{id})) &= f_{q2graph}(q_1, v, u, (V_f, E_f, L, v_s, v_t, F_{id})) \\
f_{merge}(V_f, E_f, L, v_s, v_t, F_{id}) \text{ if } (F_{id} \neq \emptyset) &= \begin{cases}
\{\text{Let } (v_{id}, u_{id}) \text{ be an element in } F_{id}\} \\
f_{merge}(\\
\quad V_f \setminus \{v_{id}\}, \\
\quad E \cup \{(u_{id}, u, l) \mid (v, u, l) \in E_f \wedge v = v_{id}\} \\
\qquad \cup \{(v, u_{id}, l) \mid (v, u, l) \in E_f \wedge u = v_{id}\} \\
\qquad \cup \{(u_{id}, u_{id}, l) \mid (v, u, l) \in E_f \wedge v = v_{id} \wedge u = v_{id}\} \\
\qquad \setminus \{(v, u, l) \mid (v, u, l) \in E_f \wedge (v = v_{id} \vee u = v_{id})\}, \\
\quad L, \\
\quad v_s \text{ if } v_s \neq v_{id} \text{ otherwise } u_{id}, \\
\quad v_t \text{ if } v_t \neq v_{id} \text{ otherwise } u_{id}, \\
\quad F_{id} \cup \{(u_{id}, u) \mid (v, u) \in F_{id} \wedge v = v_{id}\} \\
\qquad \cup \{(v, u_{id}) \mid (v, u) \in F_{id} \wedge u = v_{id}\} \\
\qquad \setminus \{(v, u) \mid (v, u) \in F_{id} \wedge (v = v_{id} \vee u = v_{id})\} \\
)
\end{cases} \\
f_{merge}(V_f, E_f, L, v_s, v_t, F_{id}) \text{ if } (F_{id} = \emptyset) &= (V_f, E_f, L, v_s, v_t)
\end{aligned}
$$

In these equations we introduce a special operator $\cup_G$ to denote taking the union of two query graphs, which we define as follows $(V_1, E_1, L, v_s, v_t, F_1) \cup_G (V_2, E_2, L, v_s, v_t, F_2) = (V_1 \cup V_2, E_1 \cup E_2, L, v_s, v_t, F_1 \cup F_2)$. Finally, to use these functions to construct the query graph of CPQ q with source vertex v_s, target vertex v_t and label set L, the following should be computed $G_q = f_{merge}(f_{q2graph}(q, v_s, v_t, (\{v_s, v_t\}, \emptyset, L, v_s, v_t, \emptyset)))$.

2.3 CPQs & Treewidth

Treewidth is a popular metric to indicate the difficulty of a graph for various algorithmic tasks. Essentially it is a metric to describe how far removed from a tree a given graph is, as many problems are easy or trivial to solve on a tree. By definition trees have a treewidth of 1.

Treewidth has not always been such a popular metric and has been rediscovered many times over the years under different names. Originally it was introduced as the *dimension* of a graph by Umberto Bertelè and Francesco Brioschi in 1972 [7]. Then later in 1976 it was reintroduced by Rudolf Halin in his study of S-functions [28]. Finally, it was introduced under its current name of treewidth in 1984 by Neil Robertson and Paul Seymour [66]. Treewidth was later also generalised to directed graphs [34].

However, there are still many alternative definitions that were later discovered to be equivalent to treewidth. Some notable definitions include the minimum width of a tree decomposition, the minimal k for which a graph is a partial k-tree, the minimum clique number of a chordal supergraph, and the minimum cost of an elimination order. A thorough overview of these and more definitions and why they are equivalent is given in a survey paper by Hans Bodlaender [13]. One more notable equivalence is that graphs with treewidth at most k are k-degenerate, which implies $|E| \leq k \cdot |V|$ for a graph with vertex set V and edge set E [27]. In particular, since CPQs have treewidth 2 we get $|E| \leq 2 \cdot |V|$.

The concept of treewidth is very relevant to CPQs as CPQs have a very low treewidth of 2. This means that for example our core problem of isomorphism testing, although hard in general, may admit a much more efficient solution on CPQ query graphs. For example, fixed parameter tractable algorithms with a runtime exponential in the treewidth could be viable solutions to problems involving CPQs. In addition to this, graphs of treewidth 2 are known as series-parallel graphs and series-parallel graphs are known to be planar by Kuratowski's theorem [42, 12]. Isomorphism testing on planar graphs is also known to be in log space [21].

2.4 Graph Transforms

To evaluate various algorithms on their ability to compute canonical forms of CPQ query graphs we first need to make sure these algorithms can take a CPQ query graph as input. For many existing canonization algorithms, both theoretical and those with an existing implementation, this is not the case. CPQ query graphs are directed multigraphs with edge labels. In particular parallel edges and labelled edges are not often supported by existing algorithms. Therefore, in order to be able to use these algorithms anyway, we need to transform the CPQ query graph to a supported input format. For our use case it is important that the isomorphism relation between graphs is preserved. Formally, if G_1 and G_2 are graphs and $f : G \to G'$ is our transformation function, then G_1 and G_2 are isomorphically equivalent if and only if $f(G_1)$ is isomorphically equivalent to $f(G_2)$. In this section we will discussion various methods that can be used to accomplish this.

For the main algorithms we intend to evaluate in Section 4 we will first give an overview of the types of input they accept in Table 1.

Algorithm	Edge Labels	Vertex Labels	Directed Graph	Self Loops	Parallel Edges
Bliss	No	Colour	Yes	Yes	No
Nauty	No	Colour	Yes	Yes	No
Nishe	No	Colour	Yes	Yes	No
Scott	Yes	Yes	Partial	Yes	Yes
Traces	No	Colour	No	Yes	No

Table 1: Overview of canonization algorithms and their supported input.

Most algorithms support vertex labels in some form. However, it is worth noting that support for arbitrary vertex labels is often missing. Instead most algorithms instead offer support for a coloured graph. Here the graph is divided up into subsets of vertices that have the same colour. What makes this different from explicit vertex labels is that while these colours are distinct, they are also interchangeable.

For example, suppose we have two copies of the same graph with three nodes. In one of these graphs two nodes are 'blue' and the last one 'red', while in the other graph two nodes are 'red' and the last one 'blue'. These graphs would be equivalent as coloured graphs as the labels of 'blue' and 'red' we assign to these subsets do not actually exist. Instead we just have nameless sets of nodes.

It is worth noting that there are ways to augment a transformed graph further to make these colour sets distinguishable. The general idea would be to attach nodes with the same label to a uniquely identifiable structure of that colour instead. Some more information about this can be found in the manual for Nauty and Traces [55]. An explanation for the partial support for directed graphs for Scott will be given in Section 3.4.

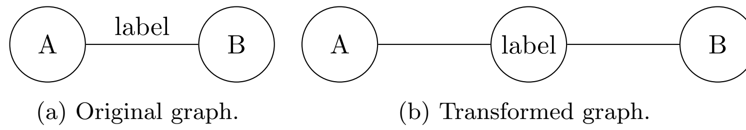
2.4.1 From edge labelled graph to edge unlabelled graph

In this section we will give an overview of considered methods to turn an edge labelled graph into a graph without edge labels. Section 2.4.1.1 will introduce the transform actually used for the algorithm evaluation.

2.4.1.1 Using labelled vertices The simplest way to turn a graph with edge labels into a graph without edge labels is to move the edge labels from the edges to newly created vertices. This method has also been suggested on the mailing list for Nauty by the original creator of Nauty [52]. Formally, given an edge labelled graph $G_1 = (V_1, E_1, L_1)$, where the edge set is defined as $E_1 \subseteq V_1 \times V_1 \times L_1$. Then we can create a transformed graph $G_2 = (V_2, E_2)$, where the edge set is defined as $E_1 \subseteq V_1 \times V_1$ as follows. The vertex set is copied as is $V_2 = V_1$. Then for each edge $(v, u, l) \in E_1$ we do the following. We first add a new vertex m with label l to V_2 and then add the edges (v, m) and (m, u) to E_2. Since this transform introduces a new vertex for each edge in the original graph and also splits each original edge into two edges, we can say the following about the size of the resulting graph. If an input graph has $|V|$ vertices and $|E|$ edges, then the transformed graph has $|V| + |E|$ vertices and $2 \cdot |E|$ edges. An example for this transform is shown in Figure 2.

Note that it does not matter if the graph was using vertex labels already. However, if the graph was using vertex labels before, then care should be taken that former edge labels are not also valid vertex labels. One way of doing this would be to add a prefix to both former edge and vertex labels to indicate their origin. It is also possible to use colours instead of edge labels.

Finally, because CPQs have a label on every edge, an added benefit of this transform is that all parallel edges and self loops get transformed such that the output is a simple graph.



(a) Original graph. (b) Transformed graph.

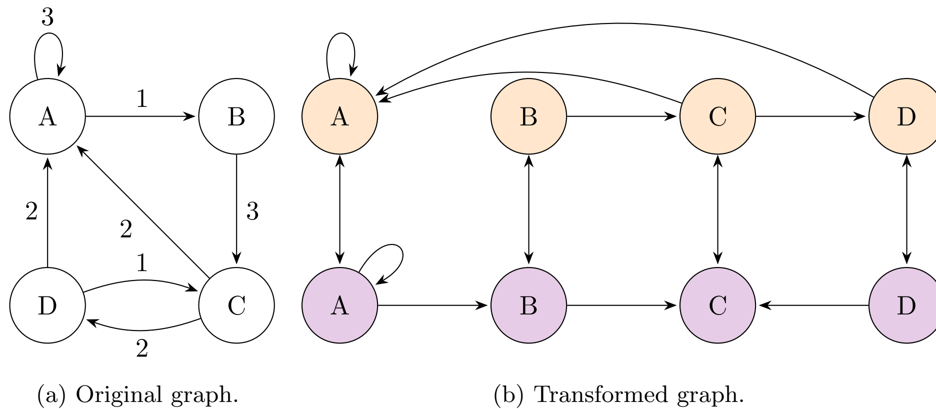
Figure 2: Moving edge labels to vertices.

2.4.1.2 By duplicating the graph The manual for Nauty & Traces [55] suggests an efficient isomorphism preserving method for transforming an edge labelled graph to an edge unlabelled graph by making a few linked copies of the graph and adding edges from the original graph only to specific layers in the new graph. After fully transforming the graph in this way we obtain a layered graph where the canonical form of the first layer is a canonical representation for the original input graph.

Unfortunately, this method does not support parallel edges in the input graph. This means the transform cannot be used directly on CPQ query graphs. This transform also introduces vertex colours to the transformed graph as each layer has a unique colour.

For completeness the core details of the transform will be explained here, but a full explanation can be found in the aforementioned Nauty & Traces manual. The figure from the manual is reproduced here as Figure 3 for clarity. The first step is to sequentially associate a numerical ID with each edge label. Next we determine the number of layers by picking the smallest value of d for which $2^d - 1$ is equal or larger to the highest edge label ID. Having computed the number of layers d we populate each of these layers with the same number of nodes as originally present in the original graph, all the nodes in the same layer have the same colour. In addition, the nodes representing the same vertex from the original graph in each layer are connected with a bidirectional edge. Finally, we add unlabelled directed edges to specific layers depending on the binary expansion of their ID. For example, an edge with a label with ID 3 would go in the first and second layer as the binary form of 3 is 11, meaning the second and first bit are set. Similarly, an edge with a label with ID 2 would only go in the second layer as the binary form of 2 is 10, which only has the second bit set. This completes the construction of the transformed graph.

Apart from the mentioned limitations, this approach is fairly efficient using only $O(|V| \cdot \log k)$ vertices, where $|V|$ is the number of vertices in the original graph and k the number of edge labels. The number of edges is similarly bounded by $O(|E| \cdot \log k)$.



(a) Original graph. (b) Transformed graph.

Figure 3: Moving edge labels to graph layers.

2.4.1.3 By duplicating the graph for each colour The idea proposed in Section 2.4.1.2 can be reformulated as a more naive idea as also suggested on the mailing list for Nauty [72]. Instead of trying to keep the number of vertices low we simply make a copy of the graph for each edge label. So if we have k edge labels we make k copies of the graph, each of these copies containing only edges that originally had the same label and each copy representing a different label. Essentially each edge label gets a graph of its own. Finally, we add a new vertex for each original graph vertex that is connected to all the vertices in each of the copy graphs that represent it.

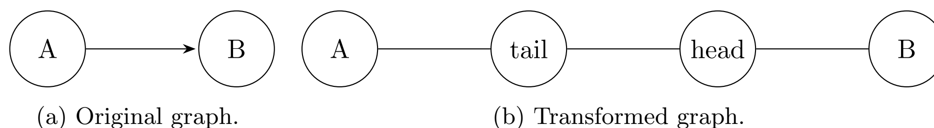
Although being applicable to any input graph, this transform is not particularly efficient and is extremely unfit for large sparse graphs with many different edge labels. If an input graph has $|V|$ vertices, $|E|$ edges, and k colours, then the transformed graph has $(k+1) \cdot |V|$ vertices and $|E| + k \cdot |V|$ edges.

2.4.2 From directed graph to undirected graph

In this section we will give an overview of considered methods to turn a directed graph into an undirected graph. Section 2.4.2.1 will introduce the transform actually used for the algorithm evaluation.

2.4.2.1 Using labelled vertices Similar to the approach discussed in Section 2.4.2.1 we can turn a directed graph into an undirected vertex labelled graph by replacing every directed edge with a sequence of new edges and vertices that encodes that direction of the original edge. This involves breaking every existing directed edge into 3 new undirected edges that are connected together in a chain with two new vertices labelled 'head' and 'tail' indicating the direction of the old directed edge. Formally, given a directed graph $G_1 = (V_1, E_1)$ with edge set $E_1 \subseteq V_1 \times V_1$ we create the undirected transformed graph $G_2 = (V_2, E_2)$ with edge set $E_2 = V_2 \times V_2$ as follows. First the vertex set is copied as is $V_2 = V_1$. Then for every directed edge $(v, u) \in E_1$ we do the following. First we add two new nodes, n_1 with label 'tail' and n_2 with label 'head' to V_2. Second we add the following 3 undirected edges (v, n_1), (n_1, n_2), and (n_2, u) to E_2. Since this transform introduces two new vertices for each edge in the original graph and also splits each original edge into three edges, we can say the following about the size of the resulting graph. If an input graph has $|V|$ vertices and $|E|$ edges, then the transformed graph has $|V| + 2 \cdot |E|$ vertices and $3 \cdot |E|$ edges. An example of the transformation is shown in Figure 4.

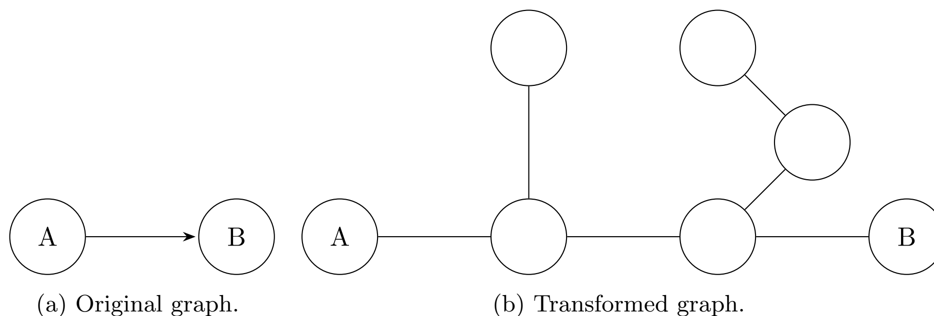
Note that it does not matter if the original graph was already using vertex labels. However, care should be taken that the special 'head' and 'tail' labels are distinct from any existing vertex labels. Note that it is also possible to use colours instead of labels. If the original graph was using edge labels then we can slightly extend the approach to accommodate them. If an original directed edge has an edge label then we simply copy this label to the middle of the three newly added edges (the one between n_1 and n_2). The reason we do not copy the label to all the newly added edges is to prevent the graph size from increasing more than required when also running the edge label transform from Section 2.4.2.1 after this undirected transform.



(a) Original graph. (b) Transformed graph.

Figure 4: Turning directed edges into vertices.

2.4.2.2 By building arrows on edges Building on the approach in Section 2.4.2.1 we can modify it to also work without introducing vertex labels [65]. Instead of marking the head and tail nodes with a label we can mark them using some additional graph structure. To do this we will attach a newly added vertex with an edge to every tail node and add a chain with two vertices to every head node. When properly arranging the nodes this can be seen as building an arrow with undirected edges where the original directed edge was. The relationship between the size of the original graph and the transformed graph can be given as follows. If an input graph has $|V|$ vertices and $|E|$ edges, then the transformed graph has $|V| + 5 \cdot |E|$ vertices and $6 \cdot |E|$ edges. An example of the transform is given in Figure 5.

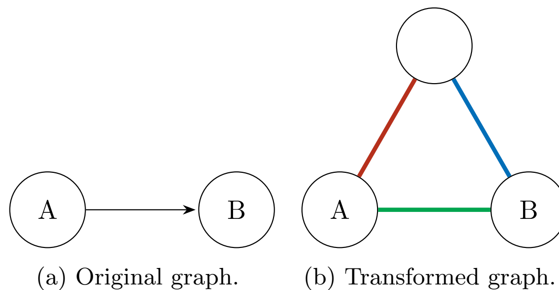


(a) Original graph. (b) Transformed graph.

Figure 5: Turning directed edges into arrows.

2.4.2.3 By constructing a tri-coloured undirected graph It is possible to transform a directed graph into an undirected graph by introducing edge colours [72, 48]. The general idea here is to turn all existing edges into undirected edges with the same colour, e.g. green. In parallel with each original edge we then place a newly added vertex with a differently coloured edge to the original source and target vertex of the edge, e.g. a red edge to the original source vertex and a blue edge to the original target vertex. This way the graph becomes undirected while all the relevant information is retained. A disadvantage is the introduction of edge colours

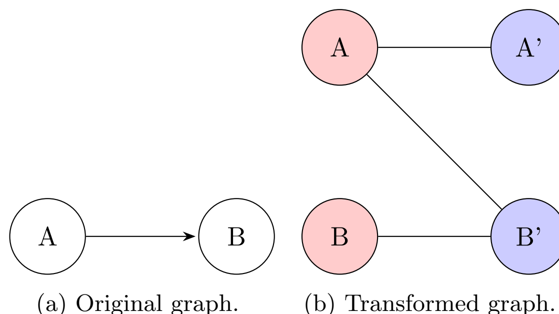
or labels into the graph. Since this transform introduces one new vertex and 2 new edges for each edge in the original graph, we can say the following about the size of the resulting graph. If an input graph has $|V|$ vertices and $|E|$ edges, then the transformed graph has $|V|+|E|$ vertices and $3 \cdot |E|$ edges. An example of this transform is shown in Figure 6.



(a) Original graph. (b) Transformed graph.

Figure 6: Turning directed edges into tri-coloured subgraphs.

2.4.2.4 By duplicating the graph We can make a directed graph undirected by constructing a bipartite vertex coloured graph [73, 22]. The idea is to construct a bipartite vertex coloured graph with all the edge tails in one set and all the edge heads in the other set. Formally, given a directed graph $G_1 = (V_1, E_1)$ we construct the transformed graph $G_2 = (V_2, E_2)$ as follows. For each vertex $v \in V_1$ add vertices v and v' to V_2 and edge (v, v') to E_2. Then for each directed edge $(a, b) \in E_1$ add edge (a, b') to E_2. All the v vertices get the same colour and all the v' vertices get the same colour. Since this approach effectively duplicates the graph we can express the size of the transformed graph as follows. If an input graph has $|V|$ vertices and $|E|$ edges, then the transformed graph has $2 \cdot |V|$ vertices and $|V| + |E|$ edges. An example of this transform is shown in Figure 7.



(a) Original graph. (b) Transformed graph.

Figure 7: Turning directed edges into bipartite graphs.

2.4.2.5 Special cases Besides generalised methods to transform a graph into an undirected graph as discussed in the previous sections, there are also classes of graphs that admit a trivial conversion to an undirected graph as noted by Brendan McKay on the Nauty mailing list [49]. For example, if the directed graph is a tree with all edges directed away from some root node, then it suffices to give this root note a distinct colour and to make all edges undirected. A similar approach is possible when the graph is bipartite and all edges point from one of the partitions to the other. In this case all the nodes in each partition can simply be given the same colour, while ensuring that distinct colours are used for both partitions.

2.4.3 Other Transforms

As shown in Table 1 in Section 2.4, native support for various graph patterns can be lacking. In particular, to canonize CPQ query graphs we require algorithms to be able to handle both self loops and parallel edges in addition to edge labels and directed graphs. Handling edge labels and vertex labels is already detailed extensively in Section 2.4.1 and Section 2.4.2 respectively. However, it is useful to touch briefly on self loops and parallel edges.

2.4.3.1 Self Loops Although none of the algorithms planned for evaluation lack support for self loops, this was not the case for all considered algorithms, like some of the algorithms that will be discussed in Section 3.6. Due to the nature of the transform we will use to deal with edge labels introduced in Section 2.4.1.1, all self loops will naturally disappear as they will be split into two edges joined by a new vertex. In general, this solution could also be applied only on self loops, by splitting them with a specially coloured or labelled vertex.

2.4.3.2 Parallel Edges Note again that the issue with parallel edges is automatically resolved due to the transform from Section 2.4.1.1 we will use to deal with edge labels. After all, if two vertices contain two edges between them, then both edges are broken up into two edges joined by a newly added vertex. Similar to the case with self loops, we could also only transform parallel edges like this, again introducing a specially coloured or labelled vertex to keep them distinguishable from normal parts of the graph.

2.4.3.3 Generalised Solution As suggested by Section 2.4.3.1 and Section 2.4.3.2, we can often deal with problematic graph structure by moving the problematic information to newly added vertices instead. In fact, this is exactly the same principle the transforms from Section 2.4.1.1 and Section 2.4.2.1 are also based on. However, one important thing to note is that the newly introduced vertices should be distinguishable from normal vertices in the graph. Otherwise it is impossible to tell apart original graph structure from transformed graph structure. This in turn would mean the transform does not preserve the isomorphic equivalence relation between the original graphs.

2.5 gMark Extensions

In order to facilitate the evaluation approach and make the various required utility functions more accessible for future research, most of the utilities have been added to gMark [31]. Originally gMark was introduced by Bagan et al. as a domain- and query language-independent graph instance and query workload generator [6]. Later during a project to automatically generate schema based CPQs for benchmarking [29], I started a rewrite of the gMark software. This rewrite is publicly released under the GPL v3.0 on GitHub[4] and also where all utility functions were implemented. Specifically, all the utilities that will be introduced in the rest of this section are available starting from gMark release v1.1.

2.5.1 Plain CPQ generation

As noted in the introduction, the gMark rewrite already had the required capabilities to generate CPQs. However, these CPQs would be generated for a specific graph schema and potentially targeting a specific selectivity. This means that generating CPQs like this would not truly represent a random selection of CPQs from the space of all CPQs. Moreover, this way of generating CPQs would also be relatively expensive. Ideally we just want to generate CPQs randomly, following the rules in the CPQ grammar introduced in Section 2.1. This would both be significantly faster, therefore allowing for larger CPQs to be generated, and be a better representation of a random sample of CPQs. The algorithm created for this is fairly straightforward and is given in Algorithm 1.

Algorithm 1 Conjunctive Path Query Generation

```
1: procedure GeneratePlainCPQ(r ∈ ℕ, i ∈ {true, false}, L) ▷ Rule applications, allow identity, labels
2:     if r = 0 then
3:         if i ∧ RandomBoolean() = true then
4:             return id
5:         else
6:             l ← SelectRandom(L)
7:             if RandomBoolean() = true then
8:                 return l
9:             else
10:                return l⁻
11:            end if
12:        end if
13:    else
14:        s ← UniformRandom(0, r − 1)
15:        if RandomBoolean() = true then
16:            return GeneratePlainCPQ(s, false, L) ∘ GeneratePlainCPQ(r − s − 1, false, L)
17:        else
18:            return GeneratePlainCPQ(s, true, L) ∩ GeneratePlainCPQ(r − s − 1, false, L)
19:        end if
20:    end if
21: end procedure
```

[4] https://github.com/RoanH/gMark

It should be noted that we are intentionally preventing two technically valid but otherwise uninteresting CPQ patterns from being generated. The first is the conjunction of the identity operation with itself, given that $id \cap id = id$. The second is the join of identity with another CPQ, for a CPQ q this would result in $q \circ id = q$. The algorithm takes three arguments as input. The first r is the number of join and conjunction rule applications. From the CPQ grammar, this refers to the exact number of times the grammar steps containing join '$\circ$' and conjunction '$\cap$' are applied. The second argument i is a boolean indicating whether it is allowed for the algorithm to return the CPQ containing only the identity operation. The last argument L is a set of labels that the algorithm is allowed to pick from to form the CPQ. This set of labels should not include inverse labels.

For the initial call to the algorithm the value of i should be set to **false**. Not doing this makes it possible for the CPQ consisting of only id to be generated, for most applications this will be an undesired query. The set of labels L should also contain at least one label. There are no restrictions on r as long as it is a natural number (zero is allowed).

2.5.2 Other Utilities

The other added functionality is mostly made up of self explanatory utilities. The notable functions will be briefly summarised below:

1) **CPQ API**: An API has been added to allow for easy construction of CPQs. The main goal of this API is to allow for easy prototyping and to form a basis for simple algorithms that work with CPQs.

2) **CPQ query graph**: Functionality has been added to compute the query graph of a CPQ following the theory discussed in Section 2.2.

3) **Edge labels to vertices**: A utility function was added to convert an edge labelled graph to a graph without edge labels following the procedure given in Section 2.4.1.1.

4) **Adjacency lists**: Logic has been added to easily compute an adjacency list representation of graphs.

The remaining changes are mostly small utilities and usability improvements. Some of these are smaller features that the introduced notable features are based on.

2.6 Evaluation Framework

In order to be able to evaluate all the algorithms we will introduce in Section 3 a single codebase with bindings for all the algorithms was created. This codebase is released as open source on GitHub[5] under the GPL v3.0 and makes use of gMark [29] and the extensions introduced in Section 2.5. The setup of the evaluation framework is such that adding more algorithms is straightforward to do and a docker container is provided to eliminate any possible dependency issues. The evaluation framework itself is implemented in Java and currently has support for Java, C, C++ and Python algorithms. Technical details on getting started with the evaluation framework can be found in Appendix B. The remainder of this section will focus on explaining the key features of the evaluation framework.

The evaluation framework is made up of 3 major parts: the algorithm implementations, the evaluation & timing logic, and various utilities for transforming graphs.

2.6.1 Algorithm Implementations

The algorithm implementations are essentially small programs bridging the original implementation of an algorithm with the evaluation framework. If the algorithm was originally written in C or C++ then the Java Native Interface (JNI) [59] can be used to efficiently link the algorithm at a negligible performance cost (a few microseconds for large amounts of data). For Python algorithms a Python subprocess is used which has a relatively high but constant performance cost associated with starting a Python kernel. Algorithms originally implemented in Java can simply be linked directly.

Algorithm implementations are also responsible for converting the input CPQ query graphs to a format they can accept, possibly using some of the utility functions available in the framework or gMark.

Finally, algorithm implementations are required to return three runtime values in nanoseconds. The first one representing time spent on the framework side running graph transforms, the second one representing time spent setting up in the native language for the algorithm, and the last one the time spent by the algorithm canonizing the input graph. More details on these times will be given in Section 2.6.2.

[5] `https://github.com/RoanH/CPQKeys`

2.6.2 Evaluation Logic

The general procedure when evaluating an algorithm using the framework is as follows. First the algorithm is evaluated on a small randomly generated warmup dataset containing 10 graphs generated in the same way as the smallest dataset to be evaluated. No runtime information is collected about this warmup run.

Second all the datasets to be evaluated are sent to the algorithm in order of increasing size. The exact details of these datasets will be discussed in Section 4.1. Runtime information for these runs is collected and stored. Algorithms are also interrupted if their runtime starts to exceed the configured timeout value.

Next we will discuss each of the collected runtime values in more detail:

1) **Setup Runtime**: This accounts for the time spent on the framework side preparing to call the algorithm. The framework side can also be thought of as the Java side if the algorithm being evaluated is not itself also implemented in Java. The main factors contributing to this time are actions required to transform a CPQ query graph to a suitable algorithm input. Major time sinks under this category include graph transforms to make the graph undirected, transforms to remove edge labels, and the transform to turn the graph into a coloured graph.

2) **Native Setup Runtime**: This category accounts for time spent setting up to call the canonization subroutine of the algorithm. This step primarily consists of initialising the proper algorithm specific data structures.

3) **Canonization Runtime**: As its name suggests, this category accounts solely for the time spent running the canonization subroutine of the algorithm.

4) **Total Runtime**: This is the total time spent by the algorithm working on a single CPQ query graph as measured on the framework side. This runtime category encompasses all the other runtime categories.

5) **Other Runtime**: This category accounts for time spent that does not belong to any of the other categories (except for the total). Essentially this category is the total runtime minus the setup, native setup and canonization runtime.

2.6.3 Utilities

While most more general utilities were added to gMark [31] and can be found in Section 2.5.2, some more niche or limited utilities are provided directly in the framework. The notable functions will be summarised below:

1) **Runtime Reports**: Various utilities have been added to aggregate, summarise and save the results of algorithm evaluation runs.

2) **Directed graph to undirected graph**: A utility function was added to convert a directed graph to an undirected graph following the procedure given in Section 2.4.2.1.

3) **Coloured Graph**: Utilities have been added to convert a vertex labelled graph to a coloured graph in a format suitable for passing to an algorithm implementation. This output format presents the graph as an adjacency list together with a map indicating vertex colours.

4) **Datasets**: A utility class has been added to represent a dataset. This class also reports statistics about the dataset and has a utility function to easily generate a CPQ query graph dataset.

3 Canonization Algorithms

In this section we will introduce all the algorithms that will be evaluated. The first five sections will focus on introducing the algorithms that already have an existing implementation. In Section 3.6 we will discuss some promising theoretical algorithms from the literature that were not evaluated as they currently do not have a concrete implementation.

In Table 2 an overview is given of the software evaluated for this report. The change date indicates the last date the canonization algorithm itself was updated, while the release date indicates the most recent release date of the software package as a whole. It is worth noting that older versions of bliss have a Java and Python implementation. However, since the C++ version is much more recent, that is the one evaluated in this project.

Software	Version	Change Date	Release Date	Language	License
Bliss	0.77	Unknown	2021-02-18	C++	LGPL v3.0
Nauty	2.7	2017-10-14	2022-07-01	C	Apache v2.0
Nishe	36b97d3	2010-07-20	Unknown	C++	LGPL v3.0
Scott	01d26aa	2020-05-05	2020-05-10	Python	MIT
Traces	2.2	2018-06-07	2022-07-01	C	Apache v2.0

Table 2: Overview of canonization software.

3.1 Nauty

One of the oldest and most well known algorithms for practical graph isomorphism testing and canonical labelling was theorised in a masters thesis by Brendan McKay in 1976 [50] and further developed and published [51, 53]. This algorithm was based on the refinement-individualization paradigm proposed by Parris and Read [60], which was later refined by Corneil and Gotlieb [16] and Arlazarov et al. [79]. This approach refines partitions of the vertex set while keeping some vertices fixed. The main difference between the refinement-individualization paradigm and McKay's algorithm was that McKay's algorithm made clever use of automorphisms to reduce the size of the search space. Sometime after the publication of the first paper [53] the proposed algorithm was actually implemented and later became known under the name 'nauty' (**no** **aut**omorphisms, **y**es?).

Currently Nauty is the most established and most extensive software available for computing automorphism groups. This is especially true after Nauty and Traces [61] were merged and became a single software package [56]. Since Traces covers for some of the shortcomings of Nauty, this made the software package as a whole more versatile. More details on Traces will be given in Section 3.5. However, instead of only focussing on computing these automorphism groups, Nauty, or rather the Nauty & Traces combined software package also contains many utilities for working with graphs and generating graphs. Nauty is also the software that is used internally by popular larger packages such as Magma [14], SageMath [67] and GAP [24].

Nauty is implemented in two different ways. One implementation is the original version, which is aimed at very dense graphs and uses an adjacency matrix to represent the graph data. This version of Nauty is referred to as the dense version of Nauty. Later a program called saucy [19] was published, which is a program exclusively focussed on isomorphism detection. Saucy was heavily inspired by Nauty, but made use of sparse data structures to represent the graph data. This modification made saucy much more suitable for many real world use cases. In response to this a version of Nauty was released that used sparse data structures. This version of Nauty is referred to as the sparse version of Nauty. Throughout this report, where space is limited, the sparse and dense version of Nauty will be referred to as Nauty(S) and Nauty(D) respectively.

Both implementations of Nauty accept a directed coloured graph as input that is not allowed to contain parallel edges. For this project that means we need to transform CPQ query graph edge labels before we can use Nauty. The version of Nauty that will be evaluated is version 2.7 which is included in the 2.7r4 release of the Nauty & Traces software package [54].

3.2 Bliss

In their papers [39, 38] Tommi Junttila and Petteri Kaski introduce a new piece of open source software called Bliss to compute canonical forms and automorphism groups of graphs. Their approach is based on the well established individualization and refinement scheme [60, 16, 79] and introduces several optimisations for traversing the search tree. The optimisations focus primarily on large sparse graphs, but the tool still offers comparable or better performance on other types of graphs. It is noted in the paper that the family of test graphs made with the Miyazaki construction [57] are the only family of graphs that still show clear exponential runtime scaling with the graph size.

For input Bliss can accept both directed and undirected coloured graphs. However, Bliss does not accept parallel edges in the input graphs. This means that some transform will be required to use Bliss to compute a canonical form for CPQ query graphs.

Bliss is also well established software and is included in several popular larger pieces of software such as igraph [17], SageMath [67], SCIP [8], Arabesque [74], GraKeL [70], Peregrine [33, 32] and Pangolin [15]. After a long period of inactivity from its version 0.73 release in 2015 [37], a number of new releases came out in 2021 on a new site [36]. For this project the most recent of these releases Bliss 0.77 [35] will be evaluated.

3.3 Nishe

Originally started as a PhD thesis by Greg Tener [75] and later continued and published as a paper [78], Nishe was created specifically to address Nauty's weakness when handling Miyazaki graphs [57]. The paper proposes a modification for standard canonical labelling algorithms that allows them to process Miyazaki graphs in polynomial time, improving on the exponential runtime scaling observed in Nauty (release 1.7). Standard here was used to refer to algorithms that use the refinement and individualization technique [60, 16, 79], which is the most popular general approach to canonization, originally popularised by Nauty [53] and used by many other canonization algorithms. Nishe also follows the standard approach to canonization, but introduces the the concept of a guide tree which is used to improve the branching choices that are made by the algorithm. These improved branching choices allow it to avoid the worst case behaviour that was observed in Nauty on Miyazaki graphs.

As input Nishe accepts a coloured graph without parallel edges. This means that a graph transformation is required to convert edge labels before a CPQ query graph can be passed as input to Nishe. For this project the most recent version of Nishe as present in the GitHub repository [76] is evaluated, which currently has head commit `36b97d3`. This GitHub repository is a mirror of the original archived Google code project page [77].

3.4 Scott

In their paper [10] Nicolas Bloyet, Pierre-François Marteau and Emmanuel Frenod introduce a novel algorithm called Scott based on graph rewriting to compute the canonical labelling of graphs with both edge and node labels. They later prove their algorithm correct in a follow up paper [11]. The general idea behind Scott is as follows. First a suitable root vertex is selected using heuristics. Second the graph is transformed into a tree rooted at the selected root vertex by using a number of proposed graph rewriting rules to eliminate cycles. Finally, the resulting tree is transformed into a canonical string representing the isomorphism class of the input graph, this follows from the fact that a planar representation of a rooted tree admits an unambiguous canonical form expressed as a sequence of words [58].

Although both papers on Scott note that extending the algorithms to handle directed graphs is an open research question, the provided implementation does appear to support directed graphs. Therefore, in this report we will attempt to use both the proven to be correct undirected version of Scott, and the directed version of Scott. If the directed version of Scott then ends up performing exceptionally well it could be worth trying to prove its correctness.

Depending on the version of Scott used it either requires a transform to transform directed edge or CPQ query graphs can be used as input directly. The version of Scott evaluated in this project is the most recent version present in the GitHub repository [9], which currently has head commit `01d26aa`.

3.5 Traces

In his paper [61] Adolfo Piperno introduces a new tool named Traces for canonical graph labelling. Similar to many other tools Traces also makes use of the individualization-refinement paradigm [60, 16, 79]. The main contribution of the paper is a new search space reducing algorithm design. Instead of following Nauty's approach, Traces instead revisits all of the core concepts that make up the individualization-refinement paradigm and suggests a new approach of dealing with them. Therefore, unlike other derivative works that primarily focus on improving heuristics, Traces does not have trouble with the same classes of graphs that Nauty does. Notable is that Traces does away with the backtracking strategy employed by Nauty to traverse the search tree, instead using a breadth-first search. Another notable change is the introduction of the Schreier-Sims algorithm [71] to manipulate detected automorphisms. Traces derives its name from the concept of a *trace*, which is a linear representation of a partition that is used to compare partitions without having to compute them completely.

Traces was experimentally shown to be capable of efficiently dealing with graphs that existing tools had trouble with, though the paper notes that some of these graphs were designed specifically to evoke worse case behaviour in these existing tools. Traces was later merged with Nauty to form the Nauty & Traces software package [56], bundling both tools starting with the 2.5 release.

The current version of Traces is still fairly restrictive when it comes to the accepted input graphs, as only undirected graphs without edge labels are accepted. This means that for our purpose of canonizing CPQ query graphs we will need to run multiple transforms to transform the input query graph before Traces can be used. The version of Traces that will be evaluated is version 2.2 which is included in the 2.7r4 release of the Nauty & Traces software package [54].

3.6 Without existing implementation

Unfortunately many promising canonization algorithms found in the literature were never implemented by their authors. This is especially true for canonization algorithms focussed on dealing with special classes of graphs. A possible explanation [39] for the absence of actual implementations is that Nauty and its derivatives based on the individualization and refinement principle [60, 16, 79] are usually sufficient to handle real world use cases.

As discussed in Section 2.3 CPQs have a very low treewidth of 2. Therefore, algorithms specifically designed for graphs with a treewidth of 2, planar graphs, or more general, algorithms designed for bounded treewidth graphs, could be promising options for the canonization of CPQ query graphs. In fact, it is known that canonization of graphs of bounded treewidth is in the AC^1 complexity class [80]. Algorithms canonizing graphs of bounded treewidth in log space also exist [23].

However, all the discovered promising algorithms were without implementation. This section will briefly highlight some of them and their general approach to the problem.

3.6.1 Graph Rewriting

In their paper Arnborg et al. [1] present a canonization algorithm for partial 2- and 3-trees based on the idea of using reduction rules to continuously reduce the size of the graph until just a single vertex remains. Information about the application of these reduction rules is stored as vertex and edge labels. Although the original form of the algorithm does not support edge labels, vertex labels, directed graphs, multi graphs or self loops. The algorithm itself does make use of these, meaning it is possible that extending it to work with a labelled directed graph as input is realistic.

3.6.2 Using Trees

In their paper Arvind et al. [2] introduce a canonization algorithm for partial 2-trees based on exploiting what they call the 'tree of cycles'. Their idea is a combination of a novel approach of dealing with partial 2-trees and Lindell's well established canonization algorithm for trees [43].

3.6.3 Group-theoretic Work

Grohe et al. proposed a new isomorphism algorithm [26] based on a group-theoretic approach and extending the general quasipolynomial-time algorithm for the graph isomorphism problem by Babai [4], which in turn was based on Luks' framework [47]. They later [27] use these results to improve an FPT algorithm by Lokshtanov et al. [44] to obtain a faster algorithm for isomorphism testing. With some modifications that lead to a slightly worse running time this algorithm could also be used for canonization.

Babai himself later also extended his quasipolynomial-time isomorphism algorithm [4] to a canonization algorithm [3]. At the same time Pascal Schweitzer and Daniel Wiebking published a paper extending Luks' framework, presenting a unified framework for the design of canonization algorithms based on canonizing combinatorial objects [69]. Daniel Weibking later [81] builds upon this framework and Babai's extension to generalise Babai's work. The result is a versatile framework for canonizing various objects in quasipolynomial-time, ending with the canonization of graphs with bounded treewidth.

With all these frameworks being actively developed for isomorphism testing and canonization, a group-theoretic approach to canonization might end up being very flexible.

4 Performance Evaluation

In this section the results of evaluating all the algorithms will be discussed as well as the general evaluation methodology. All the used datasets will also be introduced in detail.

4.1 Methodology

To evaluate all the algorithms we make use of version 1.0 of the framework [30] introduced in Section 2.6. This means that all collected runtime results will be in nanoseconds. The general idea for the evaluation is to continuously send larger datasets to an algorithm until the algorithm is no longer able to deal with the dataset within the set time limit. The specifics of these datasets are given in Section 4.2. If an algorithm needs to make a directed graph undirected the method from Section 2.4.2.1 is used and if an algorithm needs to make labelled edges in a graph unlabelled then the algorithm from Section 2.4.1.1 is used.

Each of the algorithms introduced in Section 3 will be evaluated as follows. First, an algorithms needs to compute the canonical form of a small randomly generated dataset as a warmup. Second, each algorithm will be tasked with computing the canonical form of all the CPQ query graphs in the smallest dataset. Each dataset will contain exactly 100 CPQ query graphs. If the algorithm manages to compute the canonical form of all CPQ query graphs within 100 seconds, then it will be given a new dataset approximately twice the size of the previous dataset. As long as the algorithm can canonize all the graphs in the dataset within 100 seconds it will keep getting a larger dataset. This equates to an average of at most 1 second per query graph. It should be noted that the time required for graph transforms and data structure initialisation is counted against the time limit. Evaluation of an algorithm ends either when the largest dataset has already been processed by the algorithm or when the algorithm fails to canonize all the CPQ query graphs within the time limit.

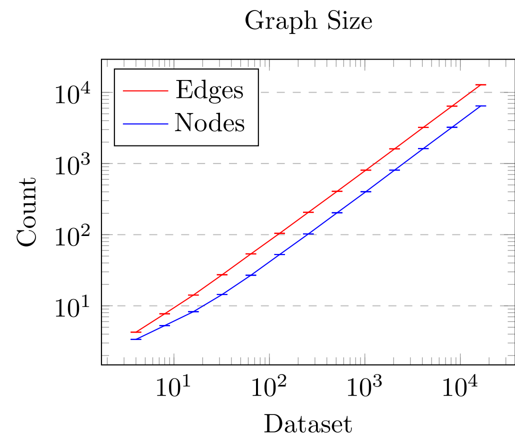
All tests were executed on a server running Ubuntu Linux 20.04.4 LTS with an Intel Xeon D-1541 CPU with 8 cores and 16 threads at 2.7 GHz and 32 GB of DRR4 ECC RAM at 2133 Mhz. Since most algorithms are single threaded only a single CPU thread will be used most of the time, Scott is multi-threaded and will be able to use as many threads as it needs.

4.2 Datasets

The datasets used to evaluate the algorithms are composed of CPQ query graphs. Each of these datasets was obtained by running the CPQ query graph algorithm introduced in Section 2.2 on 100 random CPQs generated by the CPQ generation algorithm introduced in Section 2.5.1. To determine the size of the dataset the rules parameter of the CPQ generation algorithm is used. The number of rule applications is also the name of the dataset, e.g. dataset 16 was generated with 16 as the rules parameter for the CPQ generation algorithm. All the datasets are generated at runtime by the evaluation framework using 1234 as the random seed and using gMark version 1.1 [31]. Saved XML files of the CPQs in each dataset are also available for download[6]. An overview of the average size with standard deviation of the resulting query graphs is given in Figure 8a, the sizes are also plotted in Figure 8b with errors bars representing the standard deviation. As can be seen in the figure, the size scaling is very consistent with very little variation in graph size within each dataset.

Dataset	Graphs	Nodes	Edges
4	100	3.37 ± 1.33	4.26 ± 0.82
8	100	5.27 ± 1.82	7.72 ± 1.13
16	100	8.27 ± 2.63	14.16 ± 1.95
32	100	14.42 ± 3.65	27.33 ± 2.89
64	100	26.90 ± 4.23	53.65 ± 4.41
128	100	52.60 ± 7.53	104.39 ± 7.97
256	100	102.33 ± 9.52	206.06 ± 12.03
512	100	203.21 ± 13.58	406.47 ± 19.45
1024	100	401.73 ± 20.52	806.90 ± 30.26
2048	100	808.19 ± 26.29	1607.19 ± 41.74
4096	100	1622.43 ± 44.41	3217.24 ± 64.02
8192	100	3231.41 ± 55.90	6408.11 ± 95.01
16384	100	6444.40 ± 64.58	12790.79 ± 114.15

(a) Overview of average dataset sizes with standard deviation.



(b) Dataset size scaling with error bars.

Figure 8: Dataset size overview.

[6] https://research.roanh.dev/cpqkeys/datasets

The sizes given in Figure 8 of the query graphs in these datasets is the size of the CPQ query graph. Therefore, these numbers are not representative of the size of the graphs the algorithms are actually asked to canonize. The only exception being the directed version of Scott. For all the other algorithms one or more graph transforms are required before they can accept the query graph as input. In Table 3 the effective average input graph size is given for each of the algorithms.

	Scott (directed)		**Bliss, Nauty & Nishe**		**Scott (undirected)**		**Traces**	
Dataset	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
4	3.37	4.26	7.63	8.52	11.89	12.78	16.15	17.04
8	5.27	7.72	12.99	15.44	20.71	23.16	28.43	30.88
16	8.27	14.16	22.43	28.32	36.59	42.48	50.75	56.64
32	14.42	27.33	41.75	54.66	69.08	81.99	96.41	109.32
64	26.90	53.65	80.55	107.30	134.20	160.95	187.85	214.60
128	52.60	104.39	156.99	208.78	261.38	313.17	365.77	417.56
256	102.33	206.06	308.39	412.12	514.45	618.18	720.51	824.24
512	203.21	406.47	609.68	812.94	1016.15	1219.41	1422.62	1625.88
1024	401.73	806.90	1208.63	1613.80	2015.53	2420.70	2822.43	3227.60
2048	808.19	1607.19	2415.38	3214.38	4022.57	4821.57	5629.76	6428.76
4096	1622.43	3217.24	4839.67	6434.48	8056.91	9651.72	11274.15	12868.96
8192	3231.41	6408.11	9639.52	12816.22	16047.63	19224.33	22455.74	25632.44
16384	6444.40	12790.79	19235.19	25581.58	32025.98	38372.37	44816.77	51163.16

Table 3: Effective input graph size for each algorithm.

To clarify the data in Table 3 we categorize the algorithms based on their capabilities as reported in Table 1 in Section 2.4. This divides the algorithms into four categories, based on whether they support edge labels or directed graphs. Assume that originally the graphs have $|V|$ vertices and $|E|$ edges.

1) **Edge labels & directed graphs**: The only algorithm in this category is the directed version of Scott. Algorithms in this category can take the input CPQ query graph as is without any modifications. Therefore the input for these algorithms matches what is reported in Figure 8a and $|V|$ and $|E|$ remain the same.

2) **Only edge labels**: Algorithms that only support edge labels need to run a transform to make a directed graph undirected, the only algorithm that falls in this category is the undirected version of Scott. The exact transform used is the one introduced in Section 2.4.2.1. Therefore algorithms in this category need to deal with $|V| + 2 \cdot |E|$ vertices and $3 \cdot |E|$ edges.

3) **Only directed graphs**: This is the most common category for algorithms to fall into. Algorithms in this category need to run a transform to make all labelled edges unlabelled. The exact algorithm used for this purpose is the one introduced in Section 2.4.1.1. Therefore algorithms in this category need to deal with $|V| + |E|$ vertices and $2 \cdot |E|$ edges.

4) **Neither edge labels nor directed graphs**: The only algorithm that falls in this category is Traces. Traces needs to run both the directed to undirected transform and the labelled edges to unlabelled edges transform. The setup logic for traces first executes the undirected transform, recall that this transform only copies the original edge label to the middle of the three replacement edges. This means that the subsequently executed unlabelled transform will only transform this middle edge. Therefore algorithms in this category need to deal with $|V| + 3 \cdot |E|$ vertices and $4 \cdot |E|$ edges.

4.3 Results

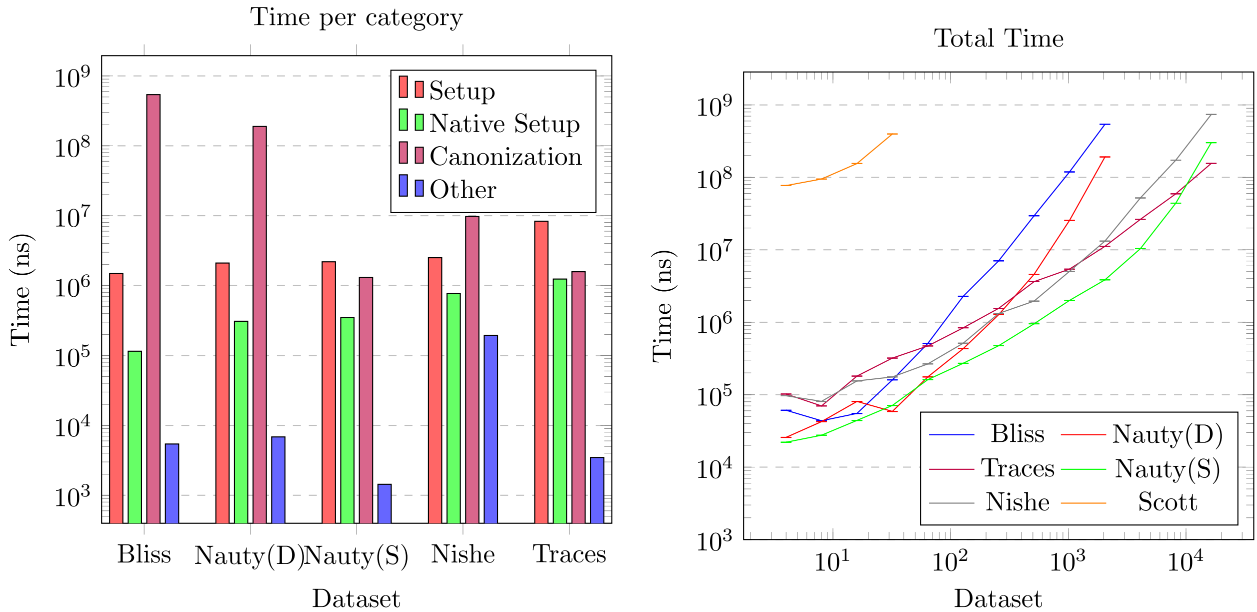
Starting with some practical details. First, the directed version of Scott did not manage to canonize the first dataset instead throwing an error. During preliminary testing Scott did generally tend to work on small graphs, but the increase in the number of graphs per dataset seems to have made it fail consistently. This is not entirely surprising, since as mentioned in Section 3.4, the algorithm was never claimed to officially support directed graphs as input, the required logic just happened to be present. However, due to this the directed version of Scott will not be present in the evaluation data. Therefore, all mentions of Scott in this section will refer to the undirected version. Second, as a general observation, during the evaluation the system never came close to running out of available RAM memory and all algorithms were able to to take a single CPU core to 100% utilisation except for Scott. While Scott did manage to use 100% of a core sometimes and even rarely 100% on all 16 logical cores, most of the time it was not able to do this.

Dataset	Bliss Canon	Bliss Total	Nauty (D) Canon	Nauty (D) Total	Nauty (S) Canon	Nauty (S) Total	Nishe Canon	Nishe Total	Scott Canon	Scott Total	Traces Canon	Traces Total
4	20 µs	61 µs	4 µs	26 µs	3 µs	22 µs	6 µs	97 µs	24 ms	77 ms	20 µs	102 µs
8	19 µs	44 µs	8 µs	42 µs	3 µs	28 µs	8 µs	81 µs	44 ms	95 ms	15 µs	70 µs
16	32 µs	55 µs	16 µs	81 µs	6 µs	44 µs	26 µs	155 µs	103 ms	155 ms	29 µs	181 µs
32	122 µs	160 µs	17 µs	59 µs	11 µs	71 µs	38 µs	176 µs	344 ms	397 ms	51 µs	320 µs
64	440 µs	510 µs	72 µs	176 µs	25 µs	161 µs	78 µs	266 µs	-	-	70 µs	471 µs
128	2 ms	2 ms	237 µs	432 µs	48 µs	271 µs	185 µs	514 µs	-	-	118 µs	838 µs
256	7 ms	7 ms	911 µs	1 ms	89 µs	474 µs	474 µs	1 ms	-	-	208 µs	2 ms
512	28 ms	30 ms	4 ms	5 ms	207 µs	952 µs	1 ms	2 ms	-	-	366 µs	4 ms
1024	118 ms	119 ms	24 ms	25 ms	536 µs	2 ms	3 ms	5 ms	-	-	724 µs	5 ms
2048	539 ms	541 ms	189 ms	192 ms	1 ms	4 ms	10 ms	13 ms	-	-	2 ms	11 ms
4096	-	-	-	-	5 ms	10 ms	43 ms	52 ms	-	-	3 ms	26 ms
8192	-	-	-	-	33 ms	44 ms	159 ms	173 ms	-	-	7 ms	59 ms
16384	-	-	-	-	275 ms	301 ms	707 ms	738 ms	-	-	15 ms	156 ms

Table 4: Canonization and total runtime for each algorithm per dataset.

Table 4 shows for each algorithm the average time spent on canonization and the average time spent in total when processing a single query in the dataset. For each dataset the fastest canonization time and the fastest total time are highlighted. Timeout runs are indicated with a '-'. A full breakdown of the times for each algorithm per category together with the standard deviation can be found in Appendix A.

When interpreting the results presented in this section it is worth noting that the 'setup', 'native setup' and 'other' time categories are not representative of an optimal implementation. The current implementation is generalised to easily support multiple algorithms instead of being tailored to a specific algorithm. A tailor made implementation focussed on performance is therefore expected to easily outperform the shown numbers for these categories. Only the canonization runtime can be taken as an accurate estimate of real world usage, the other categories should be considered upper bounds at best.

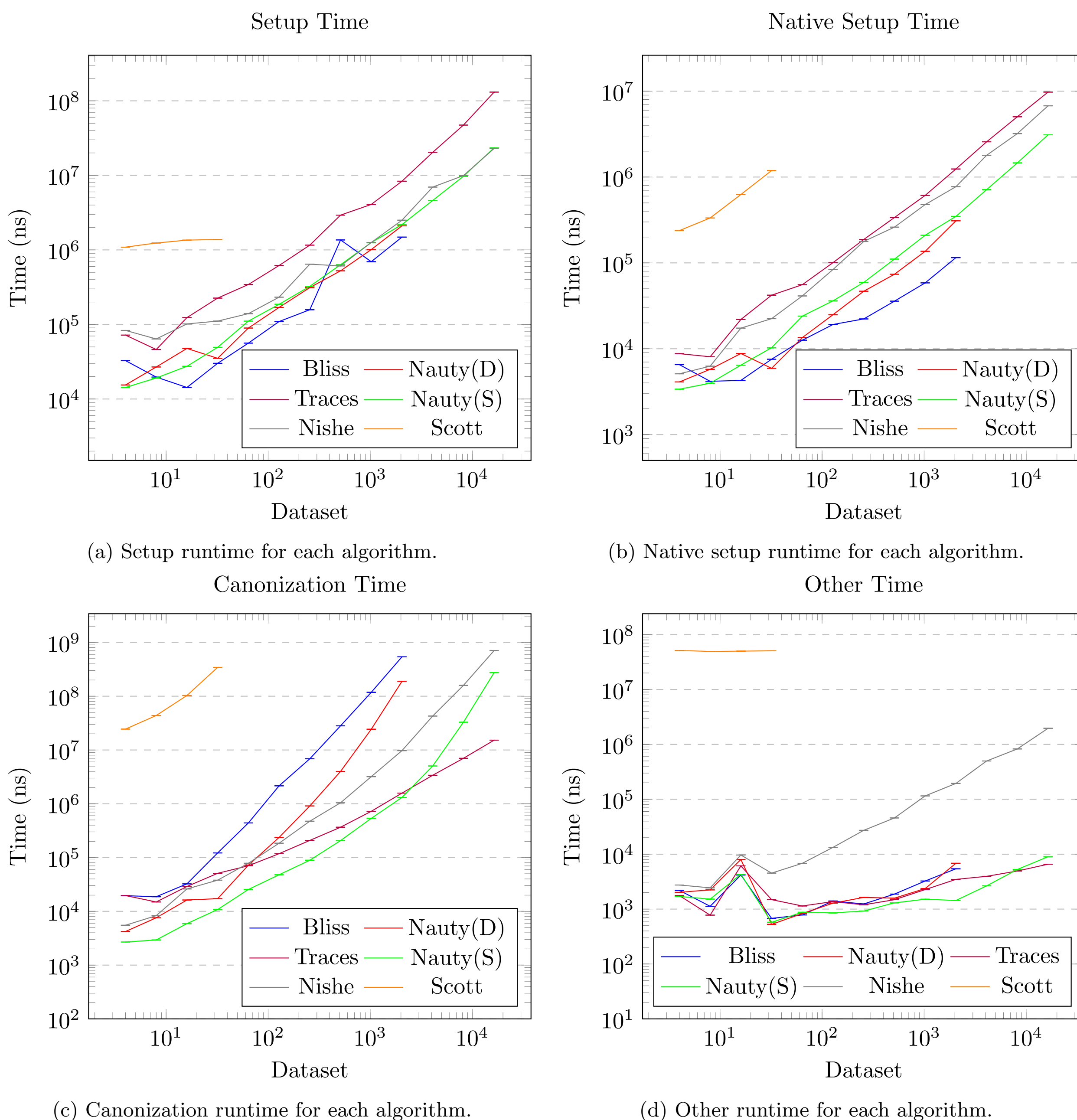


(a) Runtime per category for the 2048 dataset.

(b) Total runtime scaling per algorithm.

Figure 9: Total runtime scaling and breakdown.

In Figure 9b a plot showing the total runtime scaling for all the algorithms can be seen. All algorithms show a consistent weak exponential trend as the graph grows in size. Errors bars are used to show the standard deviation, however, none of the algorithms show any significant variation in total runtime. In Figure 9a a breakdown of the performance for each algorithm on the 2048 dataset can be seen. Scott has been excluded from this chart as it did not manage to evaluate this dataset. Notable in this chart is the relatively high 'other' runtime for nishe, this trend is also clearly visible in Figure 10d. Since it is categorized as 'other' we do not know exactly what the root cause is. A possible explanation is that this is time spent running destructors for some of the allocated data structures. It is also worth noting that even though this value is elevated, it is still only a fraction of a millisecond.



(a) Setup runtime for each algorithm.

(b) Native setup runtime for each algorithm.

(c) Canonization runtime for each algorithm.

(d) Other runtime for each algorithm.

Figure 10: Algorithm runtime scaling per category.

Figure 10 shows for each algorithm the runtime scaling per category. Mostly all the charts show either linear or weak exponential scaling in each of the categories. Some notable exceptions exist however. As noted already the 'other' time for nishe grows comparatively faster than the rest. However, the 'other' time for Scott is nearly constant and at the top of the graph. As mentioned in Section 2.6.1 this is the relatively high but constant cost associated with starting a Python kernel. Most notable however is the spike for the 3rd dataset (16) for all algorithms except for Scott. Even though this happens consistently no clear explanation for this peak was found and it being in the 'other' category makes it hard to investigate.

It is also worth noting that runtime data for the first dataset in all the graphs appears to be elevated. Even though warmup runs were performed, this suggests that perhaps more were needed. Also notable is the spike for Bliss in Figure 10a for the 8th dataset (512), this is due to one massive outlier in the data. This spike cannot be reproduced by rerunning Bliss with this dataset, suggesting that some unknown external factor caused this spike. Finally, Figure 10c clearly shows that out of all the algorithms the trend for Traces is the most favourable, even though Traces does not have the best performance on smaller graphs. Likely this is because the increased input graph size hinders it on smaller graphs. Possibly the breadth-first search employed by Traces is a good fit for CPQ query graphs, resulting in the near linear runtime scaling observed.

5 Concluding Remarks

As shown in the performance evaluation in Section 4.3 there is not a clear winner among the algorithms. However, the best performance is seen with either the sparse version of Nauty on small graphs or Traces on large graphs. Conveniently, both programs are part of the same software package and even use the same input data structure, making the Nauty & Traces software package as a whole the logical choice to go with to compute canonical forms for a language-aware index. An optimal implementation would then decide whether or use Nauty or Traces based on the size of the input query graph. More testing to determine a suitable switching point should be done after both algorithms are implemented with a focus on performance. Alternatively, since real world queries are generally expected to be on the smaller side and Nauty still outperforms Traces on reasonably large graphs, it is possible to only use Nauty.

Aside from the performance results there are also some other reasons why the Nauty & Traces software package is a good option. As mentioned in Section 3.1, the project has been consistently maintained for almost 50 years. Moreover it has more than one author, with Brendan McKay and Adolfo Piperno both working on it. In addition, it is well established and used in several popular larger pieces of software, making it likely to keep on being maintained and for new contributors to step in if required. Finally, the documentation for Nauty & Traces is very good and exhaustive. And if the documentation is not sufficient, the Nauty mailing list is active and its archive contains over 20 years worth of questions and answers.

5.1 Future Work

Even though this report presents a clear overview of the current state of the art in graph canonization for algorithms with an existing implementation, there is still much work that could be done. In general, future work can go in two main directions. Either the results of this report can be used as presented and built upon, or some of the less explored options for canonization can be investigated to see if they are competitive with or better than the algorithms evaluated in this report.

5.1.1 Using only isomorphism testing

This reports focusses on algorithms that compute a canonical form of an input graph that can be stored as this is believed to be the most scalable approach to build a language-aware index. However, it is possible to forgo the canonization step and instead compare two graphs for isomorphic equivalence directly. This raises the question if it would be viable to use a single representative graph from the isomorphic equivalence class as a key for the index. When we want to retrieve data from the index we then simply check the input graph for isomorphic equivalence with all the key set graphs.

Since checking two graph for isomorphic equivalence is much more computationally expensive than checking two canonical representations for equivalence, this method will inevitably be outperformed by a canonization based approach at some point. However, using canonization is typically slower than directly testing if two graphs are isomorphic. This is because isomorphism testing has many good heuristics it can employ. Moreover, as soon as a counter example is found that shows that two graphs are not isomorphic the algorithm can immediately return this result. Such short circuiting conditions are also expected to be relatively common. While isomorphism testing is hard in general, there are many easy tests that can be performed to confirm that two graphs can never be isomorphic. For example, if two graphs do not have the same number of vertices they cannot be isomorphic. As an other example, two graphs need to have the same number of nodes of a specific degree, otherwise they cannot be isomorphic. Therefore, we expect isomorphism testing algorithms to often execute very fast, whereas a canonization algorithm always has to process the entire input graph to produce a canonical representation.

A possible direction of future work would therefore be to investigate when exactly canonization consistently outperforms a direct isomorphism testing based approach. If this turning point is at an unrealistically high index size then direct isomorphism testing might be a viable alternative for index design.

Additionally, if canonization is not a requirement, then more existing algorithms become applicable to the problem. These algorithms usually tend to put their focus primarily on isomorphism testing. Some well established programs for this purpose are conauto [45, 46, 64], saucy [19, 20, 41, 40, 18], and GI-EXT [63, 62].

5.1.2 Output handling

Regarding future work that wants to use the output of some of the canonization algorithms additional aspects need to be taken into account. Most of the algorithms have their own way of representing the canonical form of the input graph. For the implemented algorithms the various forms are a string, a refinement trace, and a graph. Some algorithms also support more than one output format. For example, for Nauty & Traces the original canonical output is a graph, but after sorting this graph can be converted to a canonical string for the

input graph. This does however mean an extra runtime cost is incurred that is technically not required as the canonical graph can be efficiently compared using a provided utility function.

Another noteworthy point as brought up in Section 2.4 is the handling of colours as opposed to regular labels. During processing colours are distinct but indistinguishable. To check if two outputs are really isomorphically equivalent it is required to check if the colour classes that were found to be equivalent actually represent the same labels. Given the construction of the input graph this can be done by simple checking the label of any element from the colour class. Alternatively, it is possible to add extra structure to the input graph to make the colour classes distinguishable, as is further elaborated on in the manual for Nauty & Traces [55]. However, since this increases the size of the input graph this may incur a performance penalty.

None of these items represent major issues, however, they will need to be carefully considered when designing a complete language-aware index.

5.1.3 Specialised setup logic

The current implementation of the general logic around all the algorithms is written with reusability and clarity in mind. This makes it easy to implement algorithms and add new algorithms to the existing infrastructure without having to reimplement smaller utilities. However, the higher abstraction level required to make all the utilities general enough for reuse comes at a performance cost. When a specific algorithm is chosen then it would be beneficial for performance critical applications to rewrite the general logic such as transforms to be tailored specifically to the chosen algorithm. It is likely that in most cases a significant portion of the setup time and native setup time can be eliminated completely. Moreover, algorithms that use multiple transforms, like Traces, would benefit from a function that executes both transforms at the same time instead of sequentially.

5.1.4 Using specialised algorithms

As discussed in Section 3.6, theoretically fast algorithms for planar graphs or graphs of bounded treewidth exist. It is currently unknown how these algorithms fare against more general existing algorithms that can handle any input graph, as many of these algorithms were never implemented. In theory these algorithms could be faster as they can take advantage of the structure of their input graphs. However, more general existing software has had many years to develop and improve their heuristics. It is possible that a naive implementation of one of these specialised algorithms cannot outperform these heuristics. Moreover, in real world use cases we will typically encounter more smaller inputs, meaning performance on small graphs is very important. It could also be possible that augmenting general algorithms to take advantage of treewidth, or vice versa, using the heuristics in the specialised algorithms, could result in a better algorithm. On the other hand, it is also possible that this combination performs worse than the original algorithms it was created from. Further investigation of this topic could therefore lead to interesting results.

5.1.5 Custom canonization algorithm

Since CPQ query graphs have some very nice properties, being both planar and having treewidth 2, it also still remains an option to develop an entirely new canonization algorithm tailored specifically towards CPQs. Such an algorithm could for example be based on or inspired by one of the algorithms introduced in Section 3.6. Or alternatively, the approach could be completely novel.

5.1.6 Generalised Evaluation Framework

The current evaluation framework is geared towards using CPQ query graphs as input. However, it might be worthwhile to expand this framework to also be able to use other graphs as input. Parts of the framework are already setup in such a way to leave this possibility open.

5.1.7 Language-aware Index

As mentioned in Section 1, the ultimate goal is to make a language-aware graph database index. The findings in this report are an important part of finding the best way to manage the keys for this index. A natural next step for future work is then to take the findings in this report and start working towards the completion the CPQ based index. The paper on language-aware indexing for conjunctive path queries by Sasaki et al. [68] provides a good starting point for this. The current approach in this paper is to partition blocks of paths that are indistinguishable with respect to the query language. This means that the used index keys are currently paths and input queries are covered with paths to retrieve data from the index. Ideally we would want to change this to partition by CPQs instead.

References

[1] Stefan Arnborg and Andrzej Proskurowski. "Canonical representations of partial 2- and 3-trees". In: *BIT Numerical Mathematics* 32 (2 June 1992), pp. 197–214. DOI: 10.1007/BF01994877.

[2] Vikraman Arvind, Bireswar Das, and Johannes Köbler. "A Logspace Algorithm for Partial 2-Tree Canonization". In: *Computer Science – Theory and Applications*. Ed. by Edward A. Hirsch, Alexander A. Razborov, Alexei Semenov, and Anatol Slissenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 40–51. ISBN: 978-3-540-79709-8. DOI: 10.1007/978-3-540-79709-8_8.

[3] László Babai. "Canonical Form for Graphs in Quasipolynomial Time: Preliminary Report". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1237–1246. ISBN: 9781450367059. DOI: 10.1145/3313276.3316356.

[4] László Babai. "Graph Isomorphism in Quasipolynomial Time". In: *CoRR* abs/1512.03547 (2015). DOI: 10.48550/ARXIV.1512.03547. arXiv: 1512.03547.

[5] László Babai, William M Kantor, and Eugene M Luks. "Computational complexity and the classification of finite simple groups". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. 1983, pp. 162–171. DOI: 10.1109/SFCS.1983.10.

[6] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. "gMark: Schema-Driven Generation of Graphs and Queries". In: *IEEE Transactions on Knowledge and Data Engineering* 29.4 (2017), pp. 856–869. DOI: 10.1109/TKDE.2016.2633993.

[7] Umberto Bertelè and Francesco Brioschi. "Nonserial Dynamic Programming". In: *Mathematics in Science and Engineering* 91 (1972). URL: https://www.sciencedirect.com/bookseries/mathematics-in-science-and-engineering/vol/91.

[8] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. *The SCIP Optimization Suite 8.0*. ZIB-Report 21-41. Zuse Institute Berlin, Dec. 2021. URL: http://nbn-resolving.de/urn:nbn:de:0297-zib-85309.

[9] Nicolas Bloyet, Pierre-François Marteau, and Emmanuel Frenod. *Scott*. Version 01d26aa. May 2020. URL: https://github.com/theplatypus/scott.

[10] Nicolas Bloyet, Pierre-François Marteau, and Emmanuel Frenod. "Scott : A method for representing graphs as rooted trees for graph canonization". In: *COMPLEX NETWORKS 2019*. Studies in Computational Intelligence Series. Springer, Dec. 2019, pp. 578–590. DOI: 10.1007/978-3-030-36687-2_48. URL: https://hal.archives-ouvertes.fr/hal-02314658.

[11] Nicolas Bloyet, Pierre-François Marteau, and Emmanuel Frénod. "Canonical Forms for General Graphs Using Rooted Trees - Correctness and Complexity Study of the SCOTT Algorithm". working paper or preprint. Mar. 2020. URL: https://hal.archives-ouvertes.fr/hal-02495229.

[12] Manuel Bodirsky, Omer Gimenez, Mihyun Kang, and Marc Noy. "On the number of series parallel and outerplanar graphs". In: *2005 European Conference on Combinatorics, Graph Theory and Applications (EuroComb '05)*. Ed. by Stefan Felsner. Vol. DMTCS Proceedings vol. AE, European Conference on Combinatorics, Graph Theory and Applications (EuroComb '05). DMTCS Proceedings. Berlin, Germany: Discrete Mathematics and Theoretical Computer Science, 2005, pp. 383–388. DOI: 10.46298/dmtcs.3451.

[13] Hans L. Bodlaender. "A partial k-arboretum of graphs with bounded treewidth". In: *Theoretical Computer Science* 209.1 (1998), pp. 1–45. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(97)00228-4.

[14] Wieb Bosma, John Cannon, and Catherine Playoust. "The Magma algebra system. I. The user language". In: *J. Symbolic Comput.* 24.3-4 (1997). Computational algebra and number theory (London, 1993), pp. 235–265. ISSN: 0747-7171. DOI: 10.1006/jsco.1996.0125.

[15] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. "Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU". In: *Proc. VLDB Endow.* 13.8 (Aug. 2020).

[16] Derek G. Corneil and Calvin C. Gotlieb. "An Efficient Algorithm for Graph Isomorphism". In: *Journal of the ACM (JACM)* 17.1 (Jan. 1970), pp. 51–64. ISSN: 0004-5411. DOI: 10.1145/321556.321562.

[17] Gabor Csardi and Tamas Nepusz. "The igraph software package for complex network research". In: *InterJournal* Complex Systems (2006), p. 1695. URL: https://igraph.org.

[18] Paul T. Darga, Hadi Katebi, Mark Liffiton, Igor L. Markov, and Karem Sakallah. *Saucy3: Fast Symmetry Discovery in Graphs.* University of Michigan. 2012. URL: http://vlsicad.eecs.umich.edu/BK/SAUCY/.

[19] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. "Exploiting Structure in Symmetry Detection for CNF". In: *Proceedings of the 41st Annual Design Automation Conference.* DAC '04. San Diego, CA, USA: Association for Computing Machinery, 2004, pp. 530–534. ISBN: 1581138288. DOI: 10.1145/996566.996712.

[20] Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. "Faster Symmetry Discovery Using Sparsity of Symmetries". In: *Proceedings of the 45th Annual Design Automation Conference.* DAC '08. Anaheim, California: Association for Computing Machinery, 2008, pp. 149–154. ISBN: 9781605581156. DOI: 10.1145/1391469.1391509.

[21] Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. "Planar Graph Isomorphism is in Log-Space". In: *2009 24th Annual IEEE Conference on Computational Complexity.* 2009, pp. 203–214. DOI: 10.1109/CCC.2009.16.

[22] Mathieu Dutour. *[Nauty-list] digraph–>graph.* Nauty mailing list. Oct. 20, 2013. URL: https://mailman.anu.edu.au/pipermail/nauty/2013-October/000706.html.

[23] Michael Elberfeld and Pascal Schweitzer. "Canonizing Graphs of Bounded Tree Width in Logspace". In: *ACM Trans. Comput. Theory* 9.3 (Oct. 2017). ISSN: 1942-3454. DOI: 10.1145/3132720.

[24] *GAP – Groups, Algorithms, and Programming.* The GAP Group. URL: https://www.gap-system.org.

[25] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems". In: *J. ACM* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411. DOI: 10.1145/116825.116852.

[26] Martin Grohe, Daniel Neuen, and Pascal Schweitzer. "A Faster Isomorphism Test for Graphs of Small Degree". In: *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS).* 2018, pp. 89–100. DOI: 10.1109/FOCS.2018.00018.

[27] Martin Grohe, Daniel Neuen, Pascal Schweitzer, and Daniel Wiebking. "An Improved Isomorphism Test for Bounded-Tree-Width Graphs". In: *ACM Trans. Algorithms* 16.3 (June 2020). ISSN: 1549-6325. DOI: 10.1145/3382082.

[28] Rudolf Halin. "S-functions for graphs". In: *Journal of Geometry* 8.1 (Mar. 1976), pp. 171–186. DOI: 10.1007/BF01917434.

[29] Roan Hofland. *Conjunctive Path Query Generation for Benchmarking.* Capita Selecta Report. Eindhoven University of Technology, Mar. 2022. URL: https://research.roanh.dev/Conjunctive%20Path%20Query%20Generation%20for%20Benchmarking%20v2.8.pdf.

[30] Roan Hofland. *CPQKeys.* Version 1.0. July 2022. URL: https://github.com/RoanH/CPQKeys.

[31] Roan Hofland. *gMark.* Version 1.1. June 2022. URL: https://github.com/RoanH/gMark.

[32] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. "Peregrine: A Pattern-Aware Graph Mining System". In: *Proceedings of the Fifteenth European Conference on Computer Systems.* EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387548. URL: https://doi.org/10.1145/3342195.3387548.

[33] Kasra Jamshidi and Keval Vora. "A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE". In: *SIGOPS Oper. Syst. Rev.* 55.1 (June 2021), pp. 1–10. ISSN: 0163-5980. DOI: 10.1145/3469379.3469381. URL: https://doi.org/10.1145/3469379.3469381.

[34] Thor Johnson, Neil Robertson, P.D. Seymour, and Robin Thomas. "Directed Tree-Width". In: *Journal of Combinatorial Theory, Series B* 82.1 (2001), pp. 138–154. ISSN: 0095-8956. DOI: 10.1006/jctb.2000.2031.

[35] Tommi Junttila. *bliss.* Version 0.77. Feb. 2021. URL: https://users.aalto.fi/~tjunttil/bliss/download.html.

[36] Tommi Junttila. *The bliss tool.* 2021. URL: https://users.aalto.fi/~tjunttil/bliss/index.html.

[37] Tommi Junttila and Petteri Kaski. *bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs.* Sept. 1, 2015. URL: http://www.tcs.hut.fi/Software/bliss/.

[38] Tommi Junttila and Petteri Kaski. "Conflict Propagation and Component Recursion for Canonical Labeling". In: *Theory and Practice of Algorithms in (Computer) Systems – First International ICST Conference, TAPAS 2011, Rome, Italy, April 18–20, 2011. Proceedings.* Ed. by Alberto Marchetti-Spaccamela and Michael Segal. Vol. 6595. Lecture Notes in Computer Science. Springer, 2011, pp. 151–162. DOI: 10.1007/978-3-642-19754-3_16.

[39] Tommi Junttila and Petteri Kaski. "Engineering an efficient canonical labeling tool for large and sparse graphs". In: *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*. Ed. by David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick. Society for Industrial and Applied Mathematics, Jan. 2007, pp. 135–149.

[40] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. "Conflict Anticipation in the Search for Graph Automorphisms". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Nikolaj Bjørner and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 243–257. ISBN: 978-3-642-28717-6. DOI: 10.1007/978-3-642-28717-6_20.

[41] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. "Symmetry and Satisfiability: An Update". In: *Theory and Applications of Satisfiability Testing – SAT 2010*. Ed. by Ofer Strichman and Stefan Szeider. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 113–127. ISBN: 978-3-642-14186-7. DOI: 10.1007/978-3-642-14186-7_11.

[42] Casimir Kuratowski. "Sur le problème des courbes gauches en Topologie". In: *Fundamenta Mathematicae* 15 (1930), pp. 271–283.

[43] Steven Lindell. "A Logspace Algorithm for Tree Canonization (Extended Abstract)". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*. STOC '92. Victoria, British Columbia, Canada: Association for Computing Machinery, 1992, pp. 400–404. ISBN: 0897915119. DOI: 10.1145/129712.129750.

[44] Daniel Lokshtanov, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. "Fixed-Parameter Tractable Canonization and Isomorphism Test for Graphs of Bounded Treewidth". In: *SIAM Journal on Computing* 46.1 (2017), pp. 161–189. DOI: 10.1137/140999980.

[45] José Luis López-Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. "Conauto-2.0: Fast Isomorphism Testing and Automorphism Group Computation". In: *CoRR* abs/1108.1060 (2011). arXiv: 1108.1060.

[46] José Luis López-Presa, Luis Núñez Chiroque, and Antonio Fernández Anta. "Novel Techniques for Automorphism Group Computation". In: *Experimental Algorithms*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 296–307. ISBN: 978-3-642-38527-8. DOI: 10.1007/978-3-642-38527-8_27.

[47] Eugene M. Luks. "Isomorphism of graphs of bounded valence can be tested in polynomial time". In: *Journal of Computer and System Sciences* 25.1 (1982), pp. 42–65. ISSN: 0022-0000. DOI: 10.1016/0022-0000(82)90009-5.

[48] mathtalk-ga. *Graph theory algorithm for directed -> undirected graphs*. Google Answers. Aug. 24, 2005. URL: http://answers.google.com/answers/threadview?id=559159.

[49] Brendan D. McKay. *[Nauty] Re: nauty-list digest, Vol 1 #116 - 3 msgs*. Nauty mailing list. Apr. 14, 2005. URL: https://mailman.anu.edu.au/pipermail/nauty/2005-March/000275.html.

[50] Brendan D. McKay. "Backtrack Programming and the Graph Isomorphism Problem". MA thesis. Unversity of Melbourne, July 1976. URL: https://users.cecs.anu.edu.au/~bdm/papers/McKayMastersThesis.pdf.

[51] Brendan D. McKay. "Computing automorphisms and canonical labellings of graphs". In: *Combinatorial Mathematics*. Ed. by D. A. Holton and Jennifer Seberry. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 223–232. ISBN: 978-3-540-35702-5. DOI: 10.1007/BFb0062536.

[52] Brendan D. McKay. *Labeled graphs*. Message to the Nauty mailing list. Sept. 8, 2013. URL: https://mailman.anu.edu.au/pipermail/nauty/2013-September/000693.html.

[53] Brendan D. McKay. "Practical Graph Isomorphism". In: *Congressus Numerantium* 30 (1981), pp. 45–87. URL: http://users.cecs.anu.edu.au/~bdm/papers/pgi.pdf.

[54] Brendan D. McKay and Adolfo Piperno. *Nauty & Traces*. Version 2.7r4. July 2022. URL: https://pallini.di.uniroma1.it/.

[55] Brendan D. McKay and Adolfo Piperno. *nauty and Traces User's Guide (Version 2.7)*. Sept. 2021. URL: https://pallini.di.uniroma1.it/nug27.pdf.

[56] Brendan D. McKay and Adolfo Piperno. "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2013.09.003.

[57] Takunari Miyazaki. "The complexity of McKay's canonical labeling algorithm". In: *Groups and Computation, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, June 7-10, 1995.* Ed. by Larry Finkelstein and William M. Kantor. Vol. 28. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1995, pp. 239–256. DOI: 10.1090/dimacs/028/14.

[58] Jacques Neveu. "Arbres et processus de Galton-Watson". fr. In: *Annales de l'I.H.P. Probabilités et statistiques* 22.2 (1986), pp. 199–207. URL: http://www.numdam.org/item/AIHPB_1986__22_2_199_0/.

[59] Oracle. *Java Native Interface Specification.* URL: https://docs.oracle.com/en/java/javase/18/docs/specs/jni/index.html.

[60] R Parris and Ronald C. Read. "A coding procedure for graphs". In: *Scientific Report. UWI/CC* 10 (1969).

[61] Adolfo Piperno. "Search Space Contraction in Canonical Labeling of Graphs". In: *CoRR* abs/0804.4881 (2008). DOI: 10.48550/ARXIV.0804.4881. arXiv: 0804.4881.

[62] Daniel Porumbel. *GI-EXT: Graph Isomorphism using EXTended graphs.* 2009. URL: https://sites.google.com/site/danielporumbel/giext/.

[63] Daniel Cosmin Porumbel. "Isomorphism Testing via Polynomial-Time Graph Extensions". In: *Journal of Mathematical Modelling and Algorithms* 10 (2 June 2011), pp. 119–143. ISSN: 1572-9214. DOI: 10.1007/s10852-010-9145-x.

[64] José Luis López Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. *Graph Isomorphism Algorithm conauto.* Jan. 15, 2013. URL: https://sites.google.com/site/giconauto/.

[65] David Richerby. *Converting a digraph to an undirected graph in a reversible way.* Computer Science Stack Exchange. Jan. 15, 2014. URL: https://cs.stackexchange.com/q/19758.

[66] Neil Robertson and P.D Seymour. "Graph minors. II. Algorithmic aspects of tree-width". In: *Journal of Algorithms* 7.3 (1986), pp. 309–322. ISSN: 0196-6774. DOI: 10.1016/0196-6774(86)90023-4.

[67] The Sage Developers. *SageMath, the Sage Mathematics Software System.* URL: https://www.sagemath.org.

[68] Yuya Sasaki, George Fletcher, and Makoto Onizuka. "Language-aware Indexing for Conjunctive Path Queries". In: *38th IEEE International Conference on Data Engineering (ICDE)* (May 2022). (Virtual) Kuala Lumpur, Malaysia. IEEE Computer Society.

[69] Pascal Schweitzer and Daniel Wiebking. "A Unifying Method for the Design of Algorithms Canonizing Combinatorial Objects". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing.* STOC 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1247–1258. ISBN: 9781450367059. DOI: 10.1145/3313276.3316338.

[70] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. "GraKeL: A Graph Kernel Library in Python". In: *Journal of Machine Learning Research* 21.54 (2020), pp. 1–5.

[71] Charles C. Sims. "Computation with Permutation Groups". In: *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation.* SYMSAC '71. Los Angeles, California, USA: Association for Computing Machinery, 1971, pp. 23–28. ISBN: 9781450377867. DOI: 10.1145/800204.806264.

[72] Guenter Stertenbrink. *[Nauty] Can we do colored edges in nauty?* Nauty mailing list. Mar. 12, 2005. URL: https://mailman.anu.edu.au/pipermail/nauty/2005-March/000273.html.

[73] Guenter Stertenbrink. *[Nauty] canonical label for directed graphs.* Nauty mailing list. Aug. 3, 2005. URL: https://mailman.anu.edu.au/pipermail/nauty/2005-August/000328.html.

[74] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. "Arabesque: A System for Distributed Graph Mining". In: *Proceedings of the 25th Symposium on Operating Systems Principles.* SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 425–440. ISBN: 9781450338349. DOI: 10.1145/2815400.2815410.

[75] Greg Tener. "Attacks On Difficult Instances Of Graph Isomorphism: Sequential And Parallel Algorithms". PhD thesis. University of Central Florida, 2009. URL: https://stars.library.ucf.edu/etd/4004/.

[76] Greg Tener. *nishe.* Version 36b97d3. July 2010. URL: https://github.com/b0ri5/nishe-googlecode.

[77] Greg Tener. *nishe. A canonical labeling and symmetry detecting algorithm for graphs.* URL: https://code.google.com/archive/p/nishe/.

[78] Greg Tener and Narsingh Deo. "Efficient isomorphism of Miyazaki graphs". In: *Bulletin of the Institute of Combinatorics and its Applications* 61 (Jan. 2011). URL: https://www.researchgate.net/publication/264944171_Efficient_isomorphism_of_Miyazaki_graphs.

[79] Arlazarov Vladimir Lvovich, I.I. Zuev, A.V. Uskov, and I.A. Faradzhev. "An algorithm for the reduction of finite non-oriented graphs to canonical form". In: *USSR Computational Mathematics and Mathematical Physics* 14.3 (1974), pp. 195–201. ISSN: 0041-5553. DOI: 10.1016/0041-5553(74)90114-1.

[80] Fabian Wagner. "Graphs of Bounded Treewidth Can Be Canonized in AC^1". In: *Computer Science – Theory and Applications*. Ed. by Alexander Kulikov and Nikolay Vereshchagin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 209–222. ISBN: 978-3-642-20712-9. DOI: 10.1007/978-3-642-20712-9_16.

[81] Daniel Wiebking. "Graph isomorphism in quasipolynomial time parameterized by treewidth". In: *CoRR* abs/1911.11257 (2019). arXiv: 1911.11257.

A Evaluation Data

This appendix contains the raw runtime data collected by running each of the algorithms. For each dataset and for each of the timed categories the average runtime is listed together with the sample standard deviation.

A.1 Bliss

Dataset	Setup	Native Setup	Canonization	Other	Total
4	33 µs ± 5 µs	7 µs ± 5 µs	20 µs ± 12 µs	2 µs ± 980 ns	61 µs ± 23 µs
8	20 µs ± 3 µs	4 µs ± 3 µs	19 µs ± 15 µs	1 µs ± 1 µs	44 µs ± 34 µs
16	14 µs ± 3 µs	4 µs ± 3 µs	32 µs ± 13 µs	4 µs ± 35 µs	55 µs ± 44 µs
32	30 µs ± 3 µs	8 µs ± 3 µs	122 µs ± 46 µs	678 ns ± 237 ns	160 µs ± 49 µs
64	56 µs ± 4 µs	13 µs ± 4 µs	440 µs ± 167 µs	783 ns ± 329 ns	510 µs ± 178 µs
128	110 µs ± 6 µs	19 µs ± 6 µs	2 ms ± 687 µs	1 µs ± 1 µs	2 ms ± 707 µs
256	158 µs ± 8 µs	22 µs ± 8 µs	7 ms ± 1 ms	1 µs ± 587 ns	7 ms ± 1 ms
512	1 ms ± 12 µs	36 µs ± 12 µs	28 ms ± 4 ms	2 µs ± 899 ns	30 ms ± 12 ms
1024	695 µs ± 16 µs	59 µs ± 16 µs	118 ms ± 12 ms	3 µs ± 2 µs	119 ms ± 12 ms
2048	1 ms ± 26 µs	115 µs ± 26 µs	539 ms ± 35 ms	5 µs ± 3 µs	541 ms ± 35 ms

Table 5: Runtime evaluation data for Bliss.

A.2 Nauty (dense)

Dataset	Setup	Native Setup	Canonization	Other	Total
4	15 µs ± 2 µs	4 µs ± 2 µs	4 µs ± 3 µs	2 µs ± 678 ns	26 µs ± 7 µs
8	27 µs ± 1 µs	6 µs ± 1 µs	8 µs ± 4 µs	2 µs ± 1 µs	42 µs ± 10 µs
16	48 µs ± 2 µs	9 µs ± 2 µs	16 µs ± 7 µs	8 µs ± 61 µs	81 µs ± 67 µs
32	35 µs ± 2 µs	6 µs ± 2 µs	17 µs ± 5 µs	527 ns ± 252 ns	59 µs ± 10 µs
64	90 µs ± 3 µs	13 µs ± 3 µs	72 µs ± 25 µs	830 ns ± 449 ns	176 µs ± 38 µs
128	169 µs ± 10 µs	25 µs ± 10 µs	237 µs ± 125 µs	1 µs ± 716 ns	432 µs ± 194 µs
256	312 µs ± 20 µs	47 µs ± 20 µs	911 µs ± 349 µs	2 µs ± 1 µs	1 ms ± 422 µs
512	524 µs ± 24 µs	74 µs ± 24 µs	4 ms ± 2 ms	2 µs ± 618 ns	5 ms ± 2 ms
1024	1 ms ± 33 µs	136 µs ± 33 µs	24 ms ± 8 ms	2 µs ± 2 µs	25 ms ± 8 ms
2048	2 ms ± 129 µs	309 µs ± 129 µs	189 ms ± 59 ms	7 µs ± 1 µs	192 ms ± 59 ms

Table 6: Runtime evaluation data for dense Nauty.

A.3 Nauty (sparse)

Dataset	Setup	Native Setup	Canonization	Other	Total
4	14 µs ± 757 ns	3 µs ± 757 ns	3 µs ± 2 µs	2 µs ± 710 ns	22 µs ± 6 µs
8	19 µs ± 2 µs	4 µs ± 2 µs	3 µs ± 1 µs	2 µs ± 1 µs	28 µs ± 14 µs
16	27 µs ± 2 µs	6 µs ± 2 µs	6 µs ± 3 µs	4 µs ± 29 µs	44 µs ± 34 µs
32	49 µs ± 2 µs	10 µs ± 2 µs	11 µs ± 3 µs	576 ns ± 377 ns	71 µs ± 10 µs
64	111 µs ± 13 µs	24 µs ± 13 µs	25 µs ± 9 µs	876 ns ± 641 ns	161 µs ± 55 µs
128	187 µs ± 12 µs	36 µs ± 12 µs	48 µs ± 16 µs	852 ns ± 453 ns	271 µs ± 73 µs
256	325 µs ± 19 µs	59 µs ± 19 µs	89 µs ± 22 µs	922 ns ± 411 ns	474 µs ± 107 µs
512	633 µs ± 36 µs	111 µs ± 36 µs	207 µs ± 66 µs	1 µs ± 577 ns	952 µs ± 278 µs
1024	1 ms ± 59 µs	210 µs ± 59 µs	536 µs ± 171 µs	2 µs ± 746 ns	2 ms ± 484 µs
2048	2 ms ± 76 µs	348 µs ± 76 µs	1 ms ± 523 µs	1 µs ± 445 ns	4 ms ± 794 µs
4096	5 ms ± 138 µs	714 µs ± 138 µs	5 ms ± 2 ms	3 µs ± 834 ns	10 ms ± 2 ms
8192	10 ms ± 179 µs	1 ms ± 179 µs	33 ms ± 24 ms	5 µs ± 2 µs	44 ms ± 25 ms
16384	23 ms ± 185 µs	3 ms ± 185 µs	275 ms ± 215 ms	9 µs ± 3 µs	301 ms ± 217 ms

Table 7: Runtime evaluation data for sparse Nauty.

A.4 Nishe

Dataset	Setup	Native Setup	Canonization	Other	Total
4	83 µs ± 2 µs	5 µs ± 2 µs	6 µs ± 3 µs	3 µs ± 2 µs	97 µs ± 53 µs
8	64 µs ± 2 µs	6 µs ± 2 µs	8 µs ± 3 µs	2 µs ± 2 µs	81 µs ± 25 µs
16	102 µs ± 7 µs	17 µs ± 7 µs	26 µs ± 10 µs	10 µs ± 54 µs	155 µs ± 88 µs
32	111 µs ± 8 µs	22 µs ± 8 µs	38 µs ± 13 µs	5 µs ± 2 µs	176 µs ± 63 µs
64	140 µs ± 15 µs	41 µs ± 15 µs	78 µs ± 32 µs	7 µs ± 3 µs	266 µs ± 106 µs
128	232 µs ± 34 µs	84 µs ± 34 µs	185 µs ± 71 µs	13 µs ± 5 µs	514 µs ± 193 µs
256	644 µs ± 91 µs	178 µs ± 91 µs	474 µs ± 248 µs	27 µs ± 12 µs	1 ms ± 1 ms
512	610 µs ± 62 µs	261 µs ± 62 µs	1 ms ± 278 µs	46 µs ± 13 µs	2 ms ± 491 µs
1024	1 ms ± 93 µs	478 µs ± 93 µs	3 ms ± 446 µs	115 µs ± 17 µs	5 ms ± 587 µs
2048	3 ms ± 157 µs	772 µs ± 157 µs	10 ms ± 1 ms	195 µs ± 21 µs	13 ms ± 3 ms
4096	7 ms ± 537 µs	2 ms ± 537 µs	43 ms ± 9 ms	499 µs ± 145 µs	52 ms ± 12 ms
8192	10 ms ± 312 µs	3 ms ± 312 µs	159 ms ± 10 ms	827 µs ± 62 µs	173 ms ± 11 ms
16384	23 ms ± 360 µs	7 ms ± 360 µs	707 ms ± 27 ms	2 ms ± 177 µs	738 ms ± 28 ms

Table 8: Runtime evaluation data for Nishe.

A.5 Scott

Dataset	Setup	Native Setup	Canonization	Other	Total
4	1 ms ± 81 µs	238 µs ± 81 µs	24 ms ± 19 ms	51 ms ± 6 ms	77 ms ± 21 ms
8	1 ms ± 60 µs	334 µs ± 60 µs	44 ms ± 30 ms	49 ms ± 4 ms	95 ms ± 30 ms
16	1 ms ± 118 µs	626 µs ± 118 µs	103 ms ± 76 ms	50 ms ± 4 ms	155 ms ± 76 ms
32	1 ms ± 427 µs	1 ms ± 427 µs	344 ms ± 336 ms	51 ms ± 7 ms	397 ms ± 338 ms

Table 9: Runtime evaluation data for Scott.

A.6 Traces

Dataset	Setup	Native Setup	Canonization	Other	Total
4	72 µs ± 5 µs	9 µs ± 5 µs	20 µs ± 10 µs	2 µs ± 937 ns	102 µs ± 61 µs
8	46 µs ± 2 µs	8 µs ± 2 µs	15 µs ± 4 µs	776 ns ± 725 ns	70 µs ± 12 µs
16	124 µs ± 9 µs	22 µs ± 9 µs	29 µs ± 10 µs	6 µs ± 47 µs	181 µs ± 100 µs
32	226 µs ± 17 µs	42 µs ± 17 µs	51 µs ± 19 µs	1 µs ± 742 ns	320 µs ± 114 µs
64	343 µs ± 14 µs	56 µs ± 14 µs	70 µs ± 20 µs	1 µs ± 505 ns	471 µs ± 121 µs
128	618 µs ± 45 µs	101 µs ± 45 µs	118 µs ± 49 µs	1 µs ± 1 µs	838 µs ± 318 µs
256	1 ms ± 23 µs	187 µs ± 23 µs	208 µs ± 36 µs	1 µs ± 319 ns	2 ms ± 187 µs
512	3 ms ± 79 µs	338 µs ± 79 µs	366 µs ± 105 µs	1 µs ± 565 ns	4 ms ± 8 ms
1024	4 ms ± 72 µs	611 µs ± 72 µs	724 µs ± 70 µs	2 µs ± 2 µs	5 ms ± 580 µs
2048	8 ms ± 108 µs	1 ms ± 108 µs	2 ms ± 154 µs	3 µs ± 2 µs	11 ms ± 1 ms
4096	20 ms ± 449 µs	3 ms ± 449 µs	3 ms ± 388 µs	4 µs ± 493 ns	26 ms ± 10 ms
8192	47 ms ± 494 µs	5 ms ± 494 µs	7 ms ± 674 µs	5 µs ± 1 µs	59 ms ± 10 ms
16384	131 ms ± 1 ms	10 ms ± 1 ms	15 ms ± 2 ms	7 µs ± 2 µs	156 ms ± 228 ms

Table 10: Runtime evaluation data for Traces.

B Evaluation Framework Setup

This appendix will detail how to use and obtain a copy of the CPQ Keys evaluation framework codebase. The software is released on GitHub[7] and the README file of the repository will always contain the most up-to-date instructions on using the software. This appendix is written for release 1.0 of CPQ Keys and may not be fully correct for newer releases.

B.1 Getting Started

Running the project can be done either by using docker or by compiling the project from source. Here docker is primarily provided to offer an easy way to run the project without having to worry about any dependencies. Configuring the settings used to run the evaluation should be done by changing the constants in the main class[8].

- Running with docker
- Running from source

B.1.1 Docker

The repository contains a docker configuration that can be used to build an image and run it. The docker image can be built by running the following command:

```
docker build -t cpqkeys .
```

You can then run the created image as follows:

```
docker run --rm cpqkeys
```

B.1.2 From Source

At the moment running the project is only supported on Linux. Docker can be used to run the project on other operating systems. To compile and run the project from source it is first required to install various dependencies. The following dependencies are required:

- **Java 8 or higher**: required to run the framework, make sure to install the JDK and not only a JRE.
- **CMake**: required to compile Bliss, Nauty, Nishe, and Traces.
- **A compiler for C**: e.g. `gcc` is required to compile Nauty and Traces.

[7] `https://github.com/RoanH/CPQKeys`
[8] `https://github.com/RoanH/CPQKeys/blob/master/CPQKeys/src/dev/roanh/cpqkeys/Main.java`

- **A compiler for C++**: e.g. `g++` is required to compile Bliss and Nishe.
- **Python**: python 3 is required to run Scott.

On a system with `apt` the following command can be used to install all dependencies:

```
apt install default-jdk cmake g++ gcc python3
```

After installing all the dependencies the JNI natives for the project can be compiled and installed by going into the CPQKeys folder and running the following script:

```
./compileNatives.sh
```

Finally, the project as a whole can be compiled and executed using Gradle by running the following command:

```
./gradlew run
```

B.2 Development

The repository contains an Eclipse[9] & Gradle[10] project with gMark[11] as the only dependency. Development work can be done using the Eclipse IDE or using any other Gradle compatible IDE. A hosted version of the javadoc for the repository can be found at cpqkeys.docs.roanh.dev[12] and the javadoc for gMark can be found at gmark.docs.roanh.dev[13]. Both documentation pages should be useful when implementing your own algorithms. For compiling and running the project refer to Section B.1.2.

B.2.1 Adding Algorithms

Adding an algorithm to be evaluated is a fairly straightforward process and essentially comes down to implementing the Algorithm interface[14] and adding it to the list of algorithms in the main class[15]. More information can be found in Section 2.6 or one of the existing algorithm implementations. An example template is also given below.

```
1  public class MyAlgorithm{
2      /**
3       * Algorithm instance, using static instances the makes it easier
4       * to have multiple variants of the same algorithm (e.g., see nauty).
5       */
6      public static final Algorithm INSTANCE = new Algorithm("MyAlgorithm", MyAlgorithm::computeCanon);
7
8      /**
9       * Runs the algorithm on the given input graph.
10      * @param input The input graph.
11      * @return An array of time measurements containing in the first
12      *         index the graph transform time, in the second index the
13      *         native setup time (graph construction) and in the third
14      *         index the canonization time. All times are in nanoseconds.
15      */
16     private static long[] computeCanon(Graph<Vertex, Predicate> input){
17         return new long[]{
18             0, //setup time in nanoseconds
19             0, //native setup in nanoseconds
20             0  //canonization time in nanoseconds
21         };
22     }
23 }
```

[9] https://www.eclipse.org/
[10] https://gradle.org/
[11] https://github.com/RoanH/gMark
[12] https://cpqkeys.docs.roanh.dev/
[13] https://gmark.docs.roanh.dev/
[14] https://cpqkeys.docs.roanh.dev/index.html?dev/roanh/cpqkeys/Algorithm.html
[15] https://github.com/RoanH/CPQKeys/blob/master/CPQKeys/src/dev/roanh/cpqkeys/Main.java