NYILATKOZAT

Név: Nagy Szabolcs Ákos

ELTE Természettudományi Kar, szak: Alkalmazott matematikus - mesterképzés

NEPTUN azonosító: LQV1AA

Szakdolgozat címe:
Algorithms for graph canonization

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2025. 06. 01.

a hallgató aláírása

ELTE Eötvös Loránd University

Faculty of Science

Algorithms for graph canonization

Szabolcs Ákos Nagy

Applied mathematics MSc

Supervisor:

Péter Madarasi

Budapest
2025

Contents

Acknowledgement

First and foremost, I would like to express my earnest and sincere gratitude toward Péter Madarasi, the supervising professor for this project. It is through him that this interesting subject was brought to my attention, and this thesis would not have been possible without the never-ending support and patience he provided whenever an issue or question arose throughout our research. This project has given me valuable experience in many ways, including tons of programming advice, a look into the workings of shared development, innovative thinking about problems, and many others, most of which is thanks to Madarasi.

I also wish to express my appreciation of Lóránt Matúz, who worked alongside me on the project during its first year. He contributed a lot to our coding project, and he was extremely amicable and supportive to me throughout the entire process, always eager to help me out with any technical questions that I posed, as well as generally raising morale whenever he could.

1 Introduction

In this paper, we discuss the topic of graph canonization. We introduce the basic concepts necessary to define what it means to canonically represent a graph, and the difficulties we might face during it. We examine the most well-performing graph canonization algorithms, as well as some of the algorithms that led up to them. We also showcase some ideas as to the improvement of these algorithms, and how these improvements perform in practice.

1.1 Motivation

Graph canonization is a tool which can help us solve many mathematical problems related to graph-isomorphism, wherein we want to compare two graphs or subgraphs, and decide the following course of action based on their similarity. In the most simple case, we want to decide whether two given graphs, G_1 and G_2 are isomorphic, that is, whether a permutation on the nodes of G_1 exists such that permuting the vertices of G_1 yields G_2 as a result. A slightly more complex version of the problem is the subgraph-isomorphism problem, in which we want to decide whether a subgraph G_3 of G_1 exists which is isomorphic to G_2. This version of the problem is often more useful in practice and is NP-complete, as one can for example check for a hamiltonian cycle with a polynomial-time subroutine which can decide the subgraph-isomorphism problem.

Such problems often come up in biology [22] when analyzing the structure of proteins or other chemicals, and is also regularly encountered in general graph-theory related theoretical and practical problems. In particular, we will take a closer look at how graph canonization can help us efficiently generate all non-isomorphic graphs, or even just ones which satisfy certain specific properties, something for which no conceptually different algorithm is known. While the canonization of very large graphs is an interesting discussion of its own, we will mostly look at the problem in the context of graph generation, which is mostly restricted to graphs with around 20

vertices even in the best case, as the sheer number of different graphs quickly gets out of hand even with respect to isomorphism, so unless otherwise stated, the graphs discussed in the paper should be thought about as having around 10-15 vertices.

1.2 Our results

For the purposes of testing out new strategies for both graph canonization and non-isomorphic generation, a C++ implementation of one of today's most well-performing canonization algorithms was coded, which while not reaching the sheer efficiency of the best canonization tools available, performs canonization with a runtime of the same magnitude as these programs, and is of a higher level construction, meaning that modifications and utilization is much more convenient than with some of the other available options. This implementation was successfully used both in the development of a non-isomorphic graph generating tool with the collaboration of Lóránt Matúz, as well as in the testing and analysis of various heuristic approaches to improving graph canonization, namely a generalization of Mckay's equitability condition [17] and a new vertex invariant, both of which have been shown to be capable of finding finer partitions than simple equitability refinement.

1.3 Basic definitions, notations

Here, we give a brief overview of terms and notations used throughout the paper. Any notations that are specific just to a given chapter are elaborated on there only.

Sets Given a graph G, $V(G)$ denotes the set of the vertices or nodes of G, and $E(G)$ the set of its edges. We often examine the sizes of sets, this is always denoted as the absolute value symbol, that is, $|X|$ is the number of elements of the set X. When we say a graph is of size n, it means that $|V(G)| = n$. All graph examined throughout the paper are simple unless stated otherwise, and their size is usually denoted by n.

Another type of set we will often use is $\{1, \dots k\}$, that is, for some $k \geq 1$, we take the set of all integers between 1 and k. We will denote this set as $[k]$. For all graphs discussed in the paper, their vertices are assumed to be labeled simply from 1 to $|V(G)|$, that is, $V(G) = [|V(G)|]$. The set of all such graphs of order n is denoted as $\mathcal{G}_n$, and the set of all graphs as $\mathcal{G}_* = \bigcup_{n=0}^{\infty} \mathcal{G}_n$. The set of all subsets of X or its *power set* is denoted as 2^X. For any graph G and $v \in V(G)$, $N(v)$ denotes the neighborhood of v, that is, the set vertices adjacent to v in G.

Permutations In search for a canonical labeling, we will utilize lots of *permutations*, particularly permutations of graph nodes. A permutation μ of the set $[k]$ is simply a bijection $\mu : [k] \mapsto [k]$, that is, each number is "moved" to another, unique number. A permutation μ will be denoted as $(\mu(1), \dots, \mu(k))$, and the set of all permutations of $[k]$ will be denoted as S_k. Given a graph G of size n, and a permutation $\mu \in S_n$, $\mu(G)$ is the resulting graph after μ is applied to the vertices of G, or more precisely, $V(\mu(G)) = V(G)$ and $E(\mu(G)) = \{\mu(u)\mu(v) \ : \ uv \in E(G)\}$.

Generally, given a set of indices $X \subseteq [n]$ and some permutation $\mu \in S_n$, we define $\mu(X)$ as $\{\mu(x) \ : \ x \in X\}$.

If $\mu(G) = H$, then we say that G and H are *isomorphic*, denoted as $G \sim H$ with μ being an *isomorphism* between the two graphs, and when $\mu(G) = G$, we call μ an *automorphism* of G.

If for vertices u, v there is some automorphism μ for which $\mu(u) = v$, then we say that u and v are in one *orbit*. The naturally resulting equivalency classes on the vertices of the graph are likewise referred to as its *orbits*.

Partitions Finally, some notations related to *partitions* need to be mentioned, as partitioning of vertices is a crucial element of more advanced graph canonization algorithms. We say that a set of vertex-sets π is a partition of the set X if the elements of π are disjoint sets made up of the elements of X, and together make up the entirety of X, that is, $c_1 \cap c_2 = \emptyset$ for any two distinct $c_1, c_2 \in \pi$, and $\bigcup_{c \in \pi} c = X$. The elements of π are referred to as the *cells* of the partition, and the elements inside the cells as *atoms*. The set of all partitions of $[n]$ is denoted as Π_n.

The concept of isomorphisms and automorphisms for partitions of the vertices of a graph can be defined analogously to the case of graphs. Given graphs G, H with partitions π_G, π_H respectively, we say that the pair (G, π_G) is isomorphic to (H, π_H) if there is a μ permutation for which $\mu(G) = H$ and $\mu(\pi_G) = \pi_H$, where $\mu(\pi_G)$ is the partition resulting from applying μ to each atom of π_G. When $G = H$, we simply say that the two partitions are isomorphic.

An *ordered partition* is just like a regular partition, except that the cells are also given a specific order. Note that the order of the atoms themselves is still arbitrary within the cells. An ordered partition with cells $X_1, \ldots, X_k$ in that order is denoted as $(X_1, \ldots, X_k)$. The set of all ordered partitions of $[n]$ is denoted as Π_n^*.

We say that a partition π_1 is *finer* than π_2 if every cell of π_2 is a union of cells within π_1. Conversely, π_2 is *coarser* than π_1. When π_1 is finer than π_2, we denote it as $\pi_1 \leq \pi_2$, and if they are not equal, we write $\pi_1 < \pi_2$.

2 Graph canonization

In this section, we introduce the concepts related to canonical graph representation, and discuss some of the most basic ideas and approaches that are necessary to properly construct an algorithm which consistently gives us the same representative for any given isomorphism class.

2.1 Canonical representation of a graph

To put it simply, graph canonization is the process of taking a graph, and computing from it some completely isomorphism-invariant output.

Definition. For any target set U, A function $f : \mathcal{G}_n \mapsto U$, is called a *canonization* algorithm if, for any two graphs $G, H \in \mathcal{G}_n$, we have $f(G) = f(H)$ exactly when $G \sim H$.

This function is also sometimes called a *complete isomorphism-invariant*, as the same f-value is sufficient and necessary for the isomorphism of two graphs. Here, $f(G)$ is called the *canonical representation* or *form* of G according to f.

Different canonizations can give the same graph completely different canonical forms. For two canonizations f_1, f_2, even if $R(f_1) = R(f_2)$, it is perfectly plausible to have $f_1(G) \neq f_2(G)$ for some $G \in \mathcal{G}_n$. In this case, G simply has different canonical forms for the different canonizations.

It is not difficult to see that given access to an oracle for such a function, one can easily give a method for deciding the graph isomorphism problem: with an oracle to an f canonization, to check whether $G \sim H$ holds, we simply need to see whether $f(G) = f(H)$ holds.

One simple example of such a function would be the following: for a graph G, let C_G be the set of all graphs isomorphic to G. Then, defining our function as $f : G \mapsto C_G$, we can confirm that f is indeed a canonization, as $C_G = C_H$ will clearly hold exactly if G and H are in the same isomorphism class. This approach works only in theory of course, since a graph of size n can have as many as $n!$ graphs isomorphic to it, a sharp estimate in the case of graphs without any automorphisms, so generating C_G every time a graph needs to be canonized is not feasible.

The reach of the canonization is usually chosen to contain fairly simplistic and compact elements, as easy comparisons of canonical forms is commonly performed during isomorphism-testing. Some canonizations merely assign a string of characters to the graph, others build up a proper data structure to represent the graph. We will be taking a particular look at a specific method of canonically representing a graph which provides great convenience and usefulness in various ways for certain methods that utilize canonical representations, that being canonization by vertex re-labeling.

2.2 Canonical labeling

The idea of computing the C_G isomorphism class is not a very efficient solution to the canonization problem, but one may think to compute just one specific element of C_G. Suppose we have two graphs, G and H, and we also had their respective isomorphism classes, C_G and C_H. Then if we wanted to know whether $C_G = C_H$ holds, it would be sufficient to decide whether there is any common element $c \in C_G \cap C_H$ between the two sets, as according the nature of isomorphism, if they have one element in common, then they must be exactly the same set. If there was a way to calculate to compute the c graph, given only G, or any other graph in C_G without actually having to examine the entire isomorphism class, then this c graph would be a more practical canonical form for G.

Thus, we can introduce the concept of canonical labeling, the most widely-used method of practical graph canonization.

Definition. Let a canonization $f : \mathcal{G}_n \mapsto \mathcal{G}_n$ be such that for any $G \in \mathcal{G}_n$, $G \sim f(G)$ holds. In this case, $f(G)$ is called the *canonical labeling* of G according to f.

One can imagine the application of f as simply permuting the vertices of G, re-labeling them to gain the new form, hence the name. In fact, sifting through

various permutations on the vertices of G is exactly what the main canonization algorithms do. Once again, the canonical labeling of a graph is only "canonical" as far as the canonization goes, different appropriate canonization functions can assign different canonical labelings to the same graph.

This particular type of graph canonization is remarkably interesting because in many cases, we do not only want to identify the isomorphism class of a graph, but we might want to perform operations on it in ways that are consistent across different runs that are started with isomorphic input graphs, which the ability to canonically label a graph provides a solution for straight away. For instance, in Section 4, we discuss a non-isomorphic graph generating algorithm whose main ideas include examining what properties certain vertices would hold according to some canonical labeling.

No algorithm for computing a canonical labeling in polynomial time is currently known. A big hurdle in the calculation of a canonical labeling, or canonical representations in general, is that the output simply cannot, in any way rely on the "order" in which the vertices of the graph are labeled. For example, a simple approach such as running a depth-first search on the graph, then labeling the vertices in order of when the search was done examining them is entirely wrong, as this order is dependent completely on the initial labeling of the nodes, so any element of C_G can yield a completely different result, despite the graphs themselves being isomorphic. Throughout our search for a canonical labeling, it is important to proceed in an isomorphism-invariant manner, thus, whenever a decision must be made in relation to choosing between multiple vertices with no given order other than their labels, we will have to consider each possibility individually, which can lead to widely branching recursions, so efforts should be made to keep these decisions as few and as far in-between as possible.

2.3 Historical overview

The graph isomorphism problem and its related problems have been subject to exhaustive research, with a wide array of different approaches and heuristics having been used or attempted to tackle the problem. To this day, no polynomial-time algorithm is known that can decide whether two arbitrary graphs are isomorphic, but many approaches have been found that provide perfectly satisfactory results in the vast majority of cases. One such approach was through graph canonization.

The idea of using canonical forms and labelings to decide isomorphism and similarity between graphs can be traced back to the 1960's, when people such as Morgan [21] or Duijvestijn [11] used the ideas of canonical coding to identify chemical compounds, and to aid in the computation of squared rectangles, respectively.

Since then, many minds have dwelled on the canonical representation of a graph. For example, Read [25] and Babai [5, 6, 4] have studied algebraic, string-focused canonical representations, the former having introduced a linear-time algorithm for trees, and the latter having introduced, among other things, an algorithm of expected linear time on a majority of graphs, and, in 2019, a quasi-polynomial algorithm for the general case.

However, the essence of most practical canonization lies in the search-tree based

canonical labeling algorithms, first introduced and studied by Parris and Read [26], Corneil and Gotlieb [9] or Arlazalov [2], then later, independently fleshed out and popularized by McKay [20], who greatly improved the performance of the algorithm on graphs of high automorphism count. The algorithm of McKay has since been extensively chiseled and perfected, and still to this day remains one of the leading tools in canonization in the form of McKay's and Piperno's `nauty` and `Traces` [17, 19] project. Other tools have since been developed that utilize the same general paradigm provided by this algorithm, Such as Junttila's `bliss` [13] project, or Darga's `saucy`[10] and many more, each of which aim to improve upon the general algorithm with certain divergences that improve performance in a select group of cases.

Even after years of study, there are still open questions regarding graph canonization. One of the main undecided questions regarding the complexity of graph canonization is whether deciding the graph-isomorphism problem is polynomial-time equivalent [3] to the computation of a canonical form. Clearly, given a canonization oracle, solving the isomorphism problem is trivial, but it is not yet known whether an isomorphism oracle could allow us to compute a canonical form for any graph in polynomial time.

3 Canonization Algorithms

In this section, we will go over and examine various notable algorithms which employ various methods and strategies to compute a canonical representation for a graph, and potentially a canonical labeling. Among others, we also introduce the basics of search-tree based canonization, which is especially important for the ideas listed in Sections 4 and 5.

3.1 Read's algorithm for trees

While no polynomial-time canonization algorithm is yet known for general graphs, one can often "simplify" the problem by only examining specific families of graphs. Trees (connected graphs without any cycles in them) are much more easily handled in many problems due to their specific properties. Isomorphism can be checked between trees in polynomial time and similarly, canonization is also feasible. The following method of tree-canonization by Ronald C. Read [25] computes a binary string (a string of zeroes and ones) for each vertex by iteratively concatenating the strings of various vertices. The original algorithm involves the conversion of the original tree into a rooted binary tree, which allows the algorithm to run in linear time and memory. Here we will discuss a simpler version of the algorithm, which still runs in polynomial time.

Initially, all vertices are given the string "01". Then, in what we call the concatenation cycle, for every non-leaf vertex v, the `Concatenate` subroutine removes the leading 0 and trailing 1 from $s(v)$, effectively "opening up" the string for the current concatenation. Then it takes all the leaf neighbors $L \subseteq V(T)$ of v, and then all $s(v)$, as well as all $s(l) : l \in L$ are concatenated together in lexicographic order. It then adds the leading 0 and trailing 1 to the new, concatenated string, effectively

Algorithm 1 Read's tree-canonization

```
procedure CanonTree(T)
    for all v ∈ V(T) do
        s(v) ← "01"
    end for
    while |V(T)| > 2 do
        for all v ∈ V(T) : d(v) > 1 do
            Concatenate(T, v)
        end for
        for all v ∈ V(T) : d(v) = 1 do
            T ← T − v
        end for
    end while
    if |V(T)| = 2 then
        Merge(T)
    end if
    z ∈ V(T)
    return s(z)
end procedure
```

"closing down" the collected neighboring strings. Clearly, the string of a vertex only changes if it has at least one neighboring leaf. Once we have done this for all non-leaf vertices, the current concatenation cycle ends, and we delete all leaves from the graph, finalizing their respective binary strings, and the concatenation cycle begins once again. We do this until we have only one vertex left. In the case where there are two vertices left at the start of an iteration step, we simply call the Merge subroutine, in which we treat one with a lexicographically maximal string as the "non-leaf", and finish in one similar concatenation step.

Since we are dealing with trees, at least two vertices are guaranteed to be removed in each iteration. It can be shown that this method gives us a canonical representation of the tree in the form of the binary string belonging to the final remaining vertex z.

Theorem 3.1. Within the family of tree graphs, CanonTree(T) given by Algorithm 1 is a canonization.

The most simple way to think about how this final string is a representation of the tree is to consider the it as a representation of a laminar system of branches, with the final vertex being the "root" set. Zeroes open up a collection of subtrees, and ones close it off. Thus, the input tree itself can be easily reconstructed from $s(z)$, and is therefore only obtained by trees isomorphic to the original input.

In Figure 4, we show how the algorithm runs on a simple tree. First, the only non-leaf vertices with leaf neighbors are v_3 and v_6, the former receives the new string $0 + 01 + 01 + 1$, the latter $0 + 01 + 1$. Afterwards, the vertices v_1, v_2 and v_4 each with the label 01. Then v_3 and v_6 both become leaves, meaning that v_5 now has leaf neighbors. In the next cycle, Now v_5 receives the new string $0 + 001011 + 0011$

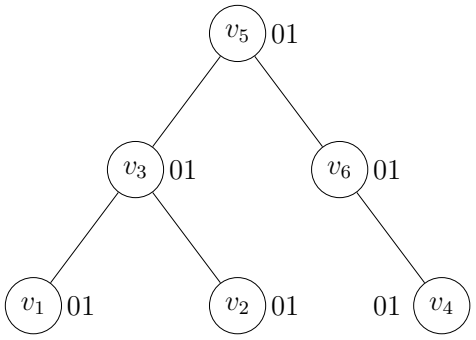


Figure 1: Initial labels.

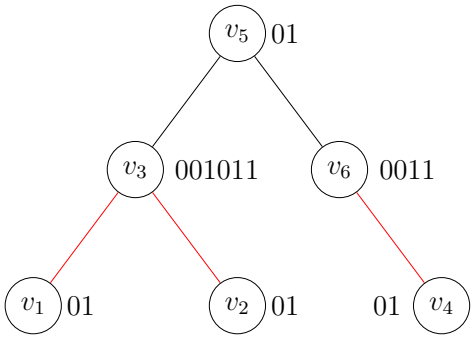


Figure 2: v_1, v_2, v_4 finalized.

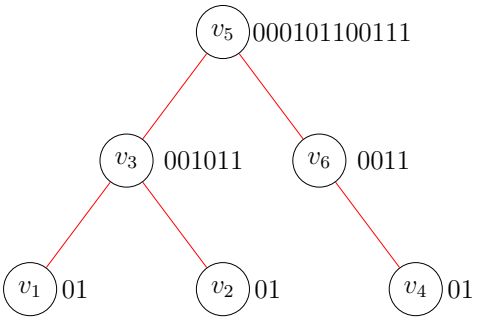


Figure 3: Rest of vertices finalized.

Figure 4: An example of resulting binary strings, first the bottom three vertices are removed, then the one in the middle, v_5 is the final vertex.

+ 1, and both of the contributing leaves are subsequently deleted. v_5 is the last remaining vertex, so the algorithm terminates, and the canonical representative of our tree is "000101100111".

Note how in this version of the algorithm, certain vertices may need to have their corresponding binary strings altered multiple times throughout the run of the algorithm, whenever they receive a new neighboring leaf at the start of a concatenation cycle. Read details a version of the algorithm in which each vertex only needs to be examined once and which still leads to a canonical form, however, it is a lot more difficult to see the isomorphism-invariance of the process, and the final binary strings may differ greatly between the corresponding vertices of isomorphic graphs. In the above version, any two isomorphic trees will have the same binary strings on vertices that correspond to one another for some isomorphism between the graphs, which is beneficial for establishing a canonical labeling.

The canonical labeling While the string of each vertex is clearly a canonical representation of the given subtree branching from the vertex, it is also clear that certain vertices will have the same final string at the end of the algorithm. All leaves, for instance, will have the initial "01" string as their final one set in the very first concatenation cycle. This means that if we want an actual canonical labeling on the vertices, we cannot simply label them from 1 to n based on lexicographic order of the strings alone, as there is no straightforward way to handle ties. However, it is not particularly difficult to acquire a proper labeling from the strings given by

the above algorithm in polynomial time. One such method would be the following: First, order the different resulting string in lexicographic order, then give each family of vertices with identical strings an interval of labels based on their positions in the established order. More precisely, if the resulting different binary strings in order are $b_1, b_2, \ldots, b_k$, and there are $p_1, p_2, \ldots, p_k$ vertices with these strings respectively, then the vertices possessing b_1 will receive the labels $1, 2, \ldots, p_1$, the ones possessing b_2 will receive $p_1 + 1, p_1 + 2, \ldots, p_1 + p_2$, and so on.

Now for the actual distribution of labels: find a central node z in the tree T, one for which $\max\{\text{dist}(v, z) \ : \ v \in V(T)\}$ is minimal (the final vertex of the above algorithm is appropriate for this), and then run a depth-first search on the tree starting from there. Every time a new vertex is encountered throughout the search, we give it the smallest label within established interval of its binary string that we have not yet given out. It is not difficult to show that given the total isomorphism-invariance of the strings, the resulting labeling will indeed be canonical. Once again we have to be careful of the special case where there are two central nodes in the graph, but as long as we make sure to step into this other node last when examining the neighbors of z, the algorithm will give us consistent labelings.

3.2 Generalized vertex-hashing

The efficiency of the algorithm detailed in the previous section relies heavily on the restriction of tree-graphs. Both the low number of concatenation-cycles and the simple acquisition of a canonical labeling are made simple due to the simplistic structure of the graph, however, the general idea of iteratively modifying binary strings based on those of neighboring nodes can be applied to general graphs to a certain extent.

Jüttner and Madarasi [14] proposed an isomorphism-testing algorithm that in implementation operates in a similar vein to Read's algorithm in the sense that labels are computed on the vertices of graphs by concatenating certain strings. It attempts to categorize vertices based on the different number of unique walks between each other by means of dynamic programming, collecting information from neighbors in a similar way to Read's algorithm. Here we present it in a manner that shows us how the algorithms are similar. The process consists of the repetition of a similar concatenation cycle as the one performed by Algorithm 1, except that no consistent way of "finalizing" labels can be relied on as with leaves in the case of trees, so instead of working back from leaves, every vertex simply aggregates the labels of its neighbors in every concatenation cycle. As a result of this, in order to stay isomorphism-invariant, vertex-labels need to be modified simultaneously: if we were to continually change labels as they are computed like in the previous algorithm, it would immediately begin to affect other labels, which would make the end result highly reliant in the order of the examination of the vertices. Instead, all new labels are applied only once the entire concatenation cycle is finished.

Another luxury utilized by Read's algorithm is the convenience of the laminar method of labeling provided by the subtrees encoded within the canonical form. By containing each concatenated group of strings within zeroes and ones, an efficient way of recreating the input tree from the output string can be defined, which also

proves the correctness of the algorithm. In the general case however, connections between vertices can be much more complex, and thus such a structure would not be attained from this method of concatenating labels. Instead, the strings themselves can simply be concatenated without the closing-opening steps, however, in order to make the strings of vertices in different orbits stand out more from one another, labels are calculated individually for each group of vertices, with unique starting labels to help the specific "environment" of each vertex shape its eventual label on a case-by-case basis.

This is achieved by computing characterizations of G based on the individualization of certain sets of distinct vertices $V' = \{v_1, v_2, \dots v_k\} \subseteq V(G)$. From now on, k represents the number of individualized vertices. Certain multisets, here represented as strings, are initially computed which aim to describe the walks of length at most l which begin in V' with the following method:

Algorithm 2 Hash-labeling

```
procedure HashLabel(G, V', l)
    for all u ∈ V(G) do
        s_0^k(u) ← "(∅, Neighbors(u, V'))"
    end for
    for all i ∈ [l] do
        for all u ∈ V(G) do
            s_i^k(u) ← (s_{i-1}^k, Aggregate(N(u), i − 1))
        end for
    end for
     return Aggregate(V(G), l)
end procedure
```

Here, $\textsc{Neighbors}(u, V')$ is string comprised of $k - d(u, V')$ zeroes followed by $d(u, V')$ ones, essentially concatenating the indicators of the edge-relations between u and the elements of V'. $\textsc{Aggregate}(X, i)$ on the other hand takes the $s_i^k(.)$ strings of all vertices within X and concatenates them in lexicographic order. The strings computed in the final cycle are all taken together, and used to give the "final" walk-characterization of V' on G, hereby referred to as $s_G^k(V')$.

Clearly, for any vertex-subsets of size k, if $s_G^k(V_1) \neq s_G^k(V_2)$, there can be no automorphism μ of G for which $\mu(V_1) = V_2$. In [14], it is shown that no "useful" information is gained after $n+1$ concatenation cycles, that is, if $s_G^k(V_1) = s_G^k(V_2)$ for $l = n + 1$, then the same follows for any $l > n + 1$ too. From here on, l is assumed to be $n + 1$.

A drawback to this approach may become apparent at this point, regarding the length of the individual strings. Initially, each string describes only describes the relation the vertex holds to the individualized vertices, so $|s_0^k(v)| = \mathcal{O}(k)$ for all $v \in V(G)$. However, each vertex can have up to $n - 1$ neighbors, so with each concatenation cycle, the length of each string can increase at most n-fold. $|s_G^k(V)| = \mathcal{O}(k * n^l)$. If we want to make multiple iterations, then this is clearly going to become too long to reasonably compute.

One method of circumventing this issue is by replacing the strings with shorter

representatives after each concatenation cycle to keep things on a smaller, more manageable level while still preserving the unique information contained within the aggregated strings. In Jüttner's and Madarasi's implementation, this is achieved by simply passing the strings through a hash function (namely `SHA512`), which replaced each label with an "arbitrary" string of consistent length, for which any two non-identical strings are highly unlikely to receive identical hash-values, this way, strings can still provide useful differentiating information throughout aggregations, while staying at a manageable length.

So far we have shown how a label can be computed for each vertex, now as for the representation of the graph: We wish to derive a representation of the entire graph, not just in relation to a given vertex-set of size k. This can be achieved by a recursive formula (which in practice is of course computed via dynamic programming):

Definition. For some $q \in [k-1]$ and vertex-set $V' = \{v_1, \ldots, v_2\} \subseteq V(G)$, let $s^k_G(V') := \textsc{Concatenate}\{s^k_G(V' \cup \{v\}) \ : \ v \notin V'\}$.

Here, $\textsc{Concatenate}(X)$ simply concatenates together all the strings within X in lexicographic order. This way, for $q = 0$, we get an absolute representation of the entire graph, Which is simply denoted as s^k_G. It can be shown that following the previously described methods, s^k_G can eb computed in polynomial time, assuming that k is a bounded constant.

Identifying graphs It is not difficult to see that for isomorphic graphs, this representation will indeed be the same, however it is not a complete isomorphism-invariant. It is proven in [14] that connected strongly regular graphs with the same parameters all have the same $s^1_\cdot$ value, and that a violating pair must exist for $k = 2$ as well. as well. This is why $s^k_\cdot$ is referred to as a *fingerprint* of the input graph, rather than a canonical form. This way of representing the graph however is still very much an identifier for many intents and purposes. In the same paper, it is shown that this fingerprint uniquely identifies and differentiates graphs in many scenarios. For instance, it was determined through exhaustive examination that $s^2_\cdot$ is a complete isomorphism-invariant on $\mathcal{G}_{12}$, that is, for any two graph G, H of size at most 12, $s^2_G = s^2_H$ exactly if the two graphs are isomorphic. The following are a few more of the most notable results regarding the efficacy of this graph fingerprint.

Theorem 3.2. $s^1_\cdot$ is a complete isomorphism-invariant on trees.

Theorem 3.3. $s^3_\cdot$ is a complete isomorphism-invariant on 3-connected planar graphs.

Theorem 3.4. If G is k-connected and H is not, then $s^k_G \neq s^k_H$

We will also examine in Section 3.5 how in the case of $k = 1$, the different $s^1(v)$ labels computed for the vertices themselves in a slightly modified version of the above algorithm can help us achieve a canonical labeling, and how this method relates to a more practical method of graph canonization.

3.3 Search-tree algorithms

In this section, we discuss one of the most widely used methods of graph canonization, in which a search-tree of ordered partitions is used to sift through various permutations of the vertices of a graph.

Let us take a step back and examine how one could pick out a labeled graph to represent a given isomorphism class when a reasonable runtime is not a concern. If we could define a order on the labeled graphs belonging to this isomorphism class, then we could choose the first graph in the order to be the canonical representation. With this approach, we can give a very simple canonization algorithm, as the isomorphism class of any given input graph G of size n can be computed by simply permuting the vertices of the graph in all possible ways, and adding the resulting graph to our (initially empty) collection. It is not even an issue if different permutations lead to the same graph, as the lexicographically minimal representative will be found and kept either way. This brute-force approach clearly gives us a proper canonical representative in the form of a canonical labeling, but is also completely unfeasible in practice, simply because of the sheer amount of labeled graphs that need to be examined, $n!$.

Search-tree canonization algorithms essentially attempt to take the above approach and make it a bit more friendly with certain heuristics and reductions. The main goal is to reduce the number of permutations that actually need to be examined in the above method, while still guaranteeing that the chosen graph is indeed a lexicographically minimal re-labeling of the initial graph according to some ordering.

Definition. A *canonization search-tree* of the graph G is the pair formed by a rooted tree $T = (V_T, F)$ and a collection of ordered partitions $\{\pi_t : t \in V_T\}$ such that:

- for the root $r \in V_T$, we have the unit partition $\pi_r = (V(G))$,
- for any $t \in V_T \setminus \{r\}$, we have: $\pi_t \leq \pi_{p(t)}$, where p is the parent function within T,
- for any leaf $l \in V_T$, we have an ordered trivial partition $\pi_l = (\{\mu_1\}, \{\mu_2\}, \ldots, \{\mu_n\})$ for some permutation $\mu \in S_n$.

Clearly, for each leaf of such a tree, we can apply the permutation corresponding to its partition to G, giving us a new labeling of the graph. However, in search-tree algorithms, we usually apply the inverse of the above permutation, for reasons which should become clear once the nature of these algorithms is more closely examined. See Figure 5 for an example of how an ordered trivial partition translates to a permutation of $V(G)$.

Now we can properly introduce the core idea of search-tree canonization algorithms: if we can generate a canonization search-tree such that the leaves of the tree yield an identical collection of graph forms for different, yet isomorphic starting graphs, then it is sufficient to search for a lexicographically minimal form amongst those.

Clearly, the more children we give to each partition, the wider the search-tree becomes, and the more permutations we need to examine. Excessive calculations are generally not ideal, so we want to minimize the number of children that we give

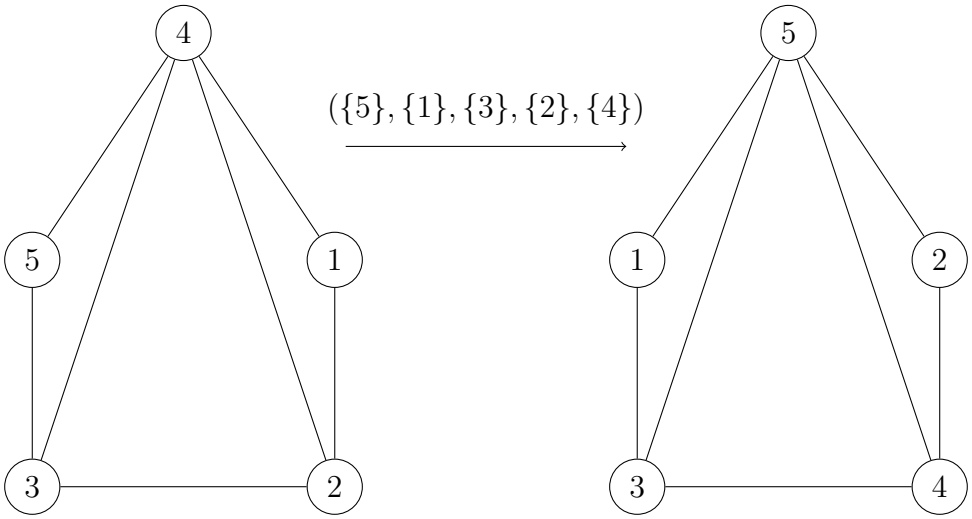


Figure 5: Permuting vertices based on an ordered trivial partition.

to each node, that is, make the partition of each child as fine as possible compared to its parent, while still guaranteeing that the resulting permutations give us the lexicographically smallest form. To achieve this, our best tool is to use certain graph properties that can be calculated somewhat efficiently, but can also help us tell apart "different" vertices from one another, and preferably gives us a straightforward way of ordering the different vertex-groups, giving us a way of determining how to make the children ordered partitions as fine as possible.

How to choose children partitions The general idea is the following: we start out with the ordered partition π of the current node, and we calculate an isomorphism-invariant vertex-function on the graph with the given partition.

Definition. We say that a function $g : \mathcal{G}_n \times \Pi_n^* \times V(G) \mapsto U$ with arbitrary U target set is *isomorphism-invariant* if for any input $G \in \mathcal{G}_n, \pi \in \Pi_n^*, v \in V(G)$ and permutation $\mu \in S_n$, we have $g(G, \pi, v) = g(\mu(G), \mu(\pi), \mu(v))$.

Let us assume that we have an isomorphism-invariant function g, and its reach U is a naturally ordered set, such as $\mathbb{R}$. If two vertices, u and v are within one cell of π such that $g(G, \pi, u) < g(G, \pi, v)$, we can separate them based on $g(G, \pi, .)$-value. In fact, we can split up the entire W cell containing u and v: the elements of W are put into new cells, which take the place of W in the ordered partition. If $\pi = (X_1, X_2, \dots, X_k, W, X_{k+2}, \dots)$, then $\pi' = (X_1, X_2, \dots, X_k, W_1, W_2, \dots, W_l, X_{k+2}, \dots)$, where W_1 contains the elements of W with minimal $g(G, \pi, .)$-value, W_2 those with the second-smallest, and so on. We can clearly do this for any cell in which there are atoms for which the $g(G, \pi, .)$-value differs.

Once we have a partition π^* in this manner for which vertices in one cell always have the same $g(G, \pi^*, .)$-value, we say that $\pi*$ is the *g-refined* version of the partition π. At this point, if π^* is trivial, we can simply give the search-tree node of π a single child containing π^*. Otherwise, we cannot efficiently differentiate between the different vertices with g, so we must resort to more naive methods of vertex identification, in the form of *individualization*. We choose a non-trivial cell W from

π^* in an isomorphism-invariant way, such as simply choosing the earliest one in the order, and we can give a child partition to the node of π for each element x of W, in which W is replaced by $\{x\}, W \setminus \{x\}$ in the partition.

Thus, given some refining procedure g, we can construct our search-tree by the aptly named *refinement-individualization* paradigm:

Algorithm 3 Refinement-individualization search-tree

```
procedure CANON(G)
    π ← {V(G)}
    CANONREC(G, π)
end procedure
procedure CANONREC(G, π)
    REFINE_g(G, π)
    if |π| = |V(G)| then
        LEAF(G, π)
    else
        X ← CHOOSECELL(G, π)
        for all x ∈ X do
            π' ← IND(π, x)
            CANONREC(G, π')
        end for
    end if
end procedure
```

Here, the $\textsc{Refine}_g$, Leaf, ChooseCell and Ind subroutines behave in the way previously described:

- $\textsc{Refine}_g$ is an isomorphism-invariant subroutine which transforms the input partition π into a g-refined version of it.
- Leaf Examines the graph form given by the permutation gained from the ordered trivial partition π, and decides whether it is lexicographically smaller than the current minimal form, and replaces it if so.
- ChooseCell is an isomorphism-invariant subroutine which chooses a non-trivial cell of the input partition π.
- Ind Replaces the cell W of x within π with $\{x\}, W \setminus \{x\}$.

Ideally, g should be able to tell apart and sort different vertices while still being able to be calculated for every node of the search tree without problem. Notice that if g is a constant, then we only ever refine our partitions through individualization, and our search-tree will end up performing the same amount of graph-comparisons as the brute-force method, as the leaves of the tree will just be all possible permutations of $V(G)$, see Figure 6 for an example. In the figure, only the ordered partitions of the two leaves on the side are written for the sake of brevity.

It is also important to point out that in practice, this "search-tree" is never property constructed in its entirety, and is instead traversed through recursion in a

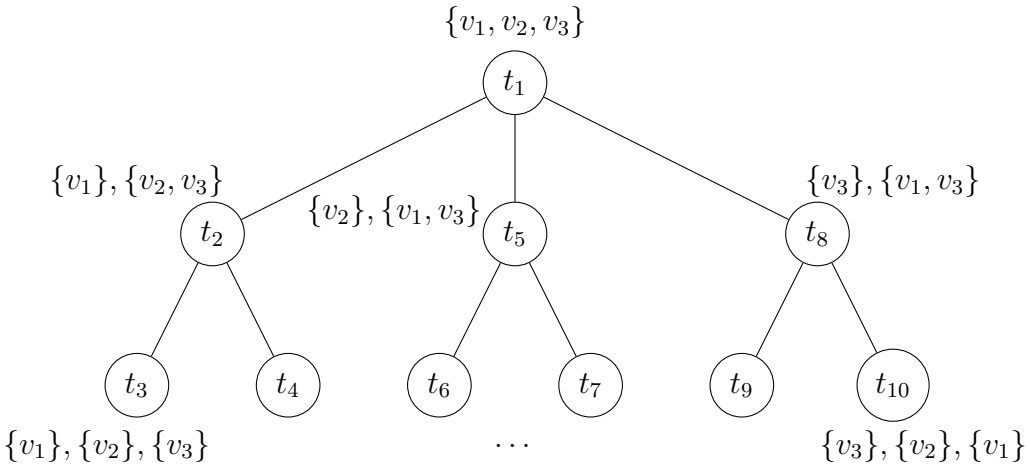


Figure 6: A search-tree representation of the brute-force method.

manner similar to a depth-first search on such a search-tree, storing only information that is relevant in tracking the lexicographically smallest form of the input graph. It is referred to as a search-tree simply to make it easier to visualize and imagine how the methods and applications take advantage of the order in which partitions are examined.

3.4 McKay's algorithm

Brendan McKay [17], a notable figure in the development of search-tree-based canonization, introduced one of the most practical g-functions in the form of *equitable* partitions, and in addition to the tactics above also uses automorphisms found throughout the run of the algorithm to prune certain parts of the search-tree when different branches are noticed to be "similar". This is especially useful when the input graph has a large amount of automorphisms, as such graphs are usually notoriously difficult to refine using g-functions in the manner described above, as many vertices that are not even in one orbit can act similar to one another in many ways, so their g_π values will be the same for even relatively fine π ordered partitions, so more individualization steps need to be taken than with asymmetrical graphs.

3.4.1 Equitable partitions

Most isomorphism-invariant calculations take great advantage of certain degree and distance-related information, as these relations and properties stay constant when the graph is permuted. When permuting the vertices of a graph, the degree of a given node will be the degree of its original image, no matter how we calculate it. McKay's algorithm uses the idea of a generalized "degree" property that also takes great advantage of the partition that we already have at whatever place in the search-tree we may be.

Definition. Given a graph G, we say that a partition π of $V(G)$ is *equitable* when for any two, possibly equal cells $X, Y \in \pi$, there exists $k \in \mathbb{N}$ such that for any $x \in X$ we have $d(x, Y) = k$.

To put it simply, the elements of any given cell in an equitable partition have the same number of neighbors in every cell. This gives us a suitable g-function to refine our ordered partition π: for each vertex $v \in V(G)$, we calculate how many neighbors v has in each cell of π. Since the cells themselves have proper order, this gives us an ordered array of $|\pi|$ integers, which gives us a straightforward way of lexicographically sorting the vertices among the cells of π.

Note that if this refinement is "successful," that is, at least one cell was split into subsets due to different vertices having different amounts of neighbors in certain cells, the resulting partition is not yet guaranteed to be equitable. Let us say, for example, we have an ordered partition $\pi = (X_1, X_2, \dots X_k)$ in which $u, v \in X_1$ violate the equitability condition: $d(u, X_2) < d(v, X_2)$, but there are otherwise no bad cell pairs. We split the elements of X_1 based on the size of their neighborhoods in X_2, resulting in $\pi' = (Y_1, Y_2, X_2, \dots X_k)$. Now, for every vertex who had neighborhoods in X_1 have those neighbors split among Y_1 and Y_2, and it is entirely possible that this ruins their uniform degrees toward the subsets. Say, for example, that vertices in X_3 all had one neighbor in X_1. Now each vertex $z \in X_3$ that had its neighbor in Y_1 will have $d(z, Y_1) = 1$, and those that had it in Y_2 will have $d(z, Y_1) = 0$. Clearly, equitability needs to be reevaluated whenever we split cells, and only stopped once all pairs of cells are confirmed to abide by the equitability condition. The most basic version of equitability refinement would therefore be the following: Check for every pair of cells $X, Y \in \pi$ whether any two vertices violate the condition. If so, split elements of X by their degree in Y. The resulting subsets of X are added to the list of cells that need to be checked for violating pairs. This is done until we run out of cell-pairs that need to be examined, at which point we can conclude that the partition is equitable.

Efficient equitability computation It is not difficult to verify that indeed, given isomorphic input graphs, the above method will always give us isomorphic equitable ordered partitions when using the above method, but the calculations, despite being polynomial, are somewhat excessive, as we might calculate a lot of "useless" data if we carelessly calculate the cell-degrees of every vertex every time a cell is split. Say we have $\pi = (X_1, X_2, X_3, X_4)$. We notice that X_1 and X_2 violate the equitability condition, and we get $(Y_1, Y_2, X_2, X_3, X_4)$ after the split. Notice how X_2, X_3, X_4 remain unaffected, so the cell-degrees of their vertices towards one another remain the same, we do not need to calculate them. We can use McKay's classic theorem [17] to make the proving of correctness a little simpler:

Theorem 3.5. For any graph G and any partition π of $V(G)$, there exists a unique coarsest equitable partition π^* for which $\pi^* < \pi$.

This means that as long as we can choose the order of the cells of π^* in an isomorphism-invariant manner, we can reach this partition consistently among isomorphic graphs more efficiently, so long as we always make cell-splits that are clearly necessary to ensure equitability. Thus comes the idea of active cells: we want to mark which cells are interesting to us in terms of whether they could potentially violate the equitability condition, and only calculate the cell-degrees of the vertices of G toward these cells. This is simply achieved by marking cells of π *active* whenever

they are split into subsets. Whenever we split a cell, whether through individualization or because of refinement, the resulting subsets are all marked as active. Now, we only need to examine the degrees of vertices toward active cells. We take an active cell Y, and then compare with every cell in the graph. We go cell by cell, and whenever we notice that some cell X has vertices with different degrees in Y, we split X right away, marking the resulting subsets as active, and continuing with examining neighborhoods in Y. If it is concluded that all remaining cells satisfy the equitability condition with Y, then we can mark Y as no longer active, and we can do the same check for the next active cell. It can naturally be inferred that once there are no longer any active cells, our partition is equitable, and we can move on to individualization.

3.4.2 Automorphism pruning

While equitability-based refinement is usually very efficient in practice, it does not always solve the issue posed by very symmetrical graphs, that is, graphs with large automorphism groups. Take the simple case of K_n, the complete graph. Clearly, all vertices "behave" in the exact same way in any scenario, and any partition on its vertices is trivially equitable. This means that the only way for the algorithm so far to make progress is through individualization steps, which results in $n!$ leaves, and all possible permutations will need to be checked, despite the fact that they all result in the one and only form that labelings of K_n can take. To combat this, McKay introduces the idea of pruning branches of the search-tree which are guaranteed to yield leaves that have permutations resulting in graphs that some earlier branch has already provided.

Let us imagine that in our search tree, there are two nodes a and b with ordered partitions π_a and π_b that are isomorphic. Consider the subtrees below a and b. Because of the isomorphism-invariant manner in which we construct our search-tree, the entire subtree must be isomorphic as well. We always individualize cells that correspond to one another, and refinement will result in isomorphic partitions as well. Clearly, isomorphic ordered trivial partitions give us permutations that result in identical graph-forms, so it would be sufficient to examine only the subtree below a in order to find our canonical form. Unfortunately, no efficient method for determining whether two ordered partitions are isomorphic or not is currently known, and even if such a subroutine existed, comparing the currently examined partition to every single one that came before which it could potentially be isomorphic to could be troublesome. Instead, McKay's algorithm finds automorphisms as the search-tree is traversed, and examines how they relate to the current partition with each individualization step.

Finding automorphisms McKay's method of finding automorphisms on G is by utilizing the graph-forms given by the leaves as they are examined. Once the first leaf t_0 of the tree is reached, we check what labeled graph its permutation μ_0 leads to, and store the graph $\mu_0(G)$ as our basis for checking automorphisms. Then, for every subsequent leaf, we check whether $\mu(G) = \mu_0(G)$ holds for the permutation μ yielded from their ordered trivial partition. If so, then clearly, $\mu_0 \circ \mu^{-1}$ is an automorphism

of G: $\mu^{-1}(\mu_0(G)) = \mu^{-1}(\mu(G)) = G$. In fact, it is shown by McKay [17] that any automorphism in the graph can be obtained this way:

Theorem 3.6. Given the search-tree T produced by equitability refinement without any further pruning. For any $\mu \in \mathrm{Aut}(G)$, if there is a node in T whose ordered partition is π, there is also a node whose partition is $\mu(\pi)$.

One might think to compare each found graph to not just the initially examined graph, but to all graphs that were examined up until this point in order to find automorphisms that we might potentially miss otherwise, but this is not a beneficial idea for various reasons. Firstly, in order to make all of the necessary comparisons, we would need to store at least all the permutations inferred from the leaves of the search-tree, which, given that there is no polynomial bound on the size of the search-tree, could make the memory requirements infeasible. Furthermore, the sheer number of comparisons could square our already non-polynomial runtime, which would defeat the entire purpose of finding automorphisms in the first place.

Additionally, this would not result in the finding of any "new" automorphisms anyhow. Consider once again the isomorphism-invariant shape of our search-tree. Let us say, for example, that the currently examined leaf z gives us a permutation whose resulting labeled graph is identical to the one we found from a sibling leaf t_1 of t_0. It then follows that the partition belonging to the parent p_t of t_1 must be isomorphic to that of the parent p_z of z, and, transitively, z must have some sibling y whose partition is isomorphic to t_1's sibling t_0. Furthermore, since the isomorphism μ for which $\mu(\pi_{p_t}) = \pi_{p_z}$ is also an isomorphism between t_1 and z, as well as t_0 and y, it is not difficult to see that the automorphisms provided by the two pairs are indeed the same exact automorphism.

This naturally works for any two leaves of the search-tree, as for any leaves l_1, l_2 with identical resulting labeled graphs, t_0 will have a corresponding vertex y in a subtree given by the relation of l_1 and l_2 to t_0 for which the partition of y gives a graph form identical to $\mu_0(G)$. Therefore, comparisons with μ_0 alone should be sufficient.

Using automorphisms We have mentioned that with equitability refinement, any automorphism present in G can in fact be obtained from the search-tree through the above method. However, we do not simply want to collect automorphisms, we want to prune our search-tree. Here is the basic method in which McKay uses automorphisms to decrease necessary computations: whenever an individualization would occur on a partition π, we can reduce the number children generated based on the current partition and the automorphisms found so far. Consider that we have stored an automorphism μ for which all cells of π are fixed "points" of μ. That is, for any cell $X \in \pi$ and $x \in X$, we have $\mu(x) \in X$.

Now let us say that $X \in \pi$ is the cell chosen for individualization in this step of the construction of our search-tree. Consider two distinct vertices $u, v \in X$ for which $\mu(u) = v$. In accordance to the reasoning earlier, it is plain to see that breaking out u or v from the graph results in isomorphic partitions, with μ being the clear isomorphism, which in turn results in isomorphic subtrees, giving us isomorphic trivial partitions in the leaves, which give identical labeled forms. Therefore, it is

enough to generate a child partition for only one of u or v. So when individualizing X, we check for the following for each stored automorphism μ:

- Are the orbits of μ contained within cells of π?
- If so, choose an arbitrary element from each orbit within X to keep, the rest are marked as duplicates.
- When generating the children partitions of π, only individualize vertices that were not marked as duplicates.

All of this can be efficiently decided and computed with appropriate data structures. It should be noted that pruning the search-tree in this way will in turn remove permutations that lead to graph forms identical to $\mu_0(G)$. However, it can be shown that enough automorphisms are always collected that are sufficient to form a *generator* of the automorphism group of G. Any automorphism of G can be constructed by successively applying automorphisms stored throughout the examination of the search-tree. In particular, we can calculate the orbits of $V(G)$ by the time the algorithm terminates, and we can make notes of certain vertices being in the same orbit throughout the run of the process. This is especially good for pruning the root of the tree. Since the first condition is trivially satisfied for pruning, we only need to check that the individualized vertices are not within one orbit, which can be easily ensured in accordance to the automorphisms found so far.

Furthermore, having access to a generator of $\mathrm{Aut}(G)$ gives us a relatively efficient way of checking whether two *subsets* of $V(G)$ are isomorphic with one another or not, which will be of great importance to us in Section 3.

While the above conditions can prune branches that are isomorphic based on automorphisms already found, it does not take into account the actual group of automorphism that they generate. More recent versions of `nauty` can use the randomized Schreier method [27], a probabilistic permutation algorithm to test partitions for further automorphisms that can be generated from the ones already found. While not guaranteed, it can often prune branches of the search-tree that could otherwise not be recognized as isomorphic to previous ones by the above method method alone.

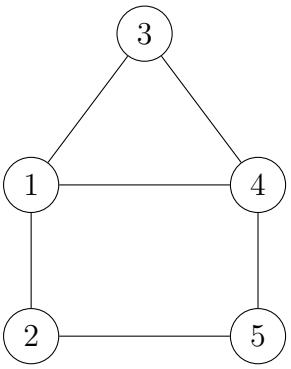
Choosing the individualization cell While the efficiency and correctness of the refinement procedure plays a large role in the eventual breadth of the search-tree, the importance of choosing the individualized cell should not be understated. While choosing the first non-trivial cell is a simple way to ensure the isomorphism-invariance of our choice, consider how the actual cell affects what our search-tree will look like. A new branch will be started and examined for each element of the target cell, meaning that we might end up with more leaves than if we were to choose a smaller target cell. In addition, as pointed out by McKay in [19], individualizing elements in a smaller cell should intuitively lead to permutations that have a higher chance of providing automorphisms that leave the correct vertices in a fixed position, and could be used to further prune the search-tree. At first, this may seem like a clear indicator that the target cell of choice should be the first non-trivial cell of minimal

size as a means of minimizing the number of branches each chance we get, however, in practice, this turned out not to be entirely the case. Kocay [15], demonstrated that simply choosing the first non-trivial cell leads to better overall performance than always resorting to the smallest one. It seems as though "random" cell choice allows for refinement to tell apart more kinds of neighboring-structures more often, resulting in more efficient partition refinements than in the naively controlled case.

Another method of trying to steer target cell choice which is still employed by `nauty` is to specifically try and choose a cell which has many kinds of neighbors in other cells, as individualizing vertices in such a cell would lead to refinement being able to split more cells right away, reducing the depth of the resulting branches. In practice, this is achieved by choosing the earliest cell which has largest number of non-maximal neighboring cells, that is, cells for which there is at least one edge between them and the target cell, but not all possible edges between the two are present. This method of choosing the target cell clearly comes with more required computation, but still performs competitively in practice.

3.4.3 An example run

Here, we show a simple example of how McKay's search-tree canonization algorithm computes a canonical labeling.



$\pi = \{\{1, 2, 3, 4, 5\}\}$

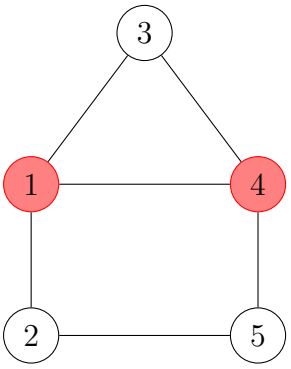
v	(1,2,3,4,5)
1	3
2	2
3	2
4	3
5	2

Figure 7: Initial partition.

The initial graph G is depicted on the left of Figure 7, and on the right are the degrees each vertex has into the so far only cell. The starting cell is always marked as active for refinement, as equitability is usually not given. (We may remark that a unit partition on a graph is equitable exactly when the graph is regular.) Clearly, the partition is not equitable, as some elements of our only cell have 2 neighbors in it, while others have 3.

We split the cell by the number of neighbors, and both resulting cells are marked as active. Continuing with refinement, we now take the first active cell $X_1 = \{2, 3, 5\}$, and check whether the number of neighbors in it is homogenous within each cell. The first cell checked, X_1 immediately fails the test, as the vertex 3 differs form vertices $2, 5$. X_1 is split based on the number of neighbors within X_1, the resulting cells are marked as active.

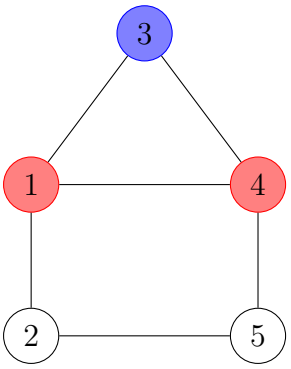
The resulting partition in Figure 9 is eventually deemed equitable once all active cells are cleared, as within each cell of the partition, all vertices have a consistent



$\pi = (\{2,3,5\},\{1,4\})$

v	{2,3,5}	{1,4}
2	1	1
3	0	2
5	1	1
1	2	1
4	2	1

Figure 8: First split in first refinement.

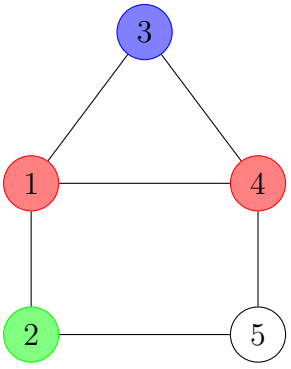


$\pi = (\{3\},\{2,5\},\{1,4\})$

v	{3}	{2,5}	{1,4}
3	0	0	2
2	0	1	1
5	0	1	1
1	1	1	1
4	1	1	1

Figure 9: Second split in first refinement.

number of neighbors withing any given cell. This refinement is therefore finished, and individualization needs to take place to continue. The first non-trivial cell, $\{2,5\}$ is chosen for individualization. Since no automorphisms have been found so far, we do not check for any pruning. First, we examine the branch produced by individualizing 2.



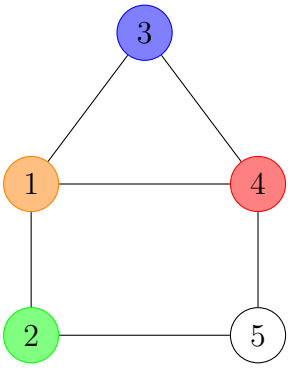
$\pi = (\{3\},\{2\},\{5\},\{1,4\})$

v	{3}	{2}	{5}	{1,4}
3	0	0	0	2
2	0	0	1	1
5	0	1	0	1
1	1	1	0	1
4	1	0	1	1

Figure 10: Individualization of 2.

This partition is not equitable anymore, the first violating cell-pair found is $\{1,4\}$ and 2. The former is split appropriately.

Our ordered partition has become trivial, which means refinement is finished (as the partition is trivially equitable), and the first leaf of the search-tree has been found. We store the graph resulting from the application of $\mu_0 = (3,2,5,4,1)^{-1} =$

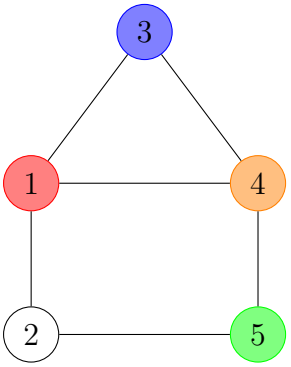


$\pi = (\{3\}, \{2\}, \{5\}, \{4\}, \{1\})$

v	{3}	{2}	{5}	{4}	{1}
3	0	0	0	1	1
2	0	0	1	0	1
5	0	1	0	1	0
4	1	0	1	0	1
1	1	1	0	1	0

Figure 11: Second refinement.

$(5, 2, 1, 4, 3)$ to G. Note how the neighbor-counting table corresponding to this partition in figure 11 contains the adjacency matrix of this graph. With this, the branch resulting from the individualization of 2 is finished. We now return to the step of individualization in $(\{3\}, \{2, 5\}, \{1, 4\})$, this time going with 5.



$\pi = (\{3\}, \{2\}, \{5\}, \{4\}, \{1\})$

v	{3}	{5}	{2}	{1}	{4}
3	0	0	0	1	1
5	0	0	1	0	1
2	0	1	0	1	0
1	1	0	1	0	1
4	1	1	0	1	0

Figure 12: Leaf resulting from the individualization of 5.

The partition is once again no longer equitable, so we split the first found violating cell during refinement, analogously to the previous case. The resulting trivial ordered partition is depicted in Figure 12. Another leaf has been found, and the graph resulting from applying $\mu_1 = (3, 5, 2, 1, 4)^{-1}$ to G is identical to $\mu_0(G)$, meaning that this form remains lexicographically minimal, and an automorphism has been found in the form of $\mu_0 \circ \mu_1^{-1} = (5, 2, 1, 4, 3) \circ (3, 5, 2, 1, 4) = (4, 5, 3, 1, 2)$.

And thus, the individualization of $(\{3\}, \{2, 5\}, \{1, 4\})$ and the recursion ends, our search is complete. The canonically labeled graph is $\mu_0(G)$, and $(4, 5, 3, 1, 2)$ is found to be the only non-trivial automorphism of G. The canonical form of G and the search-tree are depicted in Figure 13.

3.4.4 Lexicographic search-tree

Another method of pruning the search-tree is to require lexicographic minimality not just from the graph form of the individual leaves, but from the ordered partitions that are found in the path leading down to them. More specifically, let us say that we have some $\leq_\Pi$ ordering on the ordered partitions of $V(G)$, which can be computed

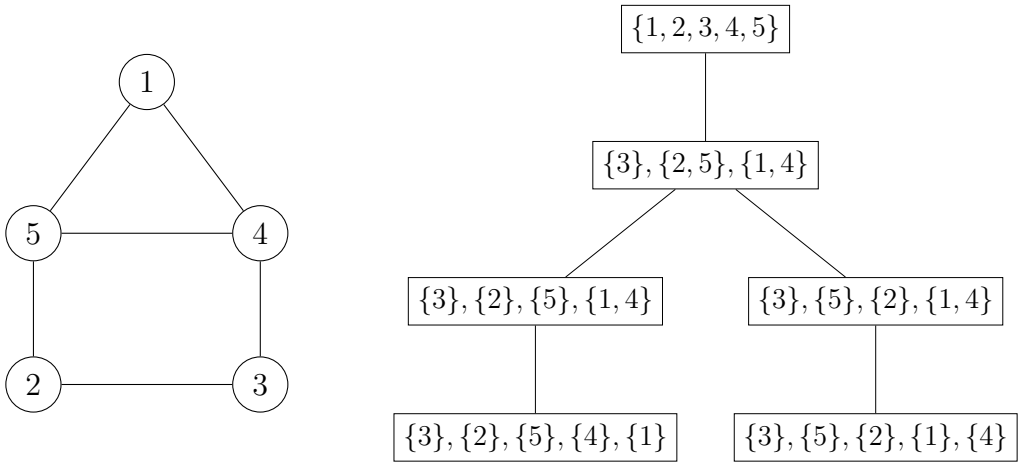


Figure 13: Canonically labeled graph and the corresponding search-tree.

in an isomorphism-invariant manner. Something that can be calculated relatively efficiently, as we would want to do this within each node of the search tree, such as taking the sizes of cells of each partition and comparing their square sum, or their geometric center, or any other isomorphism-invariant value that can be calculated from the structure of π. Information from the graph itself and the order of cells can also be used to further help identifying different partitions more easily. If such a comparison is feasibly achieved, then we can make our search shorter by requiring that whenever individualization occurs, only the lexicographically smallest ordered partitions are kept and examined further for subtrees. Since equitability refinement gives us so much specification as to how the partitions resulting from individualization "should look", this comparison is usually applied only after refinement has taken place and equitability is established.

3.5 Equitability and the hashing algorithm

Let us think back on the algorithm presented in Section 3.2. In it, we continuously updated the "structural status" of vertices in the form of modifying their representative strings based on what string their neighbors have, and eventually categorized them based on what conclusions their strings provided as to how the neighboring of each vertex distinguished it from the others. One might notice that this general paradigm is very similar to the one performed by equitability refinement, in which we continuously examine the neighborhoods of vertices, and then separate previously "similar" vertices that were in one cell when their neighborhoods are determined to be different in a specific way.

This might lead one to examine more closely the connection between the vertex-hashing canonization algorithm and the equitability condition. To better understand how these two are related, let us take a look at a generalized version of Algorithm 2 which can also utilize a given ordered partition π to further help identify vertices that are "different" in regards to the actual partition itself. The only difference required to achieve this is the nature of the initial strings given to vertices. When computing HashLabel in Algorithm 2, initial labels were only determined by whether a vertex

was in the differentiating set or not, however, the ordered partition π allows us to set these more appropriately: to compute the hash-refined partition HashPartition-Label(G, π, k) (hereby referred to as HPL(G, π, k)), we calculate a specialized hash label for each vertex, where the initial label of each $u \in V(G)$ to the position of its cell within π, except for v, which receives a unique individualizing label "c^*", where c is the position of the cell of v within π. These "c^*" characters are considered lexically minimal during concatenation ordering. From there, the resulting strings are computed through the same concatenation-cycles as in the case of the unit partition. At the end of k cycles, we split the current cells based on lexicographic ordering by the label each vertex has at the end of the last cycle. Important to note is that here, k refers to the number of concatenation cycles, not the number of individualized vertices, which in this case is always 1.

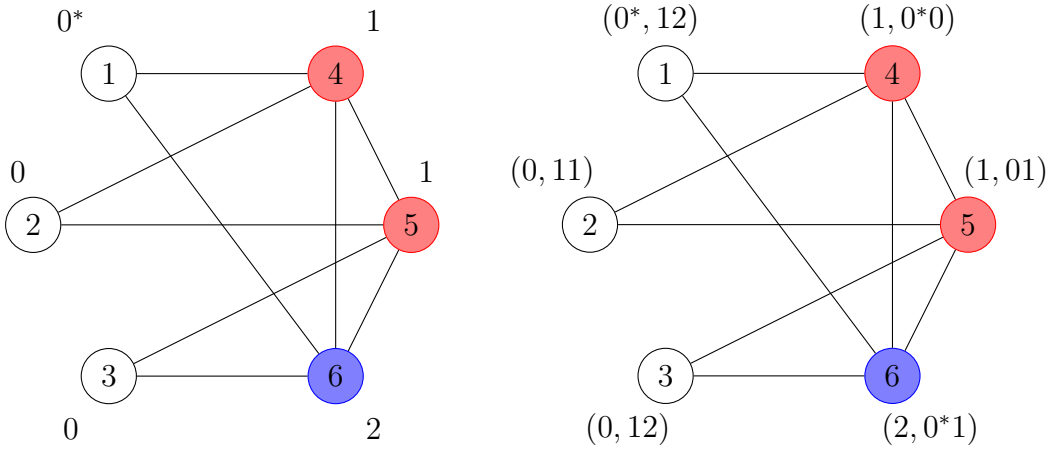


Figure 14: Strings at the end of the first concatenation cycle for $v = 1$.

See Figure 14 for an example of how this works. The graph pictured has the ordered partition $\pi = (\{1, 2, 3\}, \{4, 5\}, \{6\})$, and the strings calculated during the computation of $v_k^*(G, \pi, 1)$ are shown. Note how here, all vertices already have different labels. Aggregating these strings gives us:

$$s^*(1) = \text{“}((0^*, 12), (0, 11)(0, 12)(1, 0^*0)(1, 01)(2, 0^*1))\text{”}$$

, which is the final label of 1 with $k = 1$. In HPL$(G, (\{1, 2, 3\}, \{4, 5\}, \{6\}), 1)$, it will be in one cell with the vertices that receive this exact string after their own initialization.

Now let us consider how the equitability condition affects the way HPL affects the partition. First let us ensure that HPL never coarsens the partition, or intersect any cells by chance, that is, it can be used as a way to refine partitions.

Theorem 3.7. For any graph G and partition π of $V(G)$ and $k \in \mathbb{N}$, we have HPL$(G, \pi, k) \leq \pi$, that is, every cell of the former is a subset of a cell within π.

Proof. Consider some $x \in X_i$ atom from the i-th cell of π. Let us examine the computation of $s_1^*(x)$. Initially its label is "i^*", and, because the previous label is always kept one the left, we can show that for any $v \in V$, after k concatenation

cycles, the first k characters are parenthesis openings for each vertex, and the k-th, first non-parenthesis character within $s_k(v)$ is its starting label character, in the case of x, "i^*" specifically in the case of x.

We can use induction by the number of concatenation cycles. For $k = 1$, the claim is true by the nature of the algorithm. Assuming this hold for k, then each string in the next concatenation cycle will begin with a parenthesis opening, followed by its previous label, thus showing that the claim holds for $k + 1$.

Because this character is lexicographically minimal in all aggregation orderings, including the computation of $s_k^*(x)$ itself, it is plain that the first non-parenthesis character of this final label must be i^* as well. This holds true for all vertices, and thus any vertices with the same final label need to come from the same starting cell. Since the new partition is formed by grouping vertices with the same final string together, this proves the theorem. □

Now let us show that $\mathrm{HPL}(.,.,1)$ is a good "detector" of equitability, that is, if a partition is not equitable, it will show up as an irregularity after just one concatenation cycle.

Theorem 3.8. For any graph G and partition π of $V(G)$, if π is not equitable, then $\mathrm{HPL}(G, \pi, 1) < \pi$.

Proof. Because of the previous theorem, it suffices to show that there are two vertices which start in the same cell but end up with differing final strings.

Let us take two cells $X_i, Y \in \mathrm{HPL}(G, \pi, 1)$ which violate the equitability condition, and take $x_1, x_2 \in X_i$ for which $d(x_1, Y) \neq d(x_2, Y)$. We want to demonstrate that $s_1^*(x_1) \neq s_1^*(x_1)$. Because they are in one cell, consider what $s_1(x_1)$ and $s_1(x_2)$ look like in their respective label calculations. Because they violate the equitability condition, they have a different number of instances of the initial string belonging to the elements of Y, meaning that these strings are different.

Furthermore, because the "individualized" vertex is always the only one with "i^*" as its first non-parenthesis character, a property that all vertices keep with their initial character as we have seen in the proof of Theorem 3.7, the strings of these vertices will be first in the lexicographic order when aggregating all strings during the computation of $s_1^*(x_1)$ and $s_1^*(x_2)$, respectively. Therefore, this starting segment differs between the two vertices, thus X must split into at least two parts. □

A trivial consequence of this theorem is that the repeated application of the above partition refinement function is an isomorphism-invariant method of reaching a finer ordered equitable partition, as the partition eventually either naturally becomes equitable or ends up as trivial, which in turn would trivially make it equitable. This is essentially a more general way of describing a refinement procedure that ensures equitability in the partition produced. An examination of what the equitability-wise implications may be when k is higher than 1 are elaborated in Section 5.3.

3.6 Algorithms for larger graphs

While our research is concerned mainly with the canonization of smaller graphs, it should be mentioned that there are other heuristics and approaches which can be

implemented to more efficiently arrive at a canonical representation of a graph, but which only provide a considerable improvement when the graphs are large enough to warrant extensive computations to begin with, and are simply not preferable to other methods when the input graphs are too small. In this section, we briefly discuss some of these approaches to gain further insight into graph canonization tactics.

3.6.1 The Traces search-tree

`Traces` [19] is a leading canonization tool developed alongside `nauty` which attempts to take even further advantage of automorphism-pruning by finding automorphisms "before" actual canonization begins, therefore it is highly efficient on graphs with particularly large automorphism groups, where any two particular leaves are more likely to contain permutations that can give us automorphisms.

While most search-tree canonization algorithms, such as the one described in the previous section traverse the search-tree in a recursive, depth-first manner, `Traces` a breadth-first approach to it. Once refinement has taken place, Instead of examining each branch resulting from individualization of a cell one after another, all potential children partitions are generated right away. Now for each generated node t, a "quick-run" of the search algorithm is performed, as in the computation of a single test-leaf that could be generated on the branch of t during the course of a search-tree algorithm. The permutations yielded from these leaves are now examined for automorphisms alone, and any that are found are stored straight away.

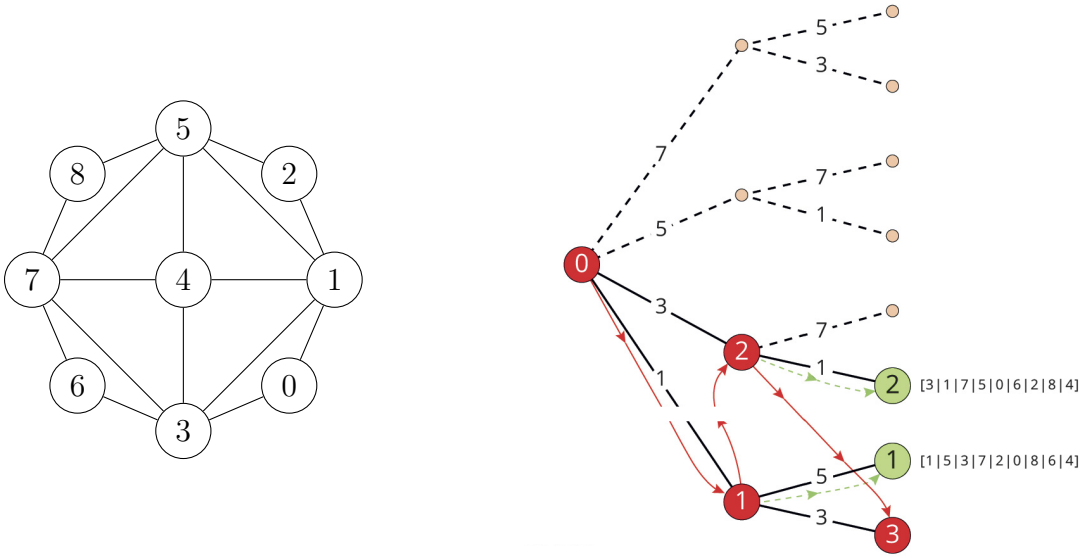


Figure 15: `Traces` quick-runs detect early automorphism.

This is demonstrated on the example shown in Figure 15, adopted from [19]. The graph on the left is examined as normal, after refinement has taken place, the cell $\{1, 3, 5, 7\}$ is chosen for individualization. In a breadth-first manner, all children partitions generated are processed one by one, and for each one we randomly compute a test-leaf from its branch. After the individualization of 1, the leaf with

$(\{1\},\{5\},\{3\},\{7\},\{2\},\{0\},\{8\},\{6\},\{4\})$ is reached, and after the individualization of 3, the one with $(\{3\},\{1\},\{7\},\{5\},\{0\},\{6\},\{2\},\{8\},\{4\})$ is. Comparing these forms gives us na automorphism right away, which in fact lets us prune the other two initial individualization branches. Therefore, only the first two branches are actually fully processed.

Another quirk of this algorithm is the preferred choice of target cell when it comes to individualization. Because of the previously mentioned quick-run tactic, the more chances of finding automorphisms in such a way, the more pruning we can perform on the tree and therefore target cells of higher size are actually preferred.

3.6.2 Graphs with bounded tree-width

Similarly to how various problems are often easier to solve on trees, another classic approach to decreasing the complexity of certain problems is through assumption of limited tree-width. While often not very practical due to extremely high constant multipliers, these can be used to show that it is theoretically possible to compute various difficult decision and optimization problems on a graph, given that a tree-decomposition of the graph can be given.

Definition. A *tree-decomposition* of a graph G is a tree T and a collection of vertex-sets $v : V(T) \mapsto 2^V(G)$ such that all vertices of $V(G)$ are present in at least one set, all edges have at least one set with both end-points in it, and $\{t \in T : u \in v(t)\}$ is a spanning tree of T for any $u \in U$.

The *tree-width* of a graph G is $\min\{\max|v(t)| \quad : \quad t \in T,\ (T,v)$ is a tree-decomposition of $G\} - 1$.

Some classic results regarding tree-decompositions include Bodlaender's and Kloks's [8] algorithm which, for a given a constant k, if a graph has tree-width at most k, constructs an appropriate tree-decomposition in linear time. A relevant result of Bodlaender is one from 1990 [7], in which he gives an algorithm which operates on partial k-trees, a constructively defined family of graphs with known tree-width, as a tree-decomposition of width at most k is known for them by definition. This algorithm shows that the isomorphism problem can in fact be solved for partial k-trees in polynomial time when k is a constant. Which in turn implies that the approximation holds true for all graphs of tree-width at most k.

The essence of the algorithm lies in the exhaustive examination of all k-tuples in the graph, and using dynamic programming to check whether it is possible to build up a proper tree-decomposition from them. It was recently discovered by Lokshtanov [16] that Bodlaender's algorithm can in fact be modified to not just decide the isomorphism of these graphs, but to construct a canonical labeling of G in polynomial time, assuming that it is of constant tree-width, as well as a canonical representation of constant size, giving the first documented canonization algorithm that runs in FPT time.

Other results regarding canonization of bounded tree-width graphs include Elberfeld's [12] proof that the computation of a canonical form for graphs of bounded tree-width is in fact in LOGSPACE, that is a canonical form for such a graph can be computed using $\mathcal{O}(\log_n^c)$ memory for some $c > 0$, and Arvind's [3] recent proof that

k-tree canonization is in $\mathrm{FN}^{\mathrm{NL}}$, that is, it can be solved in logaritmic space with the help of an unambiguous LOGSPACE oracle, which they then used to prove that the isomorphism problem is in fact in the strongly unambiguous LOGSPACE problem class.

4 Non-ismorphic graph generation

Graph canonization has applications in data mining, chemical analysis and various theoretical problems concerning graph symmetry and isomorphism. One such problem is the ability to generate graphs only up to isomorphism. Often we may want to be able to run or test a certain function or method on many graphs, but in many cases, the results are not meaningfully different amongst graphs that are isomorphic, because the properties that are examined are similar in each one. In this case, it would be sufficient to only examine one representative of each isomorphism class, which would greatly decrease the amount of computation required, while still getting all the desired information.

Thus the problem: is there an efficient way of generating all graphs of size n up to isomorphism, that is, only one representative of each isomorphism class? Canonical labeling may come to mind straight away, but their application is not as straightforward as it may seem at first. Simply generating every labeled graph of size n and only outputting each graph in case it is the canonical form defeats the entire purpose of non-isomorphic generation, as getting around the sheer amount of different labeled graphs is in itself what we are trying to achieve. However, canonical labelings can indeed be used to efficiently provide such a collection of graphs. McKay proposes a method [18] of generating graphs of a given size in such a manner that relies only on being able to calculate a canonical labeling for a graph, and having access to a generator of its automorphism group. The algorithm outlined in Section 3.4 provides both of these.

Generating graphs Generating graphs of size n without regard for isomorphisms is simple enough, we simply have to go through all $2^{\binom{n}{2}}$ possible edge-sets in some manner. This clearly takes an exponential amount of time no matter how we approach the problem due to the sheer number of graphs that need to be outputted. While time-consuming, the actual realization of iterating through $\{0,1\}^{\binom{n}{2}}$ is not particularly difficult from a technical standpoint. We will examine a method of doing so which, while not being exactly straightforward, will give us a practical way of utilizing graph canonization to remove any graphs that are not important to us in the isomorphism-free case. Imagine that we already have access to a list of all graphs withing $\mathcal{G}_{n-1}$. We could then compute all elements of $\mathcal{G}_n$ in the following way: for every $G \in \mathcal{G}_{n-1}$, we add a new vertex with the label n to G and add a new graph to our collection for every possible neighboring of the new node.

Claim 4.1. $\mathcal{G}_n = \{(V(G)+\{n\}, E(G)+F) \,:\, G \in \mathcal{G}_{n-1}, F = \{nu : u \in U \subseteq V(G)\}\}$.

To verify this, simply consider that for any $H \in \mathcal{G}_n$, we can delete the vertex with the label n, the resulting $G = H - \{n\}$ graph is a good "parent" graph for

H with the neighborhood of the deleted vertex in the above formula. This gives us a simple way of generating all elements of $\mathcal{G}_n$ from nothing: We start off with $\mathcal{C}_1 := \{K_1\}$ and then, for every $i = 2, 3, \ldots, n$, we go through each $G \in \mathcal{C}_{i-1}$, and add $G + \{i\} + F$ to $\mathcal{C}_i$ for every $F \in 2^{[i-1]}$. We can confirm, through the same line of logic as before, that this way, $\mathcal{C}_k = \mathcal{G}_k$ for any $k \in [n]$ at the end of generation. This method of graph generation is beneficial specifically because it gives an efficient way of filtering out specific graph properties.

Definition. We say that a graph property $P : \mathcal{G}_* \mapsto \{0, 1\}$ is *vertex-hereditary* if for any $H \in \mathcal{G}_*$ such that $P(H) = 1$ holds, $P(G) = 1$ also holds for any induced subgraph $G \subseteq H$.

Some examples of vertex-hereditary properties are:

- triangle-freeness,
- planarity,
- upper limit on vertex degrees,
- upper limit on number of edges,
- bipartiteness,
- "logical or" of other vertex-hereditary properties,
- "logical and" of other vertex-hereditary properties.

Let us assume that someone only wants to generate graphs within $\mathcal{G}_n$ that satisfy some vertex-hereditary property P. A straightforward approach would be to simply generate all possible candidate graphs, then check for each one whether P holds for them. However, the above method of generating graphs gives us a much more efficient method. When adding some new graph G to $\mathcal{C}_i$, we check whether $P(G) = 1$ holds, and we only keep the graph when it does. It is clear that doing this will only get rid of violating graphs down the line, as any graph generated from G will contain G as a spanning subgraph, thus if $P(G) = 0$, we will also have $P(H) = 0$ for any supergraph $H \supseteq G$. Therefore, $\mathcal{C}_k$ will contain exactly the elements of $\mathcal{G}_k$ which satisfy P. This way, we can save time on the examination of numerous graphs of higher size, as they do not even need to be considered for generation as they were already disqualified when the appropriate subgraph was first examined.

4.1 Using the canonical labeling

In the same way that vertex-hereditary properties can be removed level-by-level to save lots of computation time in the long run, we can also use graph canonization to ensure that in each $\mathcal{C}_i$, only one representative from each isomorphism class is present. Let us for a moment assume that we have managed to construct the set $\mathcal{C}^*_{n-1}$ in which there is exactly one element from each isomorphism class of $\mathcal{G}_{n-1}$. To successfully construct the appropriate $\mathcal{C}^*_n$ with the above method, we need to assure that no two graphs kept are isomorphic. This could happen by one $G \in \mathcal{C}^*_{n-1}$

having two different edge-sets F_1, F_2 for which the resulting graphs are kept, or by two different $G_1, G_2 \in \mathcal{C}^*_{n-1}$ having such F_1, F_2. While posing similar problems, the two cases are tackled differently by canonization. One can define the following conditions that will help ensure these violations do not occur:

Definition. Given a graph G, we say that a vertex $v \in V(G)$ is *canonically maximal* for some canonical labeling f if v is in one orbit with some $z \in V(G)$ for which there exists a permutation $\mu \in S_n$ such that $\mu(z) = n$ and $\mu(G) = f(G)$. The indicator of this property will be denoted as $\textsc{CanMax}_f(G, v)$.

Basically, the vertex v needs to be isomorphic with a vertex that would receive the highest index in a canonical labeling. The fact of it being the highest index in particular is not a necessary trait, it is simply a way of pinning down an orbit within $f(G)$ that can easily be identified. However, if certain properties are known about the actual canonical form given by f, such as in the case of McKay's canonical labeling, the choice of orbit can still become useful throughout graph generation. While trivially true, it should also be noted that there is always at least one canonically maximal vertex in a graph for any canonical labeling.

Definition. Given a graph G, we say that a set of vertices $X \subseteq V(G)$ is a *orbitally minimal* in G if for any automorphism $\mu \in \mathrm{Aut}(G)$, we have $X \leq \mu(X)$ for some ordering on $2^{V(G)}$. The indicator of this property will be denoted as $\textsc{OrbMin}(G, X)$

Once again, the actual ordering or set of vertices is arbitrary, we simply need a consistent way of deciding whether the subset X is a specific one withing a certain "subset-orbit" of G. One such straightforward ordering on $2^{V(G)}$ would be the lexicographic ordering of the indicator vectors, where the indicator vector $v_X \in \{0,1\}^n$ is defined as $v_X^{(i)} = \begin{cases} 1 \text{ if } i \in X, \\ 0 \text{ otherwise.} \end{cases}$

The same way that every graph contains canonically maximal vertices, the following claim regarding the density of orbitally minimal sets trivially follows from the definition, but should still be acknowledged as it will be useful to us.

Claim 4.1. Given a graph G and a set of vertices $X \subseteq V(G)$, there is a unique orbitally minimal $Y \subseteq V(G)$ such that $Y = \mu(X)$ for some $\mu \in \mathrm{Aut}(G)$.

Now we can examine how these properties can help us generate non-isomorphic graphs. Given some canonical labeling f, and a way to order sets of vertices, we have the following algorithm:

Algorithm 4 Non-isomorphic graph generation

```
procedure GEN(n)
    C*_1 ← {K_1}
    for all i = 2, ..., n do
        C*_i ← ∅
        for all (V, E) ∈ C_{i−1} do
            V' ← V ∪ {i}
            for all X ∈ 2^V do
                E' ← E ∪ {vi : v ∈ X}
                G' ← (V', E')
                if CANMAX(G', i) = 1 and ORBMIN(G, X) = 1 then
                    C*_i ← C*_i ∪ G'
                end if
            end for
        end for
    end for
end procedure
```

Here, we do the same thing as we did when generating labeled graphs, except that a new graph is only kept when the new vertex is canonically maximal, and its neighborhood is orbitally minimal in the "parent" graph. We can show that these changes alone are enough to ensure that $\mathcal{C}_k^*$ contains exactly one representative from each isomorphism class of $\mathcal{G}_k$ for each $k \in [n]$ at the end of the algorithm. First, we show that at least one representative is kept from each isomorphism class.

Claim 4.2. For any $G \in \mathcal{G}_n$, there is a $C \in \mathcal{C}_n^*$ such that $G \sim C$.

Proof. We will use proof by induction. For $n = 1$, the claim is trivially true. Let us assume that for any $i \in [k]$, the claim holds true. Now let us take some $G \in \mathcal{G}_{k+1}$, and within it, a vertex z that is canonically maximal for f, and $N(z)$ its neighborhood in G. Let G' be $G - z$. By the induction hypothesis, there is a $C' \in \mathcal{C}_k^*$ such that $G' \sim C'$. Let $X \subseteq V(G')$ be the vertices corresponding to $N(z)$ with regards to an isomorphism between G' and C', and let $Y \subseteq V(G')$ be the lexically minimal vertex-subset for which $Y = \mu(X)$ for some $\mu \in Aut(C')$.

Now consider $C = (V(C') \cup \{\mathrm{n}\}, E(C') \cup \{vn : v \in Y\})$. We can see that $G \sim C$, as the permutation moving G' to C', followed by the automorphism moving X into Y, all appended by moving z to n and keeping it there will get us from G' to C'. Furthermore, because n is the image of z in an isomorphism, it is not difficult to see that n must also be canonically maximal within C for f, as $f(G) = f(C)$ and the orbit of the maximally labeled vertex is preserved through isomorphism. Since $N(n)$ was specifically chosen to be orbitally minimal, that condition is satisfied as well. Based on this, we can conclude that $C \in \mathcal{C}_k^*$ and $C \sim G$, which proves the claim. □

So we are definitely not getting rid of too many graphs by these two conditions alone. However, this does not necessarily mean that there are not still isomorphic graphs within each set. It can thankfully be shown, however, that there are none.

Claim 4.3. For any two different $C_1, C_2 \in \mathcal{C}_n^*$, we have $C_1 \nsim C_2$.

Proof. We will use proof by induction. For $n = 1$, the claim is trivially true. Let us assume that for any $i \in [k]$, the claim holds true. Now let us take two distinct graphs $C_1, C_2 \in \mathcal{C}_{k+1}^*$. Let us indirectly assume that $C_1 \sim C_2$. Let us take the "parent" graphs $G_1 = C_1 - n$ and $G_2 = C_2 - n$. Clearly, $G_1, G_2 \in \mathcal{C}_k^*$, as these are the graph from which we generate and place C_1 and C_2 into $\mathcal{C}_{k+1}^*$.

Because n is a canonically maximal vertex within both graphs, there must be an isomorphism between C_1 and C_2 for which n is a fixed point. Clearly, applying the same permutation to the rest of the graph after n is removed is still an isomorphism between the subgraphs. Thus, we have $G_1 \sim G_2$ and from the induction hypothesis it follows that $G_1 = G_2$. But in this case, in order for C_1 and C_2 to be different yet isomorphic, the neighborhood of their new vertices must be isomorphic within G_1. But then they cannot both be orbitally minimal, as they could be moved into one another with an isomorphism, so we would not have kept both C_1 and C_2, which is a contradiction, so our assumption, $C_1 \sim C_2$ cannot be true.

□

As a direct consequence of Claims 4.2 and 4.3, we get the following statement, which proves that the above approach to non-isomorphic generation is indeed correct.

Theorem 4.1. For any $G \in \mathcal{G}_n$, there is exactly one $C \in \mathcal{C}_n^*$ such that $C \sim G$.

4.2 Canonization-specific improvements

To decide whether we add a graph to $\mathcal{C}_k^*$, the full canonization of the graph is usually warranted, both for deciding whether the new vertex is canonically maximal, and to collect enough automorphisms to deicide the orbital minimality of its neighborhood. However, if the nature of the labeling provided by f is understood, we can further decrease the amount of necessary computation.

Note for instance what a canonically maximal vertex needs to look like when the canonical labeling is the one described in Section 3.4. Here, the first refinement step always sorts the vertices of the graph by degree. This means that if the vertex n does not have maximal degree, it will immediately be placed into a non-final cell of the partition, therefore it can never end up as the element of the final cell of a leaf partition, which in turn means that no canonical form gives n the highest index. This means that a new graph can only ever be kept if the degree of the node vertex is at least has maximal degree, which, means that when examining $2^{V(G)}$ for the potential neighboring of n, it suffices to only examine subsets of size at least $\max\{d(v) \ : \ v \in V(G)\}$.

Even if n ends up being a vertex of maximal degree in the new graph G', it is still possible that it falls out of the final cell within the very first refinement step. When this is observed, we do not need to calculate the canonical labeling any further, as n will definitely not be canonically maximal.

5 Implementation, improvement attempts

In this section, we will go over the specific results of our implementations, as well as our experimentation regarding approaches to improving existing canonization algorithms with new heuristics for the examination of vertex-partitions.

5.1 Search-tree implementation

Part of this project was to create a more modern, customizable implementation of McKay's search-tree algorithm outlined in Section 3.4, which is easier to pick up and to make modifications within. The `nauty` [17] project, while operational and efficient, is not very welcoming for users planning to test out their own modifications. The code of `nauty` is written in `C`, a procedural language that provides little flexibility in terms of how users can adjust the algorithm for whatever heuristic search-tree approach they might want to try out in their program, unless they are willing to look through huge chunks of code whenever they want to adjust a certain aspect of the process. Similar projects have been taken on in recent years, such as one by Andersen [1], who made a tool that allows users to run the search-tree algorithm with their select choice of heuristics and examination processes that were available at the time, and let them compare and cross-reference the various methods that can be obtained by mixing and matching them.

Our implementation is written in `C++`, an object-oriented language that allows more flexibility as to how certain batches of data can interact with one another, and gives us more leeway in terms of how new functions and methods can be added and utilized by the algorithm without needing to modify a ton of code.

For the purposes of general performance testing on our implementation, the algorithm was ran on large collections of graphs, as this sort of generator is usually used repeatedly on many graphs, for the purpose of non-isomorphic generation, for example. The total time spent on canonization of graphs was then compared with the amount of time taken by `nauty`, one of the best-performing currently available canonization tools of our time, to perform the same task. While the exact efficacy of this tool was not reached, our higher-level implementation still canonizes small graphs in the same magnitude of time as `nauty`, canonizing massive collections of graphs without issue, and successfully computing generators for the automorphism group of each graph examined. See Figure 16 for a comparison between the runtimes of `nauty` and our own canonization tool.

5.2 Non-isomorphic generator implementation

As part of a collaboration with Lóránt Matúz, a working higher-level implementation of the non-isomorphic graph generating algorithm was also created relying on our canonization implementation. With Matúz, we have created a tool that successfully uses our canonization tool to filter out isomorphic graphs, and which can also be given customizable property-checking filters, which can be used to efficiently generate graphs which possess various vertex-hereditary properties. The graph-collections used in 5.1 to test the runtime of our implementation were all in fact generated using this tool.

Set	Size	`Can` [ms]	`nauty` [ms]
$\mathcal{C}_7^*$	1044	3.7	2.6
$\mathcal{C}_8^*$	12346	38.9	20.3
$\mathcal{G}_6$	32768	70.5	34.6
$\mathcal{C}_9^*$	274668	780.4	290.5
$\mathcal{G}_7$	2097152	4231	1811
$\mathcal{C}_{10}^*$	12005168	30104	11806

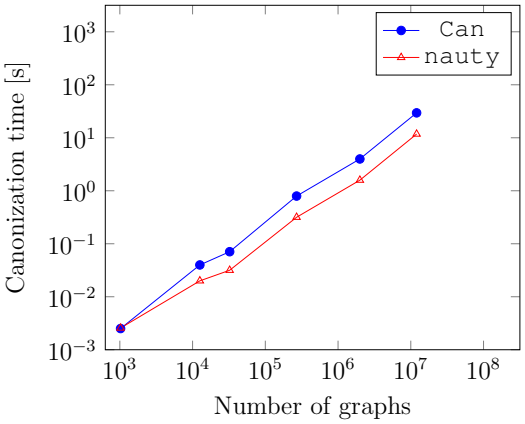


Figure 16: Runtimes for canonization by `nauty`, and by us (`Can`).

One major difference in the canonization used in generation is that orbital minimality amongst subsets of $V(G)$ needs to be computed, which is achieved by maintaining a list of currently known orbitally minimal subsets: whenever an automorphism μ of G is discovered, we check, for every vertex-subset $X \subseteq V(G)$ whether $\mu(X)$ is lexicographically smaller than X. If so, we know that X cannot be orbitally minimal. Think of it this way: Imagine an initially edgeless graph H for which $V(H) = 2^{V(G)}$, and whenever an automorphism of G is discovered, we add the edge $X\mu(X)$ for every $X \subseteq V(G)$. Clearly, after a generator of $\mathrm{Aut}(G)$ is found, any two subsets of $V(G)$ which are in one subset-orbit are in one component of H, as the appropriate automorphism can be re-created by stepping along the appropriate edges, effectively taking the composite of the corresponding automorphisms. Relatively simple use of pointers and light recursion can be used to effectively keep track of the lexicographically smallest element in each component of the implicit H graph, so this strategy can be used to map out orbitally minimal subsets given that a generator for $\mathrm{Aut}(G)$ is discovered, however, this also requires us to keep track of the current status of all possible subsets and update them accordingly every time an automorphism of G is found, which takes at least $\mathcal{O}(2^n)$ storage space and runtime to maintain. Once again, canonization-specific information can be used to somewhat mitigate the amount of processing required, as certain sizes of vertex-subsets can be known not to generate any further graphs on the next level, whether it is because of the canonical form or any additional graph-property filters, but this is still an exponential amount of required space. Thankfully, as stated before, the generational algorithm can only feasibly run for an n so high, As the sheer number of non-isomorphic graphs becomes a computational hurdle long before the subsets become unmaintainable, not to mention that they multiply just as much as subsets do.

Claim 5.1. $|\mathcal{C}_{n+1}^*| \geq 2|\mathcal{C}_n^*|$

To see this, simply consider that given an X collection of non-isomorphic graphs, we can either add a vertex of maximal or 0 degree to create a new unique graph, giving us a collection of $2|X|$ non-isomorphic graphs. In reality of course, the number of non-isomorphic graphs grows way faster.

n	$\lvert\mathcal{G}_n\rvert$	$\lvert\mathcal{C}_n^*\rvert$	Lab. time $[ms]$	Unlab. time $[ms]$
3	8	4	1.3	30.7
4	64	11	3.5	51.5
5	1024	34	39.2	160.8
6	32768	156	1256	1065
7	2097152	1044	84153	5743
8	$2.6 * 10^8$	12346	$5.9 * 10^7$	66328
9	$6.8 * 10^{10}$	274668	–	$1.7 * 10^6$
10	$3.5 * 10^{13}$	12005168	–	$9.1 * 10^7$

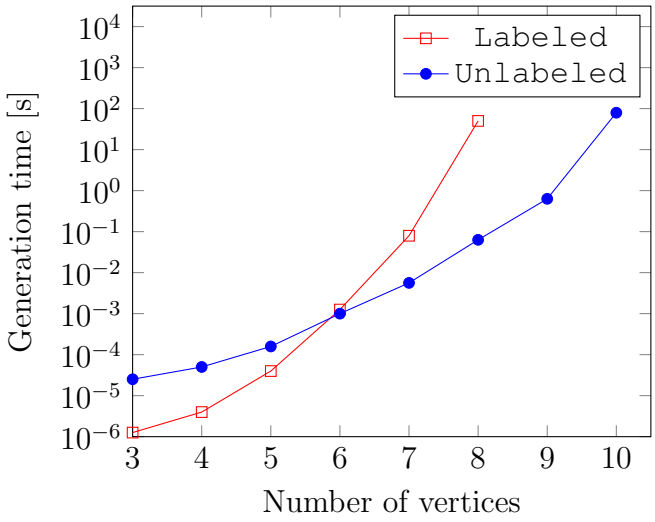


Figure 17: Runtimes for labeled and unlabeled graph generation using our generation tool.

As visible on Figure 17, the additional computations required to canonize every graph encountered during recursive generation quickly turns a profit around 7 vertices, at which point enough graphs are discarded on every level such that the sheer number of possible labeled graphs outweighs the higher runtime required at each step. While staying manageable for longer than labeled graph generation, isomorphism-invariance still only gets you so far, as the numbers become increasingly big even then. The addition of further vertex-hereditary properties, such as the ones described in Section 4, can make the generation of specific graph families even quicker.

5.3 k-equitable partitions

Aside from generally delving into the ideas and methods employed in graph canonization, a main aspect of our research was experimenting with ways of potentially making graph canonization faster with the employment of certain heuristic improvements to existing algorithms. In particular, we took a look at a few ideas regarding the modification of the refinement procedure in McKay's algorithm. In this section, we propose a generalization of the equitability condition that can necessitate finer partition splitting on certain graphs, and an isomorphism-invariant

vertex-classifying function based on Markov-Chain distributions. While these improvements are not groundbreaking, we believe that they can provide a useful tool for deciding the isomorphism of graphs with many symmetrical attributes, as they are able to detect anomalies in a partition that the usual notion of equitability condition would deem equitable.

In [17], McKay proposes the equitability condition for the refinement of partitions within the canonization search-tree of the input graph, and in Section 3.4, we have already discussed the utilization and benefits of the property. This refinement performs exceedingly well in practice for smaller graphs, however, it is a somewhat short-sighted condition in the sense that it only considers direct edge-connections within a graph, and does not immediately identify differences between vertices that are only revealed after more complex connections between vertices have been made apparent through individualization.

The idea of modifying the refinement condition is not entirely new. Tabak [23] for instance examined a refinement process which utilizes the distances of vertex groups to gleam information about the partitions. Here, we will define a generalization of the equitability property by the use of walks of different length within the graph.

5.3.1 Defininitions, connection to equitability

Recall the definition of equitability: π is equitable exactly when $d(x_1, Y) = d(x_2, Y)$ for any cells $X, Y \in \pi$ and for $x_1, x_2 \in Y$. A different way of writing this down would be that for all $x \in X$ the number of *1-long walks* starting from x and ending in Y is constant. From here, two generalizations of the equitability condition can be gleamed, with one being a further, even stricter generalization of the other.

Definition. Given a graph G, we call a partition π of $V(G)$ k-equitable if for any cells X, Y, there is a constant $c_{X,Y}$ such that for any $x \in X$, we have $|\{y \ : \ y \in Y, \ \exists w : \ w \text{ is a } k\text{-long walk starting in } x \text{ and ending in } y\}| = c_{X,Y}$.

That is, walks staring in X need to have the same number of endpoints in Y no matter what atom we start from within X.

Definition. Given a graph G, we call a partition π of $V(G)$ strongly k-equitable if for any cells X, Y, we have a multi-set $M_{X,Y}$ such that for any $x \in X$, we have $\{w_k(x, y) \ : \ y \in Y\} = M_{X,Y}$, where $w_k(x, y)$ is the number of unique k-long walks between x and y.

That is, walks starting in X will lead to the same "types" of endpoints within Y no matter what atom we start from within X.

Claim 5.2. Let us take a graph G and a partition π of $V(G)$. Then: π is strongly k-equitable, then π is k-equitable.

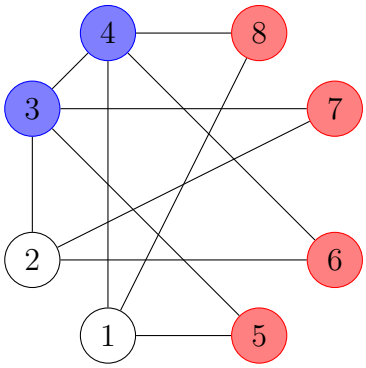
It is not difficult to see that for any $X, Y \in \pi$, the number of non-zero elements within $M_{X,Y}$ is a good candidate for $c_{X,Y}$.

Claim 5.3. The following properties are equivalent for any graph G and partition π of $V(G)$:

- π is equitable,
- π is 1-equitable,
- π is strongly 1-equitable

This also trivially follows, since 1-long walks are simply edges, and thus the walk-information evaluated by the new conditions is simply once again the degree-information examined by the simple equitability condition, the constant $c_{X,Y}$ is equal to the $d(x, Y)$, which all $x \in X$ share, and the $M_{X,Y}$ multi-set will be a collection of $c_{X,Y}$ ones and $|Y| - c_{X,Y}$ zeroes. However, the equivalency ends at $k = 1$. There are graphs that, while equitable, are not 2-equitable, which we will demonstrate through an example on 8 vertices.

$$\pi = (\{1, 2\}, \{3, 4\}, \{5, 6, 7, 8\})$$



v	{1,2}	{3,4}	{5, 6, 7, 8}
1	0	1	2
2	0	1	2
3	1	1	2
4	1	1	2
5	1	1	0
6	1	1	0
7	1	1	0
8	1	1	0

Figure 18: Initial partition.

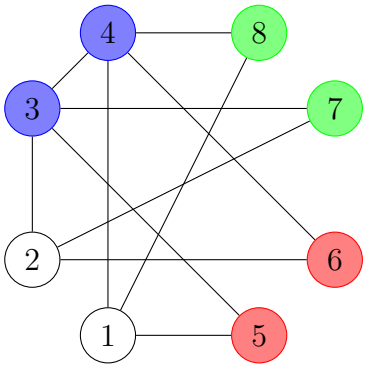
Examine the partition showcased in Figure 18. Here, the number table simply shows the number of neighbors the leftmost vertex v has in a given cell C. Clearly, the equitability condition is satisfied for all cell-pairs, but let us consider what elements of $\{3, 4\}$ can be reached from $\{5, 6, 7, 8\}$ in two steps. When starting form the vertices 7 or 8, we can reach both blue vertices, for instance, $7 - 2 - 3$, $7 - 3 - 4$ and $8 - 4 - 3$, $8 - 1 - 4$ are suitable walks. However, starting from 5 or 6, there is no way to reach both vertices. From 5 we can only reach 4, and from 6 we can only reach 3. Therefore, the pair $\{5, 6, 7, 8\}, \{3, 4\}$ violates the 2-equitability condition, and the partition can be split by the number of ending points walk starting from the former can have in the latter.

In the table of Figure 19, we can now see both $d(v, C)$, as well as the number of elements in C that can be reached from v through a 2-long walk. Clearly, 2-equitability holds.

5.3.2 Implementation, results

Our approach to computing a k-equitable refinement to a given partition π is to check the number of endpoints belonging to k-long walks that start in X, and and in the vertices to other cells alongside the checking of regular degrees into cells that

$$\pi = (\{1,2\},\{3,4\},\{5,6\},\{7,8\})$$



v	{1,2}	{3,4}	{5, 6}	{7, 8}
1	0,1	1,2	1,1	1,1
2	0,1	1,2	1,1	1,1
3	1,2	1,1	1,1	1,2
4	1,2	1,1	1,1	1,2
5	1,1	1,2	0,1	0,2
6	1,1	1,2	0,1	0,2
7	1,1	1,2	0,2	0,1
8	1,1	1,2	0,2	0,1

Figure 19: Initial partition.

are computed during regular equitability checking. This way, the condition can be ensured in much the same manner, except that we compare and sort vectors of length k when deciding whether the current out-cell violates the k-equitability condition or not, instead of simple degrees, that is, vectors of length 1 during usual equitability refinement. In order to efficiently perform this task, some preprocessing will be beneficial. During regular equitability refinement, we often needed to check whether an element form the out-cell is connected to an element of the in-cell, and we similarly often need to check whether there is a k-long walk spanning from one element to another in the case of k-equitability, and how many such walks go from one vertex to another in the case of strong k-equitability. In the case of paths, this would be extremely problematic, as even deciding whether a k-long path between two vertices exists is an NP-complete problem, but with walks, we can decide these questions with relative ease. To achieve this, we can calculate and store the following matrices for future use at the beginning of the canonization:

Claim 5.4. Let us have a graph G with adjacency matrix A. Then, $A^k(u,k)$ is the number of k-long walks starting in u and ending in v

Simple induction by $|V(G)|$ can be used to verify this, if A^k contains the right elements, then multiplying by A from the right will extend each walk by a further edge, and increase the count for the appropriate destination edge. From this, the tool for strong k-equitability checking is directly A^k, and that of k-equitability is simply the matrix $\tilde{A}^k \in \{0,1\}^{n\times n}$ for which:

$$\tilde{A}^k(u,v) = \begin{cases} 1: & A^k(u,v) > 0, \\ 0: & A^k(u,v) = 0. \end{cases}$$

In this solution, we once again run into the problem of large numbers: the elements in A^k can get exponentially big, which requires mitigation. Simply replacing the elements of the matrix with their position in an ordering is not desirable like it was in the vertex-hashing algorithm from Section 3.2, as the differences in walk count are the main way this method creates finer partitions, and a lot of different counts

could be lost through this sort of reduction. A more fitting way of reducing data would be to compute elements of A^k modulo M for some chosen modulus, preferably one low enough to keep the elements small enough to comfortably work with, yet high enough such that high variety in the amount of walks counted between vertices, and which decreases the likelihood of an element within A^k to be an actual multiple of M, therefore nullifying the count of walks up until that point. No such problems are encountered when computing $\tilde{(A)}^k$ alone, as the these can easily be computed from $\tilde{(A)}^{k-1}$ by performing a "logical or" multiplication with A, where to compute $\tilde{(A)}^k(i,j)$ we do not multiply the i-th row and j-th columns of the respective graphs, but simply check whether there is a single k index for which $A^k(i,k) = A(k,j) = 1$. To test performance, we used our canonization tool on the same collections of graphs, examining direct runtime with both 2-equitability and regular equitability checking enabled.

Set	Size	$\texttt{Ref}_{eq}$ $[ms]$	$\texttt{Ref}_{2-eq}$ $[ms]$	# finer partitions
$\mathcal{G}_5$	1024	2.2	3	34
$\mathcal{C}_7^*$	1044	3.7	5.1	25
$\mathcal{C}_8^*$	12346	38.9	52.1	35
$\mathcal{G}_6$	32768	70.5	100.8	83
$\mathcal{C}_9^*$	274668	780.4	1204	46
$\mathcal{G}_7$	2097152	4231	6910	216

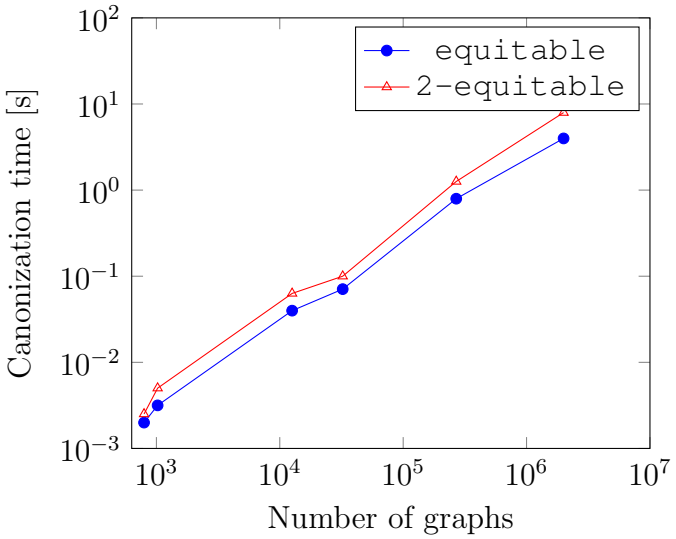


Figure 20: Runtimes and finer partitions produced by equitable and 2-equitable refinement.

In Figure 20, we can see the performance of the algorithm when using the regular 1-equitability refinement, and when 2-equitability is checked Unfortunately, this method of refinement is clearly less efficient for the vast majority of graphs. The additional information that needs to be computed in every refinement step outweighs the impact the finer partitions provide in the cases where higher equitability is useful.

It is interesting to note, however, that there are significantly more useful cases

when we are canonizing labeled graphs, as opposed to the unlabeled case. This is clearly because graphs which are prone to producing partitions that are equitable, yet not 2-equitable are necessarily somewhat symmetric, and therefore have a larger automorphism group. The more symmetries in the graph, the more labeled forms it has, therefore, the more times 2-equitable refinement comes in handy during canonization. While not an overbearing group, cases where graph partitions are finer than they would have been with regular canonization increase as the number of vertices increases, as we can see that the number of finer-than equitable partitions more than doubles over each set of labeled canonizations.

This implies that on larger graphs, this could be a somewhat reliable way of differentiating between graphs which are likely to produce equitable, yet not 2-equitable partitions, such as graphs with similar degree-sequences, or graphs which have already been collected as being "indistinguishable" based on one use of refinement, as the resulting colored graphs contain the same types of color-neighborhoods between vertices, a property which can be easily checked in polynomial time. One example of a way that this can be useful is by providing a quick test for isomorphism between such graphs: if the resulting colored graphs are different in the above way, then they are clearly not isomorphic. When the color-neighborhoods are still similar, regular isomorphism-testing occurs. With the ability to produce finer partitions, more graph-pairs can be identified as not isomorphic, without the need to fully canonize each graph.

5.4 Markov chains, stationary distributions

In this section, we examine a different approach to aiding the refinement of partitions within the graph. We note that the input graph is still assumed to be simple, but in this section we discuss modifications that make supporting graphs non-simple. Rather than attempting to modify or adjust the equitability condition itself, we instead split the partition by another isomorphism-invariant vertex function, that being the values of a stationary distribution of a Markov-chain defined on the vertices of G.

5.4.1 Defining the Markov-chain

Rudimentary knowledge of Markov-Chains from the reader is assumed, but let us quickly go over the basics. A *Markov-chain* is a certain type of sequence, where the probabilities for what the next value in the sequence may be is only determined by the last value taken on. In short, given only the current value in the sequence, we can say exactly what the probability is that a certain value will be taken on next. One of the most common ways of defining a finite-element Markov-chain is through the use of a directed graph $D = (V, A)$, with edge weights $p : A \mapsto \mathbb{R}$ such that for any $u \in V$, $\sum_{uv \in A} p(uv) = 1$. The chance of ending up in position v exactly n steps after being in v is denoted as $p^{(n)}(u, v)$. The markov-chain defined by D is straightforward: the values the chain can take on are the vertices of the graph, and from any value u, we can "step into" a neighboring value v with probability $p(uv)$, often denoted as p_{uv}. The matrix $P = \{p_{uv}\} \in \mathbb{R}^{n \times n}$ is called the *transition matrix* of the Markov-Chain. Two Markov-chains with transition matrices P_1, P_2

are isomorphic if there is some bijection μ from the elements of the first one to the second such that $p_1(x, y) = p_2(\mu(x)\mu(y))$.

A certain attribute a Markov-chain can possess that we can make use of when it comes to graph canonizations is that they can have very particular and isomorphism-invariant vertex-attributes that can be computed very efficiently under the correct circumstances. In particular, we will examine how we can make use of the stationary distribution of a Markov-Chain.

Definition. For a Markov-chain defined on the elements of X with probability matrix P, the vector $\alpha \in \mathbb{R}^X$ is a *stationary distribution* of the Markov-chain if for any $x \in X$, we have $\alpha(x) = \sum_{y \in X} \alpha_y p_{yx}$.

Basically, if the chance of the current value being x is $\mu(x)$ for all $x \in X$, then this is also the chance of the next value in the chain being x. Another way to think about it is how likely it is that a specific value is the current one after a near-infinite number of steps. For Markov-chains on finite elements, such as in the case of the directed graph, this means an n-long column vector α for which $\alpha^T = \alpha^T P$, or, to be more conventional, $P^T\alpha = \alpha$, that is, any $\mu \in [0, 1]^n$ for which $(P^T - I)\alpha = 0^n$, and $\sum_{i=1}^{n} \alpha_i = 1$.

If the stationary distribution is unique, it can be computed efficiently with proper linear equation solvers, so the idea is to use the cell-splitting idea described in Section 3.3 with the distribution itself. construct a Markov-chain on the vertices of the input graph, and compute the α stationary distribution, then split the elements of $V(G)$ based on the α value they receive. Clearly, if isomorphic graphs with isomorphic partitions lead to isomorphic Markov-chains, then the method will indeed be isomorphism-invariant, as these chains have similar stationary distributions.

Ensuring uniqueness The basics of the Markov-chain are fairly straightforward in design: the main elements are those of $V(G)$, and the transition matrix should be somehow inferred from $|E(G)|$ and π, with the more uniqueness harvested from these the better, as we desire as much difference in distribution values as possible. As mentioned earlier, the above approach only works if the Markov-chain we construct has a unique stationary distribution. The initial attempt at defining a Markov-chain would be the following: Let $M(G)$ be the markov-chain whose elements are the vertices of G, and in which we simply step into any neighboring vertex with equal chance.

Claim 5.5. $G \sim H \iff M(G) \sim M(H)$

This is most easily verified by considering how the non-zero elements of the adjacency matrix of G and the transition matrix of $M(G)$ are in the same places, and they move around together when the elements are permuted, which includes the case of isomorphisms.

This is a good start, but not good enough, as it is possible that the resulting Markov-chain has several different stationary distributions. The most simple example would be any non-connected graph. Any stationary distributions calculated

separately on the components could be combined convexly in order to achieve somewhat arbitrary ordering on the vertices, and since we want to compute the distribution trough the use of a linear equation system, the order in which the conditions are listed can impact which possible distribution we end up computing.

Luckily, there is a simple way to ensure that the stationary distribution of our Markov-chain is unique without needing to check for connected components or other conditions, and without sacrificing any information provided by the structure of the chain.

Definition. A Markov-chain is *ergodic* if there exists an $n_0 \in \mathcal{N}$ such that for any elements u, v of the chain (these can be the same) and $n > n_0$, we have $p^{(n)}(u, v) > 0$.

We care about this because one of the most crucial and often used theorems in the field of stochastic processes is the following claim:

Theorem 5.1. [24] For an ergodic Markov-chain, the stationary distribution is unique.

In the case of our graph-based chain, this would mean that the initial G graph needs tp be connected, and for some n_0, there must be an n-long walk between any two vertices for any $n > n_0$. Both of these conditions can be easily satisfied by expanding the graph. Let G^z be the graph resulting by adding an additional vertex z to G, and connecting it to all previously existing vertices within $V(G)$, as well as putting a unique loop edge on z.

Claim 5.6. $G \sim H \iff G^z \sim H^z$.

proof: If $G \sim H$, the other condition trivially holds true. If $G^z \sim H^z$, simply consider that z is the only vertex in both graphs with a loop, thus, any isomorphism between the two leaves it in place, thus the rest of the isomorphism proves $G \sim H$. □

Theorem 5.2. $M(G) \sim M(H) \iff M(G^z) \sim M(H^z)$

Proof: Because of Claims 5.5 and 5.6, $M(G) \sim M(H) \iff G \sim H \iff G^z \sim H^z \iff M(G^z) \sim M(H^z)$. □

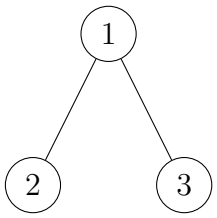
Thus, adding the z vertex does not "muddle" the appearance of the input graph. Now as to how this helps us ensure the correctness of the Markov-chain:

Theorem 5.3. $M(G^z)$ is ergodic for any $G \in \mathcal{G}_n$.

proof: $n_0 = 1$ works, as for any $n \geq 2$, we can simply step into z, travel on the loop in place for $n - 2$ steps, then step into any other element, thus the ergodic condition is satisfied. □

This, combined with the fact that we already established z to not interfere with the uniqueness of the Markov-chain gives us a convenient solution to the distribution uniqueness problem. We simply calculate the μ stationary distribution for $M(G^z)$, and then simply ignore $\mu(z)$ while partitioning the vertices.

In Figure 21, we can see the way a linear equation can be defined to give us the stationary distribution of G^z as a result. The resulting distribution in the example is $\alpha = (0.27, 0.18, 0.18, 0.37)$, which partitions $V(G)$ into $(\{2, 3\}, \{1\})$, while α_z can be ignored.



$$\begin{bmatrix} -1 & 0.5 & 0.5 & 0.25 \\ 0.33 & -1 & 0 & 0.25 \\ 0.33 & 0 & -1 & 0.25 \\ 0.33 & 0.5 & 0.5 & 0.25 \\ 1 & 1 & 1 & 1 \end{bmatrix} \alpha = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Figure 21: Linear equation resulting from simple graph.

5.4.2 Implementation, results

Using partition information While the initial hopes were that the stationary distribution by $M(G^z)$ would immediately split $V(G)$ into well-differentiated subsets of vertices, perhaps even orbits, it quickly became apparent that this model by itself was too simple for this purpose.

Claim 5.7. $\mu(v) = \frac{d_{G^z}(v)}{|E(G)|}$ is a stationary distribution of $M(G^z)$.

Not to worry however, as the nature of the Markov-chain can be altered greatly by simply adjusting the individual probabilities with which we step from certain vertices to others. We can do this with the help of our ordered partition $\pi = (_1, X_2, \ldots, X_k, \{z\})$. Let us say that $c(u)$ is the index of the cell which u is contained in. Edges can then be differentiated by using the c-values of its endpoints.

In our implementation, we used the following strategy: For any vertex u, give every connected edge $uv \in E(G^z)$ a weight of $c(v)$. Then, take $S = \sum_{uv \in E(G^z)} c(v)$ and we define the transition probabilities as: $p_{uv} = \frac{c(v)}{S}$ this simply makes the chances of getting from one vertex to another in any number of steps different based on the cell-based connections between the two.

Initial testing involved the utilization of the index of both endpoints, in order to completely differentiate between any two edges at all if they were not going between the same cells, but in our results, this turned out ot not be beneficial for the purposes of cell refinement, and simply complicated the examination of results and computation, as the differences between numbers becomes smaller as more complexity is attempted to be used to differentiate between he probabilities of steps. The important thing in efficient refinement based on stationary distributions is to simply differentiate between the probabilities of the outgoing edges of a vertex when deciding where to step next.

Using distant neighbors Through initial testing, it was found that on its own, the stationary distribution of the previously described Markov-chain still does not give finer partitions than the equitability refinement, which would be the expected recompensation for the greater amounts of computation necessitated by the linear equation systems, however, we can further specify our Markov-chain to potentially make it even stronger. Recall from Section 5.3 the matrices A^k and $\tilde{(A)}^k$, which describe the relationships of graphs based on the k-long walks between them. Consider the following: for $i = 2, \ldots, k$, we add additional edges to G^z for each u, v for whom $\tilde{(A)}^k(u, v) = 1$, that is, there is a k-long walk between u and v. This edge

can then be given a weight similar to how we did it previously based on the cell of the recipient vertex, as well as k to differentiate between each type of connection between any two differing edge from a single staring point. These edges may even be given different values based on $A^k(.,.)$, that is, the actual number of k-long walks between them, although this aspect of Markov-chain modification was not explored.

We found that doing this can indeed result in the Markov-chain producing a stationary distribution that gives us a finer partition than regular equitability refinement would, in particular, it capable of identifying and splitting up partitions that are not k-equitable.

To test efficiency, we ran our canonization tool both with and without the help of the stationary distribution invariant. The linear equation were solved using the `Eigen` linear algebra tool for `C++`.

Set	Size	$\texttt{Ref}_{eq}$ $[ms]$	$\texttt{SD} + \texttt{Ref}_{eq}$ $[ms](*)$	# finer partitions
$\mathcal{G}_5$	1024	2.2	4.14	0
$\mathcal{C}_7^*$	1044	3.3	6	1
$\mathcal{C}_8^*$	12346	37.1	67.3	15
$\mathcal{G}_6$	32768	61.8	111.7	0
$\mathcal{C}_9^*$	274668	751.4	1191	99
$\mathcal{G}_7$	2097152	4242	7752	105

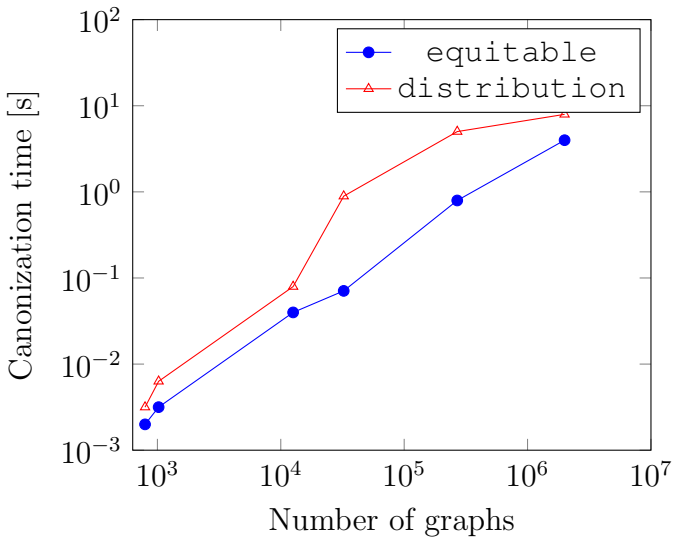


Figure 22: Runtimes and finer partitions produced by equitable refinement with and without the distribution invariant.

See Figure 22 for the runtimes resulting from our implementation of this vertex-invariant. [1]

[1]The time spent on the actual solving of the linear equation system was omitted from the figure, even though it constituted a significant portion of computation time, roughly an additional 50% to what is listed. This is because even though practical and efficient for case-by-case use, the solving subroutine of `Eigen` reserves memory for new assisting data structures with each call, which, when done such a high numbers of times can take up a deceptively large amount of time, something that was duly noted during development of our own implementation of the search-tree algorithm. With

These results clearly show that this approach to distribution-based vertex partitioning is generally inferior to equitability refinement. Firstly, it takes significantly more time to compute, as not only do we need to compute the solution of a sizable linear equation system, but the resulting partitions, while sometimes finer than what equitability would provide us, are often times coarser instead, given how the specific edge-connections between vertices do not properly differentiate with the use of this type of arithmetic. Secondly, even with the addition of higher-level neighborings into the transition matrix, the method is still not quite as adept at picking out k-equitable partitions as the regular modified equitability checker, as clearly visible on the smaller graph sets. In addition to all this, even after the distribution is used to split the cells of the partition, since the partition is often not equitable yet, regular equitability refinements still benefits us often enough to the point where full utilization of it is preferable, even though every new cell created from the previous method will now need to be treaded as active, meaning that very little work was saved for the refinement procedure, despite the trouble we went through to acquire the semi-refined partition. This is a good demonstration as to not overdo node-by-node examination in the search-tree. Even though this method results in smaller search trees for certain very specific graph types, it still simply results in gross additional computation in the vast majority of cases, as well as being downright detrimental in many.

5.4.3 An iterative approach

Clearly, the extra computations that come with the solving of a linear equation system are simply not worth the trouble, not only because it is generally slower to compute while offering little compensation in the form of occasionally finer partitions, but also because it is not dynamic in terms of differentiating vertices as more information is gathered while refining. When to vertices that are in one cell have different distribution values, they are deemed different, but this is not further utilizable in differentiating them the way it would cause a chain-reaction of resulting violating cell-pairs teh way it would with equitability refinement, without the explicit calculation of a new stationary distribution based on the new cell-distribution.

We can somewhat remedy these problems by taking a different approach to "finding" the distribution in question. Earlier we mentioned how the stationary distribution α may be thought of as the "limit" distribution of a Markov-chain, that is, there is some starting distribution α^0, for which continually stepping with the distribution, that is, taking $\alpha^{i+1} = P^T\alpha^i$, we have $\lim_{i\to\infty} \alpha^i = \alpha$. Notice how as long as the initial α^0 distribution is chosen in an isomorphism-invariant manner, the resulting distributions all behave similarly between isomorphic graphs, and their computation requires considerably less effort than that of the actual stationary distribution. So the idea is to approximate the stationary distribution by giving an isomorphism-invariant initial distribution α^0, such as by differentiating vertices by the cell they are in to provide further initial diversity, and then iterating the above process for as long as we wish.

---

the proper structuring and memory allocation, this additional time can be assumed to be much smaller, though still not remotely negligible.

This approach is reminiscent of the way neighborhoods were aggregated in 2 to produce unique strings in each cycle, except here it is not the concatenation of strings, but the linear combination of real numbers. Interestingly, we have found that this method of distribution-refinement is actually more apt for the differentiation of vertices: the same fineness that the stationary distribution provides is reached after just one iteration step in a vast majority cases, meaning that much less computation is required that if the entire linear equation needed to be solved.

Set	Size	$\texttt{Ref}_{eq}[ms]$	$\texttt{SD} + \texttt{Ref}_{eq}\ [ms](*)$	$\texttt{SD}_{it}(1) + \texttt{Ref}_{eq}\ [ms]$	# f. p.
$\mathcal{G}_5$	1024	2.2	4.14	3.9	0
$\mathcal{C}_7^*$	1044	3.3	6.3	5.8	1
$\mathcal{C}_8^*$	12346	38.9	67.3	60.3	16
$\mathcal{G}_6$	32768	61.8	111.7	100.8	0
$\mathcal{C}_9^*$	274668	751.4	1191	1152	103
$\mathcal{G}_7$	2097152	4242	7752	6415	105

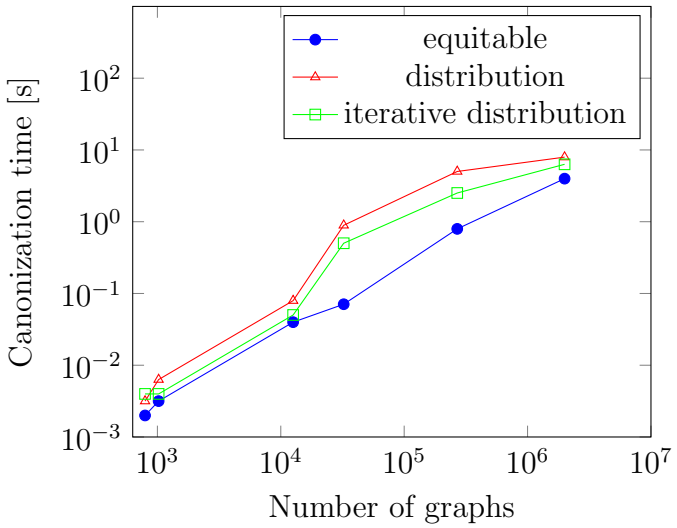


Figure 23: Comparing distribution-invariant methods.

See Figure 23 for a comparison of distribution methods. Clearly, this is still way slower than regular equitability refinement, but for the purposes of targeted partition refinement, this an improvement over the stationary distribution method. Aside from the complete dismissal of the lengthy linear equation solving, the overall search is also noticeably quicker. The number of times partitions are finer than equitable also increases, and the times when they are coarser are much fewer between. Another thing to keep in mind is that with the identity $\alpha^k = P^T\alpha^{k-1} = P^TP^T\alpha^{k-2} = \cdots = (P^T)^k\alpha^0$, we can reduce the entire computation of αk to a single matrix-vector multiplication after $(P^T)^k$ is computed, something which can be efficiently achieved with parallel computing when many processors are available.

Closing remarks

While these heuristics did not turn out to directly speed up the canonization of general graphs, the ability to create finer-than-equitable partitions with two possible methods is still remarkable. Many graph groups can be identified simply by the identification of many differing vertices as seen in Section 3.2, and with k-equitability being a natural extension of the hashing algorithm with the general realization of the equitability-refinement procedure, it may be useful in the future for identification and efficient canonization of certain groups of graphs.

Our canonization implementation and improvement attempts are just a few in a line of many attempts at tackling the problem of efficient graph canonization throughout history, see Section 2.3 for a partial list of the most major results in canonization that have come from such attempts. Readers are recommended to read up on various experimental canonization improvement attempts such as the ones in Sections 5.3 and 5.4, to check out and attempt to comprehend some of the most well-performing canonization tools such as `nauty` to see what something like this actually looks like in practice, and to ponder how one could canonically label a graph in a way previously unthought of. One can never know.

References

[1] Jakob L. Andersen and Daniel Merkle. A generic framework for engineering graph canonization algorithms, 2020.

[2] Vladimir L. Arlazarov, Ivan I. Zuev, Anatoly V. Uskov, and IA Faradzhev. An algorithm for the reduction of finite non-oriented graphs to canonical form. *USSR Computational Mathematics and Mathematical Physics*, 14(3):195–201, 1974.

[3] Vikraman Arvind, Bireswar Das, and Johannes Köbler. The space complexity of k-tree isomorphism. In *Algorithms and Computation: 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007. Proceedings 18*, pages 822–833. Springer, 2007.

[4] László Babai. Canonical form for graphs in quasipolynomial time: preliminary report. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 1237–1246, 2019.

[5] László Babai and Ludik Kucera. Canonical labelling of graphs in linear average time. In *20th annual symposium on foundations of computer science (sfcs 1979)*, pages 39–46. IEEE, 1979.

[6] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 171–183, 1983.

[7] Hans L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11(4):631–643, 1990.

[8] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996.

[9] Derek G Corneil and Calvin C Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM (JACM)*, 17(1):51–64, 1970.

[10] 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*, pages 149–154, 2008.

[11] Adrianus Johannes Wilhelmus Duijvestijn. Electronic computation of squared rectangles. 1962.

[12] Michael Elberfeld and Pascal Schweitzer. Canonizing graphs of bounded tree width in logspace. *ACM Transactions on Computation Theory (TOCT)*, 9(3):1–29, 2017.

[13] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop*

on Algorithm Engineering and Experiments (ALENEX), pages 135–149. SIAM, 2007.

[14] Alpár Jüttner and Péter Madarasi. A graph isomorphism invariant based on neighborhood aggregation. *arXiv preprint arXiv:2301.09187*, 2023.

[15] William Kocay. On writing isomorphism programs. In *Computational and Constructive Design Theory*, pages 135–175. Springer, 1996.

[16] Daniel Lokshtanov, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. Fixed-parameter tractable canonization and isomorphism test for graphs of bounded treewidth. *SIAM Journal on Computing*, 46(1):161–189, 2017.

[17] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium vol. 30*, pages 45–87, 1981.

[18] Brendan D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998.

[19] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of symbolic computation*, 60:94–112, 2014.

[20] BRENDAN DAMIEN McKAY. *Backtrack programming and the graph isomorphism problem*. PhD thesis, University of Melbourne, 1976.

[21] Harry L. Morgan. The generation of a unique machine description for chemical structures-a technique developed at chemical abstracts service. *Journal of chemical documentation*, 5(2):107–113, 1965.

[22] Nikolai Nøjgaard. *Graph Theoretical Problems in Life Sciences*. PhD thesis, 2020.

[23] Tabak Pamela. *Distance based canonical labeling algorithms with applications to graph matching*. PhD thesis, Universidade Federal do Rio de Janeiro, 2020.

[24] Alessandro Panconesi. The stationary distribution of a markov chain. *Unpublished note, Sapienza University of Rome, http://www.dis.uniroma1.it/ leon/didattica/webir/pagerank.pdf*, 2005.

[25] Ronald C. Read. The coding of various kinds of unlabeled trees. In *Graph theory and computing*, pages 153–182. Elsevier, 1972.

[26] Ronald C. Read. *Graph theory and computing*. Academic Press, 2014.

[27] Ákos Seress. *Permutation group algorithms*. Number 152. Cambridge University Press, 2003.

Alulírott Nagy Szabolcs Ákos nyilatkozom, hogy a szakdolgozatom elkészítése során semmilyen MI alapú eszközt nem használtam bármilyen célból.