

Parallel Canonical Labelling for Graph Isomorphism Testing

Jim Haslett, Daniel Grosu

Department of Computer Science, Wayne State University, USA

Detroit, USA

{haslettj, dgrosu}@wayne.edu

Abstract—The graph isomorphism problem has been studied for decades, without finding a polynomial time solution for the general case. The best known bound is quasipolynomial. However, practical solutions for solving the problem through canonical labelling, by individualization and refinement, have been shown to be very efficient for most classes of graphs. Here, we consider the individualization-refinement paradigm, as implemented in the *Nauty* software package, which is the baseline for most other software packages in this space. We design a message-passing canonical labelling parallel algorithm based on the sequential algorithm used by *Nauty*. Our parallel implementation is shown to produce identical results to the sequential implementation, while achieving a good speedup on several classes of graphs that are challenging for the baseline sequential algorithm.

Index Terms—graph isomorphism, message-passing, parallel algorithms

I. INTRODUCTION

The *graph isomorphism problem* is to determine whether an isomorphism exists between two given graphs. An *isomorphism* between two graphs is a one-to-one correspondence between their vertex sets, such that whenever two vertices are adjacent in one of the graphs, the corresponding two vertices are adjacent in the other graph. More formally, it is defined as follows [1].

Definition 1: An *isomorphism* from a simple graph $G = (V_G, E_G)$ to a simple graph $H = (V_H, E_H)$ is a bijection $f : V_G \rightarrow V_H$ such that $(u, v) \in E_G$ iff $(f(u), f(v)) \in E_H$.

The graph isomorphism problem is in NP. However, it is not believed to be NP-complete. There are no known polynomial time algorithms to solve the problem, for the general case. The best known algorithm, due to Babai [2], runs in quasipolynomial time, $\exp((\log n)^{O(1)})$, where n is the number of vertices of the graphs.

Our approach for solving the graph isomorphism problem, is using *canonical labelling*. Instead of directly comparing two input graphs, each graph is relabelled according to a deterministic algorithm. The adjacency matrices of the two relabelled graphs are then compared. If the matrices are identical, the graphs are isomorphic, otherwise they are not.

The starting point for this approach is the software package *Nauty*, an optimized library for canonically labelling a graph and finding generators for its automorphism group [3]. *Nauty* uses a backtracking algorithm that incrementally individualizes and refines partitions of the vertex set of the graph. This partitioning naturally forms a search tree that is recursively

explored in depth first order. The algorithm is efficient in practice, utilizing discovered automorphisms to prune the search tree.

In this paper, we propose a parallel algorithm based on the *Nauty* algorithm. Our *main contributions* are:

- 1) We design a message-passing parallel algorithm for computing a canonical labelling of a graph. This approach can be used to test for the existence of an isomorphism to a second input graph by comparing their canonical labelings.
- 2) We implement our parallel algorithm in C, using OpenMPI. Our code is publicly available on GitHub [4].
- 3) We provide a performance analysis using benchmark graphs, provided by the *Nauty and Traces* website [5], as well as a set of generated Miyazaki graphs [6]. Our experiments show that, for classes of regular graphs, the algorithm achieves a reasonable speedup.

II. BACKGROUND

In this section, we introduce basic concepts that are used throughout the paper. The notation and definitions follow McKay [3], and McKay and Piperno [7].

We consider a graph $G = (V, E)$, where $V = \{1, 2, \dots, n\}$ is the set of vertices, $n = |V|$, and E is the set of edges. We denote by $\mathcal{G}_V$ the set of all graphs with n vertices. The set E of edges of a graph $G \in \mathcal{G}_V$ consists of pairs of vertices (i, j) where $i, j \in V$ and $i \neq j$. If $(i, j) \in E$, then i and j are adjacent. The *adjacency matrix* A of G is an $n \times n$ matrix where $A_{ij} = 1$ if $(i, j) \in E$, and $A_{ij} = 0$ if $(i, j) \notin E$.

Definition 2: A *partition* π of V is a collection $\{C_1, C_2, \dots, C_k\}$ of cells, where the cells C_i are disjoint, non-empty subsets of V whose union is V . An *ordered partition* of V is a sequence of cells $(C_1, C_2, \dots, C_k)$ where $\{C_1, C_2, \dots, C_k\}$ is a partition of V .

A partition π can be viewed as a *coloring* of V in the sense that π assigns the same color to vertices $i, j \in V$ if and only if i and j are in the same cell. A *colored graph* is a pair (G, π) , where π is a partition (coloring) of G . The set of all partitions of V is denoted by Π_V , while the set of all ordered partitions of V is denoted by $\tilde{\Pi}_V$. A *fixed* cell of π is a cell with a cardinality of 1, the element of this cell is said to be *fixed* by π .

Definition 3: Given two ordered partitions, $\pi_1, \pi_2 \in \tilde{\Pi}_V$, π_1 is *finer than or equal to* π_2 , denoted $\pi_1 \preceq \pi_2$, if every cell of

π_1 is a subset of a cell of π_2 . If $\pi_1 \preceq \pi_2$ and $\pi_2 \preceq \pi_1$, then $\pi_1 \simeq \pi_2$.

Given an ordered partition $\pi \in \tilde{\Pi}_V$, the *length* of π , denoted by $|\pi|$, is the number of cells in π . An ordered partition π is *discrete* if $|\pi| = n$. A discrete partition is a permutation of V .

Given $G \in \mathcal{G}_V$, $W \subseteq V$, and $v \in V$, the *degree of vertex v with respect to the set W of vertices in G* , denoted by $d_G(v, W)$, is the number of vertices in W that are adjacent to v .

Definition 4: A partition $\pi \in \tilde{\Pi}_V$ is *equitable* in G if for all u, v in the same cell $C_i \in \pi$, $d_G(u, C_j) = d_G(v, C_j)$ for all $i, j \in \{1, 2, \dots, |\pi|\}$.

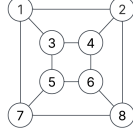


Fig. 1: Example graph, G .

We illustrate these concepts considering the graph G given in Fig. 1. An example of ordered partition of length 2 in G is $\pi_1 = (C_1, C_2) = ((1), (2, 3, 4, 5, 6, 7, 8))$. Cell $C_1 = (1)$ is a fixed cell and element 1 is fixed by π_1 . Partition $\pi_2 = ((1), (2), (3), (4), (5), (6), (7), (8))$ is a discrete partition and π_2 is finer than or equal to π_1 , i.e., $\pi_2 \preceq \pi_1$. The degree of vertex 4 with respect to $W = \{1, 6, 7, 8\}$ is $d_G(4, W) = 1$. Partition $\pi_2 = ((1, 2, 7, 8)(3, 4, 5, 6))$ is equitable.

Let S_n be the symmetric group (the group of all permutations) on the set of n elements (vertices) of V . We denote by v^σ the image of v under σ , where $v \in V$ and $\sigma \in S_n$. If σ acts on a set of vertices $W \subseteq V$ (denoted by W^σ), then it acts on every vertex of W . If it acts on a permutation π (denoted by π^σ), then it acts on every cell of π . If it acts on a graph G (denoted by G^σ) then vertex u^σ is adjacent to vertex v^σ in G^σ if and only if u is adjacent to v in G . If (G, π) is a colored graph, then $(G, \pi)^\sigma = (G^\sigma, \pi^\sigma)$.

Definition 5: Colored graphs (G, π) and (G', π') are *isomorphic* (denoted by $(G, \pi) \cong (G', \pi')$) if $\exists \sigma \in S_n$ such that $(G', \pi') = (G, \pi)^\sigma$. Here, σ is an *isomorphism*.

An *automorphism* of a colored graph is an isomorphism to itself. We denote by $\text{Aut}(G, \pi)$ the group of automorphisms of (G, π) , i.e., $\text{Aut}(G, \pi) = \{\sigma \in S_n \mid (G, \pi)^\sigma = (G, \pi)\}$.

Definition 6: A *canonical form* is a function $C : \mathcal{G}_V \times \Pi \rightarrow \mathcal{G}_V \times \Pi$, such that $C(G, \pi) \cong (G, \pi)$ and $C(G^\sigma, \pi^\sigma) = C(G, \pi)$, for all $G \in \mathcal{G}_V$, $\pi \in \Pi$, and $\sigma \in S_n$.

Definition 7: A function $L : \mathcal{G}_V \times \Pi \rightarrow S_n$ which given a colored graph (G, π) it gives the corresponding σ for which $C(G, \pi) = (G, \pi)^\sigma$ is called *canonical labelling*.

Two isomorphic graphs are identical after canonical relabelling.

III. INDIVIDUALIZATION-REFINEMENT PARADIGM

The individualization-refinement paradigm introduced by Parris and Read [8] is a general approach used by several software packages that produce a canonical labelling or the automorphism generators for a graph. In this section, we describe the individualization and refinement procedures and

the sequential algorithm for computing the canonical labelling of a graph. These will serve as the foundations for the design of our parallel algorithm.

A. Refinement

The refinement algorithm, takes as input a graph $G \in \mathcal{G}_V$, an ordered partition $\pi \in \tilde{\Pi}_V$, and an active set $\alpha \subseteq \pi$, and produces an ordered partition $\hat{\pi} \in \tilde{\Pi}_V$ that is the refined partition of the input partition π . It splits the cells of the input partition π into smaller cells, based on the degree of the vertices in each cell with respect to the cells in the active set α .

The refinement algorithm is given in Algorithm 1 REFINEMENT. First, it initializes a new working partition $\hat{\pi}$ to a copy of the input partition π (Line 1). In the main loop of the algorithm, an element of α is assigned to W for this iteration of the loop, and W is removed from α (Lines 3 - 4). The inner loop goes over each cell in $\hat{\pi}$. It calculates the degree of each vertex v in the current cell C of $\hat{\pi}$ with respect to the current cell W of the active set α (Line 6). The vertices of C are then split into new cells, based on the degree with respect to W , in ascending order of their value. It then, replaces the current cell C from $\hat{\pi}$ with the results of the split from Line 6, i.e., $D_1, D_2, \dots, D_s$ (Line 7). If cell C is part of α then it replaces C with $D_1, D_2, \dots, D_s$ in α (Line 9). Otherwise, it determines q , the smallest index of the largest cell in $D_1, D_2, \dots, D_s$, and adds to α all cells D_i except D_q (Lines 11-12). The iterations are executed until either α is empty, or $\hat{\pi}$ is discrete. When the outer loop terminates, the algorithm returns the refined partition $\hat{\pi}$.

The following theorem states the two main properties of the REFINEMENT algorithm: (i) the partition obtained by the algorithm is finer than the input partition, and (ii) the refinement is independent of the labelling of vertices. The theorem is a direct consequence of the REFINEMENT algorithm definition and thus, we will not provide a proof here.

Theorem 1 ([7]): Let $G \in \mathcal{G}_V$, $\pi \in \tilde{\Pi}_V$, $\alpha \subseteq \pi$, and $\sigma \in S_n$, then

Algorithm 1 REFINEMENT(G, π, α)

Input: G : Graph; π : Ordered partition; α : Active set, $\alpha \subseteq \pi$

```

1:  $\hat{\pi} \leftarrow \pi$ 
2: while ( $\alpha \neq \emptyset$ ) and ( $\hat{\pi}$  is not discrete) do
3:    $W \leftarrow$  any element of  $\alpha$ 
4:    $\alpha \leftarrow \alpha \setminus W$ 
5:   for each cell  $C \in \hat{\pi}$  do
6:     Partition  $C$  into subsets  $D_1, D_2, \dots, D_s$  such that
       for any  $u \in D_r$  and  $v \in D_t$ ,  $d_G(u, W) > d_G(v, W)$ 
       if and only if  $r > t$ 
7:      $\hat{\pi} \leftarrow \hat{\pi}$  in which  $C$  is replaced by  $D_1, D_2, \dots, D_s$ 
8:     if  $C \in \alpha$  then
9:        $\alpha \leftarrow \alpha$  in which  $C$  is replaced by  $(D_1, \dots, D_s)$ 
10:    else
11:       $q \leftarrow$  smallest index  $i$  for which  $|D_i|$  is
        maximum in  $(D_1, \dots, D_s)$ 
12:      Add  $(D_1, \dots, D_{q-1}, D_{q+1}, \dots, D_s)$  to  $\alpha$ 
13: return  $\hat{\pi}$ 

```

- (i) $\text{REFINEMENT}(G, \pi, \alpha) \preceq \pi$, and
- (ii) $\text{REFINEMENT}(G^\sigma, \pi^\sigma, \alpha^\sigma) = \text{REFINEMENT}(G, \pi, \alpha)^\sigma$

We show how the refinement algorithm works using the example graph from Fig. 1. The algorithm starts with graph G , a partition $\pi = ((1), (2, 3, 4, 5, 6, 7, 8))$, and the active set $\alpha = ((1))$. After Lines 1-4, we have $\hat{\pi} = ((1), (2, 3, 4, 5, 6, 7, 8))$, $\alpha = \emptyset$, $W = ((1))$. The first iteration of the for loop (Lines 5-12) does not change $\hat{\pi}$ as C_1 is a fixed cell. During the next iteration of the loop, it splits C_2 into $D_1 = (4, 5, 6, 8)$, $D_2 = (2, 3, 7)$ (Line 6). D_1 has the vertices from C_2 with a degree of 0 relative to (1), D_2 those with a degree of 1. It replaces C in $\hat{\pi}$ with the new partition, and thus, $\hat{\pi} = ((1), (4, 5, 6, 8)(2, 3, 7))$. Since $C \notin \alpha$ and D_1 is the largest cell in the new split C , it adds $D_2 = (2, 3, 7)$ to α . Thus, $\alpha = ((2, 3, 7))$. In the next two iterations of the inner loop $\hat{\pi}$ and α do not change. The for loop ends and a new iteration of the while loop starts with $\alpha = (2, 3, 7)$, $W = (2, 3, 7)$, and $\hat{\pi} = ((1), (4, 5, 6, 8)(2, 3, 7))$. At the end of the second iteration of the while loop $\hat{\pi} = ((1), (6), (4, 5, 8), (2, 3, 7))$, and $\alpha = ((4, 5, 8))$. The remaining iterations do not further refine the partition, and the algorithm returns $\hat{\pi} = ((1), (6), (4, 5, 8), (2, 3, 7))$.

B. Individualization

The individualization algorithm takes as input an ordered partition $\pi \in \tilde{\Pi}_V$ and a target vertex v , and returns an ordered partition $\pi' \in \tilde{\Pi}_V$ that is the individualized version of π . Individualization is done by selecting the cell that contains the target vertex and creating a new partition on which the target vertex becomes a fixed cell (composed of itself only).

The individualization algorithm, called INDIVIDUALIZATION, is given in Algorithm 2. It takes as input an ordered partition π and a target vertex v . It first determines the index of the cell in $\pi = (C_1, C_2, \dots, C_i, \dots, C_r)$ containing v (called the target cell) (Line 1). Then, it removes the target vertex from the target cell, and places it in a new cell that is placed into π before the target cell (Line 2). Finally, it returns the resulting partition π' .

We show how the individualization algorithm works using the example graph from Fig. 1. Assume $\pi = ((1), (6), (4, 5, 8), (2, 3, 7))$ and $v = 4$. The target cell is then $(4, 5, 8)$. The target vertex is removed from the target cell, and placed in a new cell before the target cell. The final partition is $\pi' = ((1), (6), (4), (5, 8), (2, 3, 7))$.

C. Search Tree

The individualization and refinement process, which is the core of the canonical labelling algorithm presented in Section III-D, naturally forms a search tree with nodes that

are equitable partitions of V . The root node of the tree is the equitable partition obtained by $\text{REFINEMENT}(G, (V), (V))$, where (V) is the ordered partition of V with a single cell. Any arbitrary node π in the tree has the form $\pi = (C_1, C_2, \dots, C_r)$, where C_i are the cells of the partition. If π is discrete, then it is a leaf node, or *terminal partition*. If π is not discrete, then we choose a target cell $C_i \in \pi$ using the **TARGETCELL** function. The children of π in the tree are the partitions $\text{REFINEMENT}(G, \pi', \{v\})$ for $v \in C_i$, where $\pi' = \text{INDIVIDUALIZATION}(\pi, v)$. These children are defined by the individualization process.

The target cell function (called **TARGETCELL** and shown in Algorithm 3) chooses a target cell $W \in \pi$ from the input partition $\pi \in \tilde{\Pi}_V$. The method of choosing the target cell is implementation dependent, but must be deterministic for the algorithm to work. A simple example is to choose the first non-fixed cell in π that has the smallest size.

Algorithm 3 **TARGETCELL**(π)

Input: π : Ordered partition

- 1: $C \leftarrow$ first non-fixed cell in π with the smallest size
 - 2: **return** C
-

At this point the specifics of the implementation determine how the search tree is explored. Many implementations choose a depth-first search, which is what we will focus on here. Fig. 2, shows part of the depth-first search tree built for the example graph from Fig. 1. The root corresponds to $\pi = ((1, 2, 3, 4, 5, 6, 7, 8))$, while the leftmost child of the root correspond to the refinement of $\pi = ((1), (2, 3, 4, 5, 6, 7, 8))$, which is $\pi = ((1), (6), (4, 5, 8), (2, 3, 7))$. The leftmost leaf corresponds to the terminal (discrete) partition $\pi = ((1), (6), (4), (5), (8), (2), (3), (7))$.

The terminal partitions in the leaf nodes are used as permutations to relabel the graph and produce candidate canonical labellings. These candidates are compared to the current best candidate. If a candidate is better than the current best, according to a defined ordering relation, then the candidate becomes the new best candidate. One possible ordering is defined as follows.

Let X be the set of terminal partitions in the search tree and $\sigma \in S_n$ be the permutation induced by $\pi \in X$. We derive graph G^σ by reordering the vertices of G according to σ . Let $n[G]$ be the adjacency matrix of G written as a one dimensional array in a row-by-row fashion. The array $n[G]$ can be viewed as a n^2 -binary integer. We define the ordering $\triangleright$ as follows, given two graphs G_1 and G_2 , $G_1 \triangleright G_2$ if and only if $n(G_1) \geq n(G_2)$. With this ordering, the *canonical labelling* of G is

$$\sigma = \arg \max_{\triangleright} \{G^\tau \mid \tau \text{ is the permutation induced by } \pi, \forall \pi \in X\}.$$

Pruning the Search Tree. The search tree can grow very large, on the order of $n!$. One method used to reduce the size of the search tree is to prune branches that cannot produce a better candidate for the canonical labelling than the current best candidate. While exploring the search tree, the algorithm will discover automorphisms of the graph, if they exist. These

Algorithm 2 **INDIVIDUALIZATION**(π, v)

Input: π : Ordered partition

- 1: $i \leftarrow$ index of cell in π containing v , where
 $\pi = (C_1, C_2, \dots, C_i, \dots, C_r)$
 - 2: $\pi' \leftarrow (C_1, C_2, \dots, C_{i-1}, (v), C_i \setminus \{v\}, C_{i+1}, \dots, C_r)$
 - 3: **return** π'
-

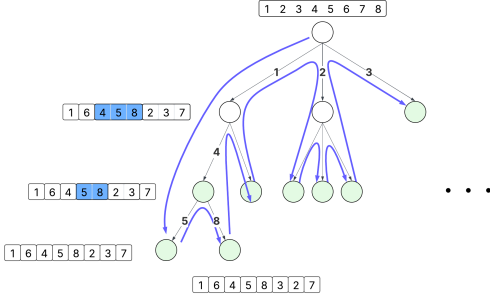


Fig. 2: Example depth first search tree for the individualization and refinement process.

automorphisms can be used to prune entire branches of the search tree, significantly reducing the overall size.

For $\pi \in \tilde{\Pi}_V$, let $\text{mcr}(\pi) = \{\min\{C_i\} \mid C_i \in \pi\}$ be the set of *minimum cell representatives* of π and $\text{fix}(\pi)$ be the set of vertices in V which are fixed by π . For example, if $\pi = ((1), (4, 5, 6, 8)(2, 3, 7))$, then $\text{mcr}(\pi) = \{1, 4, 2\}$ and $\text{fix}(\pi) = \{1\}$. While the tree is being explored, we maintain a set Ψ which contains the pair of sets $(\text{fix}(\gamma), \text{mcr}(\gamma))$ for every automorphism γ that has been discovered. For a given non-terminal node ν in the search tree, we track $W_\nu \subseteq V$ which is initialized to $\text{TARGETCELL}(\nu)$ when ν is first generated. When ν is revisited, W_ν is updated to

$$W_\nu = W_\nu \cap \text{mcr}(\gamma_1) \cap \text{mcr}(\gamma_2) \cap \dots \cap \text{mcr}(\gamma_r),$$

where $\gamma_1, \gamma_2, \dots, \gamma_r$ are the automorphisms in Ψ which fix ν . In this way, the search prunes, or ignores, children of ν where the target vertex $v \notin W_\nu$.

Another method of pruning is by using an *indicator function*. We can define an indicator function $\Lambda : G_V \times \tilde{\Pi}_V \rightarrow \mathbb{Z}^+$, such that $\Lambda(G^\delta, \pi^\delta) = \Lambda(G, \pi)$ for any $\delta \in S_n$. For our case, we will define $\Lambda(G, \pi)$ to be the number of edges of G with both endpoints in the same cell of π . For a given $\tau \in X$ suppose the sequence of equitable partitions from the root node to τ is $\pi_1 \rightarrow \pi_2 \rightarrow \dots \rightarrow \pi_k = \tau$. Then we define the sequence of indicators for τ as

$$\hat{\Lambda}(\tau) = (\Lambda(G, \pi_1), \Lambda(G, \pi_2), \dots, \Lambda(G, \pi_k)).$$

In order for τ_1 and τ_2 to be equivalent, we must have $\hat{\Lambda}(\tau_1) = \hat{\Lambda}(\tau_2)$. Assigning an ordering to the indicator vectors we can redefine the canonical labelling of G as

$$\sigma = \arg \max_{\tau} \{G^\tau \mid \tau \text{ is the permutation induced by } \pi, \forall \pi \in X\}$$

$$\text{and } \hat{\Lambda}(\tau) = \Lambda^*, \text{ where}$$

$$\Lambda^* = \max\{\hat{\Lambda}(\tau) \mid \tau \text{ is the permutation induced by } \pi, \forall \pi \in X\}.$$

With this definition of canonical labelling, we can eliminate branches of the search tree where the indicator vector cannot reach Λ^* .

D. Sequential Canonical Labelling Algorithm

The algorithm that computes the canonical labelling, called CANONICAL LABELLING, is shown in Algorithm 4. The algorithm starts by initializing the working data stores (Lines 1-4). The initial partition that becomes the partition in the root node of the search tree is determined from a one cell partition containing the set of vertices V by refining it against

itself (Line 5). This creates an initial partition π with cells $(C_1, C_2, \dots, C_r)$, where each C_i contains vertices having the same degree with respect to V (i.e., with respect to all the vertices in the entire graph). The cells of π are ordered in increasing order of the degree of their member vertices with respect to G . If this starting partition is discrete, then the algorithm terminates and simply returns the permutation σ derived from the discrete partition π (Lines 6-8). The first target cell C is chosen (by calling the function $\text{TARGETCELL}(\pi)$) (Line 9). Then, for each vertex v in the target cell C , the partition and vertex pair, (π, v) , is pushed onto the stack, $\mathcal{S}$ (Lines 10-11).

The main loop starts in Line 12, and continues until the stack, $\mathcal{S}$ is empty. The partition π to be refined and the vertex v to be individualized, are popped from the stack (Line 13). The algorithm determines if the current tree node and its children should be pruned by checking the partition and vertex against the automorphism store Ψ and the indicator function $\hat{\Lambda}$. If the current node is pruned, then the current loop iteration ends (Lines 14-17). Due to space limitation we do not provide the details of the pruning process in Algorithm 4.

If the current node is not pruned, then the vertex v of partition π is individualized and the resulting partition is stored in π' (Line 18). Then π' is refined against the active set consisting of a single cell containing only the vertex v (Line 19). If the result of this refinement is a discrete partition

Algorithm 4 CANONICAL-LABELLING(G)

Input: G : graph

```

1:  $\hat{G} \leftarrow G$ 
2:  $\mathcal{S} \leftarrow$  empty stack
3:  $\hat{\Lambda} \leftarrow$  empty indicator vector
4:  $\Psi \leftarrow$  empty automorphism set
5:  $\pi \leftarrow \text{REFINEMENT}(G, (V), (V))$ 
6: if  $\pi$  is discrete then
7:    $\sigma \leftarrow \pi$ 
8:   return  $\sigma$ 
9:  $C \leftarrow \text{TARGETCELL}(\pi)$ 
10: for  $v \in C$  do
11:   Push  $(\pi, v)$  onto  $\mathcal{S}$ 
12: while  $\mathcal{S}$  is not empty do
13:    $(\pi, v) \leftarrow$  Pop from  $\mathcal{S}$ 
14:   if  $(\pi, v)$  is pruned by  $\Psi$  then
15:     continue
16:   if  $(\pi, v)$  is pruned by  $\hat{\Lambda}$  then
17:     continue
18:    $\pi' \leftarrow \text{INDIVIDUALIZATION}(\pi, v)$ 
19:    $\pi' \leftarrow \text{REFINEMENT}(G, \pi', ((v)))$ 
20:   if  $\pi'$  is discrete then
21:      $\sigma_1 \leftarrow \pi'$ 
22:     if  $G^{\sigma_1} \triangleright \hat{G}$  then
23:        $\hat{G} \leftarrow G^{\sigma_1}$ 
24:        $\sigma \leftarrow \sigma_1$ 
25:        $\hat{\Lambda} \leftarrow$  indicator vector for  $G^\sigma$ 
26:       Update automorphism set  $\Psi$ 
27:   else
28:      $C' \leftarrow \text{TARGETCELL}(\pi)$ 
29:     for  $v \in C'$  do
30:       Push  $(\pi', v)$  onto  $\mathcal{S}$ 
31: return  $\sigma$ 

```

(tested on Line 20), then the algorithm has reached a leaf node. A permutation σ_1 is derived from partition π' (Line 21) and G is permuted by σ_1 to obtain G^{σ_1} . G^{σ_1} is compared against $\hat{G}$, the current best relabelling of G (Line 22). If G^{σ_1} is better than $\hat{G}$ (i.e., $G^{\sigma_1} \triangleright \hat{G}$) then $\hat{G}$ is updated to this labelling, the current best permutation σ is updated to σ_1 , and the indicator vector $\hat{\Lambda}$ is updated (Lines 23-25). Regardless if the permutation derived at this node represents the best relabelling, the automorphism store Ψ is updated (Line 26). If the partition π' is not discrete (Line 27), then its children are pushed onto the stack by choosing a new target cell C' , and combining the partition and vertex pairs of the vertices of C' , (π', v) for all $v \in C'$ (Lines 28-30).

The algorithm continues until all branches have either been refined or pruned. When the stack becomes empty, the algorithm terminates and returns the permutation σ , i.e., the permutation that produced the best (canonical) relabelling. The algorithm will terminate, given a graph with a finite number of vertices as input. As the tree is explored, there is no opportunity to backtrack to a previously refined tree node. Generating new tree nodes can only be done from the current node using vertices in the target cell. When the partition of a node is individualized, one vertex is placed into its own cell, and no child of this node will be able to individualize this vertex again, leading to a maximum tree depth of n , given a graph with n vertices. Given n is finite, and there is no opportunity to individualize any node more than once per branch, there are at most n branches with a depth at most n , for maximum of $n!$ nodes, the algorithm must terminate.

IV. PARALLEL CANONICAL LABELLING ALGORITHM

The proposed parallel algorithm, presented in Algorithm 5, called PARALLEL-CANONICAL-LABELLING, is designed for a message-passing parallel system. It consists of a single process as a *dispatcher* to coordinate the work, and several *worker* processes. All communication happens between the dispatcher and workers. The same algorithm, PARALLEL-CANONICAL-LABELLING, runs on all processes. Each process has a local, independent copy of all state variables. There is no shared memory between processes.

There are a few tuning parameters that affect the performance of the parallel algorithm.

(i) *Communication period* is the number of tree nodes that a worker process handles before checking for incoming messages.

(ii) *Work send limit* is the maximum number of tree nodes that the dispatcher process may send to a worker process when work is requested. The actual number of nodes sent, may be less.

(iii) *Work split* determines which nodes from the stack are sent to worker processes. A value of 1 means nodes are taken from the top of the stack. A value of 2 means every other node is taken from the top of the stack. In general, a value of i means every i^{th} node is taken from the top of the stack.

Our algorithm utilizes previously described routines, such as REFINEMENT, INDIVIDUALIZATION, and TARGETCELL.

Algorithm 5 PARALLEL-CANONICAL-LABELLING(G)

Input: G : graph

```

1:  $\hat{G} \leftarrow G$ 
2:  $\mathcal{S} \leftarrow$  empty stack
3:  $\hat{\Lambda} \leftarrow$  empty indicator vector
4:  $\Psi \leftarrow$  empty automorphism set
5: if ISDISPATCHER( ) then
6:    $\pi \leftarrow$  REFINEMENT( $G, (V), (V)$ )
7:   if  $\pi$  is discrete then
8:      $\sigma \leftarrow \pi$ 
9:     BROADCAST(work-end)
10:   return  $\sigma$ 
11:  $C \leftarrow$  TARGETCELL( $\pi$ )
12: for  $v \in C$  do
13:   Push  $(\pi, v)$  onto  $\mathcal{S}$ 
14: else
15:   WAIT(work-start)
16: while not RECEIVE(work-end) do
17:   while  $\mathcal{S}$  is not empty do
18:     if Communication Period reached then
19:       CHECKFORMESSAGESANDUPDATE( )
20:       if ISDISPATCHER( ) then
21:         FULLFILWORKREQUESTS( )
22:         Reset Communication Period counter
23:        $(\pi, v) \leftarrow$  Pop from  $\mathcal{S}$ 
24:       if  $(\pi, v)$  is pruned by  $\Psi$  then
25:         continue
26:       if  $(\pi, v)$  is pruned by  $\hat{\Lambda}$  then
27:         BROADCAST(branch being pruned)
28:         continue
29:        $\pi' \leftarrow$  INDIVIDUALIZATION( $\pi, v$ )
30:        $\pi' \leftarrow$  REFINEMENT( $G, \pi', \{v\}$ )
31:       if  $\pi'$  is discrete then
32:         if ISDISPATCHER( ) and ISFIRSTLEAFNODE( ) then
33:           BROADCAST(work-start)
34:            $\sigma_1 \leftarrow \pi'$ 
35:           if  $G^{\sigma_1} \triangleright \hat{G}$  then
36:              $\hat{G} \leftarrow G^{\sigma_1}$ 
37:              $\sigma \leftarrow \sigma_1$ 
38:              $\hat{\Lambda} \leftarrow$  indicator vector for  $G^\sigma$ 
39:             BROADCAST( $\hat{G}$ )
40:             Update automorphism store  $\Psi$ 
41:             BROADCAST(automorphism)
42:         else
43:            $C' \leftarrow$  TARGETCELL( $\pi'$ )
44:           for  $v \in C'$  do
45:             Push  $(\pi', v)$  onto  $\mathcal{S}$ 
46:       if ISDISPATCHER( ) then
47:         if All processes have asked for more work then
48:           BROADCAST(work-end)
49:         else
50:           SEND(work-return)
51:       else
52:         SEND(work-request)
53:         WAIT(for-work)
54: return  $\sigma$ 

```

Communication routines used by the algorithm include BROADCAST, SEND, RECEIVE, and WAIT. BROADCAST sends a message to all processes. This implementation relays broadcasts through the dispatcher process. This is done such that all communication are asynchronous, allowing the worker

processes to not wait for communication synchronization which is offloaded to the dispatcher. `SEND` and `RECEIVE` are one-to-one communication primitives to send and receive a message to/from a process. `WAIT` is a non-blocking wait routine. A process that is in the `WAIT` routine is not processing any refinement work, but it is polling for new messages and processing incoming messages, while it waits for the specific message.

While processes are working, they will have events that need to be communicated to other processes. These events are: (i) A branch of the tree can be pruned (Line 27); (ii) A new best candidate for canonical labelling is found (Line 39); and (iii) An automorphism is found (Line 41). These messages are sent to the dispatcher process, which broadcasts the message to all worker processes.

The parallel algorithm starts with all processes initializing their local state variables, including their own independent stacks (Lines 1-4). The worker processes wait (Line 15) for a `work-start` message from the dispatcher. The dispatcher runs through the same start as the sequential algorithm. It builds the tree and refines until it reaches the first leaf node. The information in the leaf node is needed as a starting point for the rest of the comparisons. Once the dispatcher reaches the first leaf, it broadcasts a `work-start` message to the workers (Line 33).

Each process continues as in the sequential algorithm, with a few additions to facilitate communication of important events. There is a periodic check for incoming messages (Lines 18-22). On worker processes, this is done after a number (specified by the *communication period*) of `REFINEMENT` routines have been completed. It is done after every `REFINEMENT` routine on the dispatcher. The dispatcher is tuned to focus on communication, while the workers are tuned to prioritize searching the tree.

Two communication routines are introduced here, `CHECKFORMESSAGESANDUPDATE` and `FULLFILWORKREQUESTS`. `CHECKFORMESSAGESANDUPDATE` (Line 19) is used by all processes to check the message queue and process incoming messages. This is where prune messages, new best partition, new automorphism, and `work-stop` messages are processed. `FULLFILWORKREQUESTS` (Line 21) is only run by the dispatcher. It is where any requests for more work are processed. After a process runs `CHECKFORMESSAGESANDUPDATE`, the communication period counter is reset (Line 22).

The algorithm then proceeds as in the sequential version, it pops a tree node off its local stack (Line 23). It checks if the partition corresponding to this node can be pruned (Lines 24-28) and prunes as necessary. If an automorphism prune is done, the algorithm continues with the next loop iteration. If an indicator function prune is performed, then before continuing, a message is broadcast to all processes (Line 27). This is because the indicator function pruning can prune the tree at a higher level than the current node, and nodes in the tree owned by other processes may need to be pruned.

This does not happen with automorphism prunes. After this, `INDIVIDUALIZATION` and `REFINEMENT` (Lines 29, 30) are performed.

There is a special case if the refined partition is the first leaf node found by the dispatcher. This is when the dispatcher broadcasts the `work-start` message to all processes (Line 33). The rest of the discrete partition handling is the same as the serial algorithm, with the addition that a process finding a new best canonical label will broadcast the details to all processes (Line 39). Likewise, when a new automorphism is discovered, it is broadcast to all processes (Line 41). The algorithm continues as in the serial version by expanding the current node, when it is not discrete, and pushing children to the stack.

When a process runs out of work (Lines 46-53), it has to request more work. The worker process sends a `work-request` to the dispatcher (Line 52) then it waits for a response (Line 53), continuing to process messages while waiting for more work. When the dispatcher receives a work request, it sends portions of its stack to the requesting worker process. It will send back to the worker a number of nodes determined according to the work split parameter. The amount cannot exceed the work send limit.

When the dispatcher runs out of work, it `SENDS` a `work-return` request to the worker with the largest stack (Line 50). The worker processes send a status message to the dispatcher as part of `CHECKFORMESSAGESANDUPDATE` (Line 19), that reports the size of the process's stack. The dispatcher uses these status messages to keep track of how much work each worker has. If all workers are also out of work, then the dispatcher broadcasts a `work-end` message (Line 48), all processes shut down, and the dispatcher returns the permutation determined as the canonical label.

We originally used an architecture where workers asked each other for work, and there was no dispatcher. This was less stable. We switched to the dispatcher approach, and found it to be less complex. It turned out to be more stable and faster.

Our parallel implementation is written as an extension to the *Nauty* software package. The core individualization, refinement, and pruning code is unchanged. The parallel implementation replaces the recursive depth first search with a stack based approach. The output of the parallel implementation is identical to that of the sequential implementation, making it compatible with the sequential implementation.

V. EXPERIMENTAL ANALYSIS

In this section, we investigate the performance of the proposed parallel algorithm by performing an extensive set of experiments with difficult graph instances on multi-core systems with various numbers of cores.

A. Experimental Setup. Our parallel algorithm was implemented in C, and is based on *Nauty* version 2.8.9. It uses Open MPI version 4.1.1 for inter-process communication. The program was compiled using GCC version 8.5.0. Our source code is available on *GitHub* [4].

TABLE I: Benchmark graphs.

Graph	Instance	$ V $	$ E $
Miyazaki	M-100	2000	3000
	M-120	2400	3600
	M-150	3000	4500
Hadamard	had-100	400	20,200
	had-156	624	48,984
	had-172	688	59,512
Projective plane	pp-16-3	546	4641
	pp-16-19	546	4641
	pp-16-20	546	4641
Cai-Fürer-Immerman	cfi-194	1940	2910
	cfi-198	1980	2970
	cfi-200	2000	3000

The experiments were conducted on the Pittsburgh Supercomputing Center’s Bridges-2 supercomputer, regular memory nodes. Each node has two AMD EPYC 7742 processors, with 64 cores (128 threads), 256 MB L3 cache, and 256 GB RAM, running Red Hat Enterprise Linux version 8.10. The machines are connected to a 200 Gb/s Infiniband network.

The control tests were run using the original *Nauty* implementation, single process. They were run periodically, throughout the experimental test batch for each graph. In this way, the control tests provide a baseline for comparison, and also provide a check that the machines were running consistently throughout the testing.

B. Graph Instances. We evaluated most of the *Nauty* and *Traces* library of benchmark graphs [5]. Most were processed by the sequential algorithm with a running time of less than one second. The graphs we found most difficult for the sequential implementation are highly regular graphs. These graphs partition into fewer cells of larger degree, causing a deeper search tree, and require more computation. We chose the following graphs for their regularity and non-trivial sequential running time.

(1) *Miyazaki Graphs.* An $M-k$ graph is a 3-regular construction with $20 \cdot k$ vertices and $30 \cdot k$ edges, proposed by Miyazaki [6]. The *Miyazaki* graphs used for testing were generated with a custom extension to the *NetworkX* package [9], written in Python 3. Available on *GitHub* [4].

(2) *Hadamard graphs.* An n -Hadamard graph [10] is a graph with $4n$ vertices defined in terms of a Hadamard matrix $\mathcal{H}_n = (h)_{ij}$. The graph is composed of four types of vertices, r_i^+ , r_i^- , c_i^+ , and c_i^- , where r stands for “row” and c stands for “columns”. There is an edge $(r_i^\pm, c_j^\pm)$ for each matrix element such that $h_{ij} = 1$ and an edge $(r_i^\pm, c_j^\mp)$ for each matrix element such that $h_{ij} = -1$.

(3) *Projective plane graphs.* A projective plane graph is a bipartite incidence graph representing a finite projective plane. It consists of nodes representing points and lines, where an edge exists if a point lies on a line [11]. For example in the case of projective planes of order 16 (pp-16 set of graphs in our benchmark), the incidence graph is a 17-regular bipartite graph with 546 vertices and girth 6.

(4) *Cai-Fürer-Immerman graphs* [12]. These graphs form a

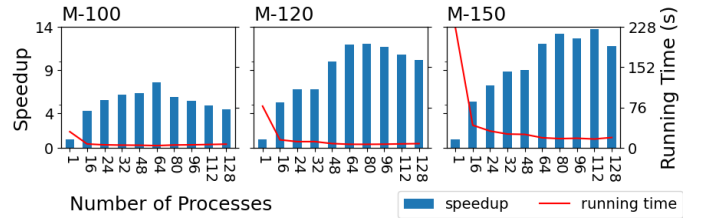


Fig. 3: Miyazaki graphs: Speedup and running time vs. Number of processes.

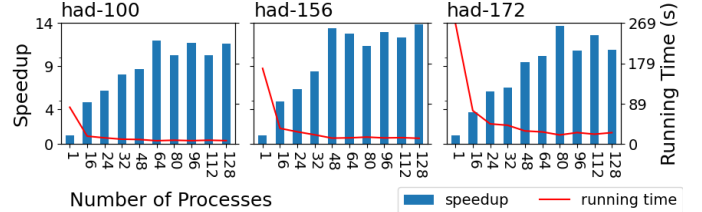


Fig. 4: Hadamard graphs: Speedup and running time vs. Number of processes.

class of pairs of non-isomorphic graphs of bounded color-class that force to have $k = \Omega(n)$ in order for the k -dimensional Weisfeiler-Leman algorithm [13] to distinguish them.

C. Experimental Results. The running time of the serial algorithm, T_s , for each graph is the average of all control tests run for that graph. The running time of the parallel algorithm, T_p , is the average running time of ten runs on p processes for each graph. The speedup $S = T_s/T_p$ is calculated using these averages.

(1) *Miyazaki graphs* (Fig. 3). The speedup obtained by our algorithm increases with the number of processes, to a point, then it decreases. The efficiency of the algorithm improves with larger graph sizes, as does the speedup. On M-150 it achieves a maximum speedup of 13.76 on 112 processes, with a running time reduction from 227.15s for one process to 16.50s.

(2) *Hadamard graphs* (Fig. 4). Our algorithm shows a consistent speedup across the three selected benchmark graphs. In the case of the had-172 graph, the algorithm obtains a maximum speedup of 13.68, with a running time reduction from 268.2s for one process to 19.6s. The peak speedup of the had-172 graph is achieved at 80 processes, after which the speedup declines. The other two graphs had better speedup, though less efficiency, with more processes.

(3) *Projective planes graphs* (Fig. 5). Our algorithm exhibits a high variance in running time between the sample graphs, given they are all the same size, though the speedup is similar. pp-16-3, which had the shortest running time, had a slight decrease of the speedup with higher processor count. pp-16-19 saw increasing speedup, up to the maximum tested 128 processes. This is the only graph tested that had a monotonically increasing speedup over the entire process test range.

(4) *Cai-Fürer-Immerman (CFI) graphs* (Fig. 6). Our algorithm increases speedup to about 14 on 64 processes, then

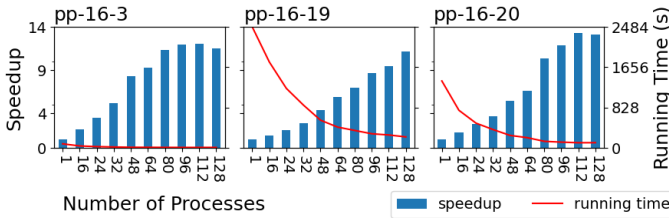


Fig. 5: Projective Plane graphs: Speedup and running time vs. Number of processes.

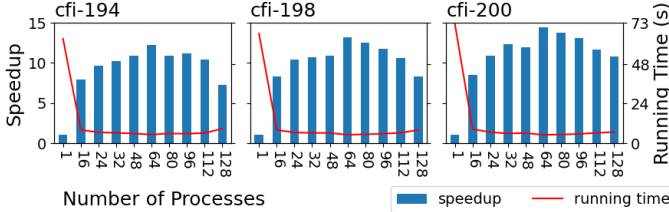


Fig. 6: Cai-Fürer-Immerman graphs: Speedup and running time vs. Number of processes.

decreases beyond 64. We observe an increase in speedup with the increasing graph size, to a maximum 12.19, 13.16, 14.44 respectively.

Our algorithm does not perform similarly across all graph classes. The Miyazaki and CFI graphs show a well defined concave downward speedup curve, Hadamard less well defined, indicating that there is a scalability bottleneck for the algorithm with these graphs. The reason is that these are small graphs with insufficient work available for the parallel execution. We see a shift of the curve to the right (more processes) as the size of the graphs increase, indicating there is potential to improve efficiency with more processes on larger graphs of these classes.

The projective plane graphs are different from the other classes, as all three shown are the same size (number of vertices and edges). The projective plane graphs have a significantly increasing average amount of work (in terms of number of refinements) done. Each graph shows an order of magnitude increase in the number of refinements from the previous: pp-16-3 (1,555,370); pp-16-20 (46,077,389); pp-16-19 (108,664,902). As the average amount of work increases, the scalability improves.

We believe the bottleneck to be the communication costs. As the overall amount of work increases, less communication is needed as processes are more self sustained, allowing for more efficient work with more processes. The projective plane graphs exhibit by far the largest average amount of work done, with the Hadamard graphs in the range of single millions of refinements, Miyazaki graphs in the high tens of thousands, and CFI graphs just over ten thousand.

Given that the sequential baseline algorithm, *Nauty*, is a highly optimized solver, we believe that achieving a speedup close to 14 (e.g., on 48 cores for had-156) is still a notable improvement in performance, especially for difficult graph instances such as those considered in our experiments.

VI. RELATED WORK

There are several software packages that implement individualization-refinement. *Nauty* [7] is widely considered to be the standard implementation of the individualization-refinement algorithm. *Saucy* [14] was developed to deal with large sparse graphs that were generated from formulas represented in conjunctive normal form (CNF) [15]. It improves performance by exploiting the sparsity of the graph, but does not provide a canonical labelling, only generators for the automorphism group. *Bliss* [16] improves performance for large sparse graphs, by altering the tree computations, and improving pruning heuristics. *Traces* [7] prefers larger target cells and a breadth first traversal of the search tree. *Conauto* [17] does not provide a canonical labelling of the graph, instead directly compares two graphs for isomorphism, looking for similar sequences of vertex partitions. *Dejavu* [18] directly compares two graphs using a probabilistic approach, repeatedly probing random walks of the individualization-refinement search trees.

The *Weisfeiler-Leman* algorithm, referred to as k -WL labels k -tuples of vertices, and iteratively refines these labels. The final labelling, can be used to compare two graphs for isomorphism. k -WL can distinguish two graphs as not isomorphic, but failure to distinguish does not guarantee an isomorphism exists. [19].

Babai [2] gives us the best theoretical time complexity for graph isomorphism. His algorithm is quasipolynomial, that is, $\exp((\log n)^{O(1)})$. The algorithm is complex, relying on group theory and combinatorics. Pham [20] et al. describe a parallel version of Babai’s algorithm. Their algorithm is described as running in polynomial time, using a quasipolynomial number of processors.

The closest work to ours is by Tener [21] who described an approach to parallelizing an individualization-refinement algorithm in his Ph.D. dissertation. His approach, called *Nishe*, is similar to the one described here, and was an inspiration for this work. However, we were unable to find any published source code or software package that implements *Nishe*. We are not aware of any other parallel algorithm or implementation of canonical labelling algorithms that is available in the literature. Thus, our work fills the existing gap in the area of parallel algorithms and software for canonical labelling of graphs.

VII. CONCLUSION

Our work demonstrates that a parallel implementation of an individualization-refinement algorithm can achieve a notable speedup over a highly optimized sequential baseline solver (*Nauty*) on difficult graph instances such as highly regular graphs. In our future work, we plan to consider other algorithms that implement the individualization-refinement paradigm, especially those that focus on large, sparse graphs. There are several available that solve the graph isomorphism problem by direct comparison instead of computing a canonical labelling. These algorithms are interesting for parallelization.

ACKNOWLEDGEMENT

This work used the Bridges-2 nodes at the Pittsburgh Supercomputing Center through allocation CIS230399 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

REFERENCES

- [1] D. B. West, *Introduction to graph theory*, 2nd ed. Upper Saddle River, NJ: Pearson, 2000.
- [2] L. Babai, “Graph isomorphism in quasipolynomial time,” *arXiv preprint arXiv:1512.03547*, Jan. 2016.
- [3] B. D. McKay, “Practical graph isomorphism,” *Congressus Numerantium*, vol. 30, pp. 45–87, Jan. 1981.
- [4] J. Haslett and D. Grosu, “Parallel canonical labelling for graph isomorphism testing,” <https://github.com/DireMonarch/Parallel-Canonical-Labelling-for-Graph-Isomorphism-Testing>, 2026, [Online; accessed 20-February-2026].
- [5] B. McKay and A. Piperno, “Libraries of benchmark graphs,” <https://pallini.di.uniroma1.it/Graphs.html>, 2026, [Online; accessed 20-February-2026].
- [6] T. Miyazaki, “The complexity of mckay’s canonical labeling algorithm,” *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 28, pp. 239–256, Jun. 1997.
- [7] B. D. McKay and A. Piperno, “Practical graph isomorphism, ii,” *Journal of Symbolic Computation*, vol. 60, pp. 94–112, Jan. 2014.
- [8] R. Parris and R. R. C., “A coding procedure for graphs,” Univ. of West Indies Computer Centre, Tech. Rep. UWI/CC 10, 1969.
- [9] N. developers, “Networkx,” <https://networkx.org>, 2024, [Online; accessed 01-June-2024].
- [10] A. E. Brouwer, A. M. Cohen, and A. Neumaier, *Distance Regular Graphs*. New York: Springer-Verlag, 1989, pp. 19–20.
- [11] D. R. Hughes and F. C. Piper, *Projective Planes*. New York: Springer-Verlag, 1973.
- [12] Cai, JY. and Fürer, M. and Immerman, N., “A reduction of a graph to a canonical form and an algebra arising during this reduction,” *Combinatorica*, vol. 12, p. 389–410, 1992.
- [13] B. Y. Weisfeiler and A. A. Leman, “A reduction of a graph to a canonical form and an algebra arising during this reduction,” *Nauchno-Tekhnicheskaya Informatsia*, vol. 2, no. 4, pp. 12–16, 1968.
- [14] S. developers, “Saucy,” <http://vlsicad.eecs.umich.edu/BK/SAUCY/>, 2024, [Online; accessed 01-June-2024].
- [15] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, “Exploiting structure in symmetry detection for cnf,” in *Proc. of the 41st Design Automation Conference*, Jun. 2004, pp. 530–534.
- [16] T. Junttila and P. Kaski, “Engineering an efficient canonical labeling tool for large and sparse graphs,” in *Proc. of the 9th Workshop on Algorithm Engineering and Experiments*, Jan. 2007, pp. 135–149.
- [17] J. Lopez-Presa, A. Anta, and L. Chiroque, “Conauto-2.0: Fast isomorphism testing and automorphism group computation,” *arXiv preprint arXiv:1108.1060*, Aug. 2011.
- [18] M. Anders and P. Schweitzer, “Engineering a fast probabilistic isomorphism test,” in *Proc. of the Symposium on Algorithm Engineering and Experiments (ALENEX’21)*. SIAM, 2021, pp. 73–84.
- [19] S. Kiefer, “The weisfeiler-leman algorithm: an exploration of its power,” *ACM SIGLOG News*, vol. 7, no. 3, pp. 5–27, Nov. 2020.
- [20] D. Pham, K. V. Palem, and M. Rao, “A polynomial time parallel algorithm for graph isomorphism using a quasipolynomial number of processors,” *arXiv preprint arXiv:2002.04638*, Feb. 2020.
- [21] G. Tener, “Attacks on difficult instances of graph isomorphism: Sequential and parallel algorithms,” PhD dissertation, University of Central Florida, Orlando, FL, USA, May 2009.