SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Graph isomorphism algorithms in nauty

av

Fredrik Stenkvist

2018 - No K27

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Graph isomorphism algorithms in nauty

Fredrik Stenkvist

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Jörgen Backelin

2018

Graph isomorphism algorithms in nauty

Fredrik Stenkvist

Contents

1 Abstract

The graph isomorphism problem, the problem of determining if two given graphs are the same up to a relabelling of the vertices, is a very important problem in graph theory. One of the best algorithms for practical use is the nauty algorithm. The subject of this paper is to show how the nauty algorithm works, discuss some of its strengths and weaknesses and show how it handles some common families of graphs.

The nauty algorithm is a backtracking algorithm which creates a canonical labelling for a graph. Then one can easily check if two graphs are isomorphic by checking if they get the same canonical labelling. It handles graphs which have vertices of many different degrees very fast, while it has more problems with graphs which are regular and which have large automorphism groups.

2 Introduction

The graph isomorphism problem or GI-problem, the problem of comparing if two graphs are isomorphic or not, is a important problem not in graph theory and in applications in many other subjects as well. One of the reasons why the GI-problem is interesting is that there are many problems, such as isomorphisms between latin square which you can reduce to the GI-problem in a fast way. [2]. The importance of this problem is such that some mathematicians have started talking about a new complexity class, called GI-complete, which are problems reducible to the GI-problem in polynomial time[7]. If we can find fast algorithms to solve the GI-problem we will be able to solve many other problems faster too.

The best upper bound found so far is given by László Babai. He has shown that the GI-problem can be solved in quasi polynomial time ($\exp \log n^{\mathcal{O}(1)}$)[1], though he does not contribute in the way of practical solutions in his paper. For practical use there are a couple of algorithms such as, traces and nauty. One part of nauty or, *no automorphisms , yes?* is an algorithm created by Brendan McKay which solves the GI-problem. In this paper I will explain how the nauty algorithm works and explain some of its strengths and weaknesses and show how it handles some families of graphs.

3 Background

3.1 Graph theory

3.1.1 Graphs

Denote V as the set $\{1, 2, 3, \dots, n\}$. Then we can define $G(V)$ as all simple connected labelled graphs with the vertex set V. The algorithm works for unlabelled graphs too, then we give the vertices a random labelling. The edge set $E(G_i)$ of a graph, will contain pairs v_i and v_j, where v_i and v_j are in V and $v_i \neq v_j$, such that if $\{v_i, v_j\}$ is an element of the edge set $E(G_i)$ then v_i and v_j are neighbours in G_i. The algorithm works for a wider assortment of graphs like digraphs and hypergraphs but I will not study those. The *adjacency matrix* A of G is a $n \cdot n$ matrix where $A_{ij} = 1$ if $\{v_i, v_j\}$ is in $E(G)$ and otherwise $A_{ij} = 0$

3.1.2 Partitions

An *unordered partition* of V is a set $\pi = \{V_1| \cdots |V_k\}$ where the cells V_i of π are disjoint, non-empty subsets of V and whose union is V. An *ordered partition* is a sequence $(V_1|V_2| \dots |V_k)$ which cells, $\{V_1|V_2| \dots |V_k\}$, creates an unordered partition of V. The set of all unordered partitions will be denoted $\Pi(V)$, the set of all ordered partitions will be denoted $\Pi_*(V)$ and the union of the two will be denoted $\Pi(V)^*$.

A cell with one element will be called a *fixed cell* of a partition. If every cell in a partition are fixed by π, π is called a *discrete partition.* If π has only one cell it is the *unit partition.* If we have two partitions π_1 and π_2 we say that π_1 is *finer* than π_2 if every cell of the partition π_1 is a subset of one of the cells in the partition π_2, which is denoted $\pi_1 \leq \pi_2$. In that case we will also say that π_2 is *coarser* than π_1. Noteworthy is that π is both coarser and finer than itself. The *length* of a partition π is the number of cells it has and is denoted $|\pi|$

Let $G \in G(V)$, $v \in V$ and $W \subseteq V$, we define $D_G(W, v)$ as the number of of elements in W that are neighbours to v in the graph G. A graph is *regular* if for all $v \in V$, $D_G(V, v)$ have the same value. We will denote a partition $\pi = (V_1|\cdots|V_n) \in \Pi(V)_*$ as *equitable* if for every elements v_1 and v_2 in the same cell V_j of π, $D(V_i, v_1) = D(V_i, v_2)$ for every i in $1, 2 \cdots |\pi|$.

Given a partition $\pi \in \Pi(V)^*$ we define $\text{fix}(\pi)$ as the set of elements which are in a cell fixed by π. We will also define $\text{mcr}(\pi)$(where *mcr* stands for minimum cell representation) as the set of elements which are the smallest in each cell of π. An example of this is that for the partition $\pi = (123|45|6)$ the minimum cell representation will be $\text{mcr}(\pi) = \{1, 4, 6\}$ and $\text{fix}(\pi) = \{6\}$. Now we can define the minimum cell representation and fix set for a permutation $\delta \in S(n)$ (the symmetric group on n elements). The minimum cell representation will be $\text{mcr}(\pi_1)$ where each cell of the partition π_1 is an orbit of δ. Similarly $\text{fix}(\delta)$ will be $\text{fix}(\pi_1)$.

3.1.3 Groups

Let $\gamma \in S(n)$ and $v \in V$, then the image of v under γ will be denoted as v^γ. Then γ acts on V_i where $V_i \supseteq V$ as if γ acted on every element of V_i, which will be denoted as V_i^γ. Then γ acts on a partition π as if γ acted on every cell of π, which will be denoted π^γ. If γ acts on a graph G, it leaves the vertex set intact but it acts on the edge set $E(G^\gamma) = \{\{v_i^\gamma, v_j^\gamma\}\}, \forall\{v_i, v_j\} \in E(G)$. For multiplication of permutations I will not use the standard notation. Typically when working with permutations $\delta_i \cdot \delta_j$ means that we first apply δ_j and then δ_i instead I will keep with the notation Brendan McKay uses where it means that we first apply δ_i and then δ_j. A permutation which has a fixed set which fixes all but two elements in it is called a *transposition.*

A partition is not an permutation but I will at times use it as one. If I say that we apply π or a subset of cells of π to for example a graph G, what I mean is that we apply $\delta \in S(n)$ where δ consists of each cell of π or the specified subset of cells as a permutation (with the rest of the elements in V mapped to their selves). The permutations will map the lowest element in the cells to the next lowest, the next lowest to the third lowest and so on until the highest element which will e mapped to the lowest. For example applying $\pi = (12|3\cdots n)$ would be to apply the permutation $\delta = (12)(3\cdots n)$ whilst just applying V_2 of π would give the permutation $(1)(2)(3\cdots n)$.

We say that a permutation δ stabilizes a partition $\pi \in \Pi^*$ with cells $(V_1, V_2 \ldots V_n)$ if for every cell V_i in π $V_i^\delta = V_i$. If a permutation or set of permutation stabilizes π will be denoted with a subscript π, one example is that $S(n)_\pi$ would be the subset of $S(n)$ which stabilizes π.

Given a permutation δ we define $\theta(\delta)$ to be the partition which cells are the orbits of δ. Given two partitions π_1 and π_2 we define $\pi_1 \vee \pi_2$ as if two elements are in the same cell of π_1 we merge the cells they belong to in π_2 and then we return π_2.

Then we can define $\mathrm{Aut}(G)$ as the subset of permutations such that $\delta \in \mathrm{Aut}(G)$ if

$$E(G^\delta) = E(G).$$

This subset of permutations will create a permutation group. The identity permutation obviously works as an identity automorphism. The subset will be closed, given two automorphisms δ_i and δ_j, the product $\delta_i \cdot \delta_j$ will also be an automorphism. Let G be any graph and δ_i and δ_j be automorphisms on the graph. Then $E(G)^{\delta i \cdot \delta j} = E(G^{\delta_i})^{\delta_j}$ according to associativity of permutations and $E(G^{\delta_i})^{\delta_j} = E(G)^{\delta_j} = E_G$ since both δ_i and δ_j are automorphisms on G and hence the product also is an automorphism. The inverse of any automorphism δ_j will also be an automorphism since $\delta_j^{-1} = \delta_j^n$ for some n and the subset of automorphisms is closed under multiplication.

$\theta(Aut(G))$ will be the partition $\pi = \{V_1 \cdots V_k\} \in \Pi(V)^*$ such that two elements are in the same cell of $\theta(\mathrm{Aut}(G))$ if and only if they are in the same orbit of $\mathrm{Aut}(G)$

3.2 Time complexity

Time complexity measures the running time of an algorithm for an input of a certain size. Since running time differs for different inputs of the same size n one generally takes the worst case scenario. For time complexity there are different classes. Where some of the most common ones are polynomial time, sub-exponential time, and exponential time, where polynomial time is the fastest with a running time of $\mathcal{O}(\mathrm{poly}(n))$(where poly stands for any real polynomial for the variable n) and exponential time is the slowest with a running time of $(2^{\mathrm{poly}(n)})$. Sub exponential comes in-between the two with a running time of $\mathcal{O}(2^{n\epsilon})$ for all constant $\epsilon > 0$. Quasi polynomial algorithms are slower than polynomial but faster than sub-exponential time algorithms, and have an running time of $\mathcal{O}(2^{\mathrm{poly}(\log((n))})$. One of the slowest classes is factorial algorithms which have a running time of $\mathcal{O}(n!)$

3.3 The graph isomorphism problem

The graph isomorphism problem is the problem of deciding whether the given graphs G_1 and G_2 are the same up to a relabelling of the vertices or in other word if there are a permutation $\gamma \in S(n)$ such that

$$G_1^\gamma = G_2.$$

One could do this by going through all permutations $\delta \in S(n)$ and calculate if $G_1^\delta = G_2$ by checking their adjacency matrices. Though with this approach the problem would be quite slow since there are $n!$ permutations in $S(n)$.

4 The algorithm

4.1 What does the algorithm do?

Instead of looking at each permutation, the nauty algorithm creates canonical labels. A canonical labelling algorithm is an algorithm that creates a relabelling of a graph G such that each isomorphism of G will get the same relabelling. If we do this for both graphs its easy to see if they are isomorphic since the relabelled graphs then will have the same adjacency matrix.

4.2 Details of different parts of the algorithm

4.2.1 Refinement

We define a refinement function R which takes a graph $G \in G(V)$, a partition $\pi \in \Pi_*(V)$, and a sequence $\alpha = (W_1, \cdots, W_M)$ which is a sequence of distinct cells of π, and which returns a sequence $\hat{\pi}$.

```
   $\hat{\pi} = \pi$
   $m = 1$
   $p = |\pi|$
While ($m \leq M$ and $\hat{\pi}$ is not discrete)
    $W = W_m$
   $m = m + 1$
   $k = 1$
   While ($k \leq p$)
      Suppose $\hat{\pi} = (V_1, V_2, \ldots, V_p)$ at this point.
      Define a partition $\pi_l = (X_1, X_2, \ldots, X_q) \in \Pi_*(V_k)$ such that for any two
       vertices $v_i \in X_j$ and $v_r \in X_o$ $D(W, v_i) > D(W, v_r)$ if and only if
      $j > o$.
      If $q = 1$
         $k = k + 1$
      else
         t=the smallest integer such that $|X_t|$ attains the maximum value for
         all cells in $\pi_l$
         If $W_j = V_k$ for some $j$  $(m \leq j \leq M)$
            $W_j = X_t$
         For $i$  $(1 \leq i < t)$
            $W_{M+i} = X_i$
         For $(t < i \leq q)$
            $W_{M+i-1} = X_i$
         $M = M + q - 1$
          replace $V_k$ with $X_1 \cdots X_q$ in that order in $\hat{\pi}$
         $p = p + q - 1$
         $k = k + 1$
return $\hat{\pi}$
```

The reason here why we can replace W_j with X_t if $W_j = V_t$ is that they will be equivalent. Checking whether or not elements have the same number of of neighbours in a partition and all but one cell of the partition is the same as checking whether or not they have the same number of neighbours in all cells since if they have the same number of neighbours in all cells they will have the same number of neighbours in the whole partition.

4.2.2 Search tree

Let $\pi = (V_1|V_2|\ldots|V_k) \in \Pi_*$ and let v be an element of the cell i for some i in $\{1\cdots k\}$. If the cell V_i only has one element we define $\pi \circ v$ as π and if it contains more than one element we define it as $(V_1|V_2|\ldots|v|V_i\setminus v|\ldots|V_k)$. Then we define $\pi \perp v$ as $R(G, \pi \circ v, v)$, in other words we refine $\pi \circ v$ with regards to v. If a set only contains one element I will represent it by that element, for example when I write $R(G, \pi \circ v, v)$what I mean is $R(G, \pi \circ v, \{v\})$.

Let Vp be the ordered unit partition, given a $G \in G(V)$, a partition $\pi \in \Pi^*$ and a sequence of elements $l = v_1, v_2, \ldots, v_n$ of distinct elements in V, we define the partition nests derived from G , π and l to be the set of partitions $\eta = [\pi_1, \pi_2, \pi_3, \ldots, \pi_n]$, where $\pi_1 = R(G, Vp, V)$ and $\pi_i = \pi_{i-1} \perp v_{i-1}$ for $i = 2 : n$. Then we define η^i to be the i first partitions in η. A permutations δ acts on η as if it acted on all partitions in η. Now we define $N(V)$ as all partition nests derived from some $G \in G(V)$ $\pi \in \Pi(V)_*$ and a sequence l of distinct elements from V.

Let $G \in G(V)$ and $\pi \in \Pi(V)_*$. I will now describe how we build the *search tree*. All the elements of the search tree will be partition nest and will be called *nodes*. When we build the search tree we do it in a depth first approach. We start at $R(G, Vp, V)$ and work our way down to a node as far as possible and then we backtrack to the latest node were we still need to refine. When going forward we create a store ζ containing of the smallest non fixed cell unless the partition is discrete, then we backtrack. The elements in ζ will be the elements we want to refine against at this branch starting with the smallest, and when we refine against an element we remove it from the ζ. When we backtrack to a node we refine against the smallest element remaining in ζ for that node. If the store is empty we backtrack to the most recent ancestor to the current node. We are done if the store at the starting node is empty when we backtrack to it.

A node will be called a *terminal node* if the last generated partition in the node is discrete. A node η is *earlier* than a node β if η is generated before β in $T(G)$, β is then *later* than η The length of a node η will be denoted $|\eta|$ and will be the number of partitions the node contains. If we have a node $\eta = [\pi_1, \pi_2, \ldots, \pi_i]$ we say that η_m is an ancestor of η_{m+1} for $m = 1\cdots i-1$ and if η_m is an ancestor of η_k, η_k is a successor of η_m. Ifη is an ancestor of β η is also an ancestor of all nodes for which β is an ancestor of.

Now we can define the search tree $T(G)$ as the following algorithm:

```
step 1
    $k = 1$
    $Vp$ is the ordered unit partition
    $\pi_1 = R(G, Vp, V)$
step 2
    if $\pi_k$ is discrete
        go to step 4
    $W_k$=the first non fixed cell of $\pi_k$ of the smallest size

step 3
    if $W_k$ is empty
        go to step 4
    $v = \min(W_k)$
    $W_k = W_k \setminus v$
    $\pi_{k+1} = \pi_k \perp v$
    $k = k + 1$
A: $\eta = [\pi_1 \cdots \pi_k]$
    go to step 2

step 4
    $k = k - 1$
    if $k \geq 1$
        go to step 3
    stop
```

The nodes η created at **A** will be the nodes of the search tree (except the start node which is $[\pi_1]$) in the described depth first approach, from the earliest to the latest.

4.3 Canonical label

A canonical label is an labelling of a graph which will represent an isomorphism class for the graph. To be able to create a canonical label for a graph G with the vertex set V we need an ordering of all possible isomorphisms of G. For this we will use two functions. The first one will be $n(G)$.

Definition Let $G \in G(V)$, then we define $n(G)$ as the length n^2 binary number we get if we look at the adjacency matrix of G as a binary number in a row by row fashion.

Definition Let η be a terminal node and $G \in G(V)$. Define $G(\eta)$ as $n(G^\delta)$ where delta is the permutation mapping the elements in η to their position in η

This will clearly give the same result for a graph G and G^δ where $\delta \in \mathrm{Aut}(G)$ but different if δ is not an automorphism and hence it would be sufficient for an ordering of the isomorphism classes. In theory we could calculate all the nodes of the search tree and check which node η maximizes $G(\eta)$. This is not practical since there can be up to $n!$ labellings so we need to define the canonical label in another way.

The other function we will use is an *indicator function.*
Definition
Let $G \in G(V)$, $\pi \in \Pi_*(V)$, $\eta = [\pi_1 \cdots \pi_k] \in N(V)$, $\mu \in N(V)$, $l = |\mu|$ and $k = |\eta|$. Then $\Lambda(G, \pi)$ is the number of elements (v_i, v_j) in the edge set $E(G)$ such that v_i and v_j are in the same cell of π [3]. From this we can define $\Lambda(G, \eta)$ as

$$(\Lambda(G, \pi_1), \Lambda(G, \pi_2), \Lambda(G, \pi_3), \ldots, \Lambda(G, \pi_k)).$$

For such vectors we will use a lexicographical ordering, $\Lambda(G, \eta) > \Lambda(G, \mu)$ if at the first position i where the two vectors differ, $\Lambda(G, \eta)_i > \Lambda(G, \mu)_i$ or if $k > l$. The indicator function will be able to handle more then one node at a time which we will see later on.

Then for a given tree $T(G)$ we define $X(G)$ as the set of terminal nodes. If for any two nodes η_1 and η_2, $\eta_1 = \eta_2^\delta$ for $\delta \in \mathrm{Aut}(G)$ then we say that η_1 is equivalent to η_2 which will be denoted as $\eta_1 \sim \eta_2$. This will be an equivalence relation. A node will be called an *identity node* if there are no earlier nodes which are equivalent to it.

Now we define $\Lambda(T(G))$ as the set of elements of $X(G)$ for which $\Lambda(G, \eta)$ is maximized, then we can define the *canonical label* as follows.

$$C(G) = \max(G(\eta)) \text{ for } \eta \in \Lambda(T(G)).$$

4.4 Pruning the search tree

4.4.1 How to prune with automorphisms

One way to find the canonical labelling would be to generate all terminal nodes and check which node η that maximizes $G(\eta)$ of those which maximized$\Lambda(G, \eta)$.

This can be problematic if $|X(G)|$ is large.

Let η be a node such that $C(G) = G(\eta)$, then η is called a canonical node.
Theorem Let $G \in G(V)$, $\pi \in \Pi^*$ and $\Lambda^* = \Lambda(T(G))$. Let X^* be any subset of elements which contains those identity nodes such that $\Lambda(G, \eta) = \Lambda^*$.

Then X^* contains a canonical node.

Our goal now is to delete sub trees of $T(G)$ by using automorphisms. If we can find that all terminal nodes on a sub tree are equivalent to some which already have been generated we do not need to generate them.

For every generated non-terminal node η we will have an associated store ζ which will be the elements we will refine against from the node. When we generate the node this will be the first smallest non-fixed cell. When we backtrack to a node $\eta = [\pi_1, \pi_2, \ldots, \pi_n]$ we update the store to be

$$\zeta \cap \mathrm{mcr}(\gamma_1) \cap \mathrm{mcr}(\gamma_2) \cap \mathrm{mcr}(\gamma_3) \ldots \cap \mathrm{mcr}(\gamma_n) \tag{1}$$

for all found automorphisms which fixes all partitions in the node. Let v_1 be smaller than v_2 and let both of them be in the same orbit of $\mathrm{Aut}(G)_{\pi_1,\ldots,\pi_n}$. Then η refined against v_1 and v_2 will be equivalent and we are only interested in the earlier node, the one where we refine against v_1, so we can prune the entire branch where we would have refined against v_2.

The only other time we will update the store for any node is when we find an automorphism δ such that $\eta_1 = \eta_2^\delta$, which we will call explicit automorphisms. We then update the store ζ of the latest common ancestor of the two nodes with the new found automorphism according to 1.

Now we create an auxiliary partition θ. θ will start as the discrete partition $(1|\cdots|n)$. Every time we find an automorphism δ we update θ to be $\theta \vee \theta(\delta)$, this makes θ the orbits of the automorphisms found so far. We do this so that when we generated the whole search tree θ will be the orbits of G, which we then return.

Now when we know how to prune the tree if we have found automorphisms we need to find them. We have two ways to find automorphism in the search tree, what will be called explicit automorphisms and what will be called implicit automorphisms.

4.4.2 How to find explicit automorphisms

When we are generating the search tree $T(G)$, we remember two terminal nodes at a time. The first one is the earliest generated terminal node, this one will be remembered during the entire algorithm. The other one is the so far best guess of the canonical labelling. This can be the same as the earliest generated node but does not need to be. When we generate a terminal node equivalent to either of these two terminal nodes , it means that there exist $\delta \in \mathrm{Aut}(G)$ mapping the newly generated node to the other in the equivalence. We then add the the sequence $(\mathrm{fix}(\gamma), \mathrm{mcr}(\gamma))$ into our storage τ for automorphisms as long as its not full. We will put a limit L to how many pairs $(\mathrm{fix}(\gamma), \mathrm{mcr}(\gamma))$ we will store. This limit will not have an effect on the end result but will effect the efficiency of the algorithm since for higher values of L we need to check fewer non identity nodes since we can prune more.

4.4.3 How to find implicit automorphisms

Implicit automorphisms is an optional of the algorithm. Whereas we found explicit automorphisms by comparing terminal nodes and explicitly calculating the automorphisms, we can find the implicit automorphism in the whole tree and do not need to calculate them explicitly. In the case of implicit automorphisms we find the entire orbits at a time.

Lemma 4.1. Let $G \in G(V)$ and $\pi \in \Pi_*$ be equitable with respect to G. Let π_1 and π_2 be any two of the cells of π with $q = |\pi_1|$ and $r = |\pi_2|$.
Then we know that the number of edges between the vertices π_1 and π_2 will be

$$k \cdot \mathrm{lcm}(q, r)$$

where k is a whole number between 0 to $\frac{q \cdot r}{\mathrm{lcm}(q,r)}$.

This is because of the fact that π are equitable so the number of edges between each vertex in π_1 and the vertices in π_2 must be a multiple of q and the number of edges between each vertex in π_2 and the vertices in v_1 needs to be a multiple of r so the total number of edges must be a multiple of q and a multiple of r. If every vertex in π_1 are connected to every vertices in π_2 there are qr edges in total. $\square$

Lemma 4.2. Let $G \in G(V)$ and $\pi \in \Pi_*$ be equitable with respect to G. Let V_1 and V_2 be any two of the cells of π, with $v_1 \in V_1$ and $v_2 \in V$ such that $\{v_1, v_2\} \in E(G)$ and let $\delta \in S(n)_{V_1,V_2}$ If there are zero or qr edges between the elements of V_1 and V_2

then the edge set between V_1 and V_2 in G and G^{δ} are the same.

Since both cells are stabilized, the number of different edges between the two cells remain the same. We get this by looking at an edge between an element of each cell (v_1, v_2), since both cells were stabilized both elements remain in their respective cell. A permutation will work bijectively(one to one) so no two edges in $E(G)$ can be mapped to the same in $E(G^{\delta})$. Then each edge between the two cells in G are mapped to a different edge between the two cells in G^{δ} so the number remains the same. If there are no or all possible edges between the two in G the same amount will be there in G^δ and hence the edge set is the same.□

Lemma 4.3. Let $G \in G(V)$, $\delta \in S(n)_{V_1,V_2}$ and $\pi \in \Pi_*$ be equitable with respect to G. Let V_1 and V_2 be any two of the cells of π with $q = |V_1|$ and $r = |V_2|$. Let $\delta \in S(n)_{V_1,V_2}$ and $GCD(q,r) = 1$.

Then the edge set between the elements of V_1 and V_2 is the same in G and G^δ.

If $GCD(q,r) = 1$, we know that $\mathrm{lcm}(q,r) = qr$, which means that either there are no edges between π_1 and π_2 or all possible edges exists according to lemma 4.1. In either case the edge set will remain the same according to lemma 4.2. □

The following is a formulation and a proof of lemma 2.25 from practical graph isomorphisms[5]. Brendan McKay left this lemma without a proof and since I did not find the lemma obvious I decided to prove it.

Lemma 4.4. Let $G \in G(V)$ and $\pi \in \Pi_*$ be equitable with respect to G. Let V_1 and V_2 be any two of the cells of π. If π has m non fixed cells, $k = |\pi|$ and one of the following conditions is met, $n \leq k+4$, $n = k+m$, or $n = k+m+1$, then

$$\pi_i = \theta(\mathrm{Aut}(G)_{\pi_i}) \text{ for any partition } \pi_i \text{ finer than } \pi$$

.

Given that an automorphism stabilizes π we know that all fixed cells are mapped to their selves, which means that they will be their own orbit in $\mathrm{Aut}(G)_\pi$. We know that elements in the same orbit have the same number of vertices in every orbit. Every cell of π will be a set of one or more orbits of $\mathrm{Aut}(G)_\pi$. You get this by using that π is equitable with the fact that the fixed cells will be their own orbit. We only split the non fixed cell if the elements in it do not have the same number of neighbours in the set $\mathrm{fix}\,\pi$ or the same number of neighbours in the rest of V and they should then not be in the same orbit. We then split the non fixed element up into set containing one or more orbits and continue to check if the elements in the same set have the same number of neighbours in the other newly created sets. If they do not have the same number of neighbours in the other sets we continue to split them up since they then can not be in the same orbit of $\mathrm{Aut}(G)_\pi$. But since elements in the same orbit will have the same number of neighbours in every other orbit or set of orbits we can not split them up in this fashion. Hence we know that element in different cells of π can not be in the same cell of $\theta(\mathrm{Aut}(G)_\pi)$.

Now we need to prove that the elements in the same part of π are in the same orbit. From the given conditions we have a couple of different partitions we need to prove it for. The last two conditions say that it holds for partitions

which have at most one cell of size three and the rest of its cells have size one or two.
From condition one we get these cases:

- One cell of size five and the rest have size one
- One cell of size four and possibly one of size two
- Up to two cells of size three
- One cell of size three and up to two cells of size two
- Up to four cells of size two

I will only show the lemma for the upper number of cells since the lower ones will be fairly similar with only fewer edges to consider. During each case I will work with a graph G and a partition π which are equitable with regards to G. These will change so they are suitable for the case I am working on.

If we have a permutation δ which is any combination of cells of π or parts of π it will stabilise π since we δ only maps the elements of each cell to an element of the same cell. If we can find permutations δ_i which stabilises π such that G and G_i^{δ} have the same edge set for all i and such that if v_i and v_j are vertices in the same cell of π for some δ_i then v_i is mapped to v_j, we know that $\pi = \theta(\mathrm{Aut}(G)_\pi)$.

I will start with the case of a partition π where π only has cells of size one or two. If we apply the whole of π as a permutation on the corresponding graph G we know that the only part of the edge set we can influence is the edges between the vertices in two different cells with size two according to lemma 4.1 and 4.3. The elements of a cell of size two cell can have zero, one or two neighbours with another cell of size two. If there are no or two neighbours between the vertices in every pair of different cells in π with the size two we will not change this part of the edge set, according to lemma 4.2, and then we know π is an automorphism on G which stabilizes π.

If there are some cells V_1, V_2., where the vertices in V_1 have one neighbour in V_2, we will change the edge set if we only apply one of them to G. But as long as we apply both we will not change the edge set. So applying π as a permutation will not change the edge set and it will stabilize π so $\pi = \theta(\mathrm{Aut}(G)_\pi)$.

Now we take the case of one cell of size three and the rest of size one or two. Let V_i be the cell of π which have the size three. If we look at the sub-graph of G without the vertices in V_i and $\pi \setminus V_i$ we know that $\pi \setminus V_i$ is an automorphism for the sub-graph which stabilizes $\pi \setminus V_i$. If we also apply V_i we know that we will not change the edge set according to lemma 4.2 and 4.3. Hence if we apply the permutation corresponding to the non fixed cells of π twice it will be an automorphism , and each vertices of V_i will be mapped to the other two in V_i so they are in the same orbit. Now we know that $\pi = \theta(\text{Aut}(G)_\pi)$.

Let π have two cells V_1, V_2 of size three and the rest have size one. If the vertices of V_1 has one or three neighbours in V_2, applying π will not change the edge set of G according to lemma 4.2 so it is an automorphism and if we apply π twice the problem is solved. Solving the problem for one or two neighbours is equivalent, since if there are two neighbours we can instead of looking at G and G^π, look at G_* and G_*^π where the edge set of G_* is the same as the edge set of G except between the vertices in V_1 and V_2 where vertices in these cells have an edge between them in G_* if there is not one between them in G. Solving the problem for G_* will be the same as for G(you can look at it as keeping the vertices of V_1 and V_2 which isn't neighbours) and is a problem with one neighbour. Now we look at the case of the vertices in V_1 having one neighbour in V_2.

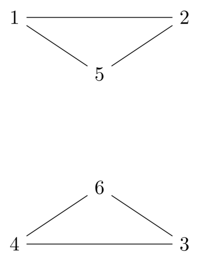
Let $V_1 = (v_i, v_j, v_k)$ and $V_2 = (v_q, v_r, v_p)$ where $(v_i, v_q), (v_j, v_r)$and (v_k, v_p) are elements of the edge set. If we apply V_1 as a permutation on G we will change the edge set between the vertices in V_1 and V_2. But if we also apply V_2 it remains the same and hence π is an automorphism. If we apply π twice the vertices in V_1 and V_2 will be mapped to each other and hence $\pi = \theta(\text{Aut}(G)_\pi)$.

Let π have one cell V_1 of size four and one cell V_2 of size two and where the rest of the cells of π has size one. As long as we stabilize π, all we need to worry about are the edges between the vertices in V_1 and those between one vertex in V_1 and one vertex in V_2 according to lemma 4.3 and 4.3. The vertices in V_1 can have zero, one, two or three neighbours in V_1. If its zero or three this part of the problem is trivial according to lemma 4.2. The problem when they have one or two neighbours are equivalent with the same reasoning as between the two cells of size three in the previous case. The vertices in V_1 can have zero, one or two neighbours in V_2 if there are zero or two, the problem is trivial according to lemma 4.2.

Now I am going to show how to solve the problem when the vertices in V_1 have one neighbour in V_1 and one in V_2 which will also will work as a solution for each other case with some slight modifications. There are two cases of edges between these cells of π. Either the neighbours in V_1 have a common neighbour in V_2 or not.

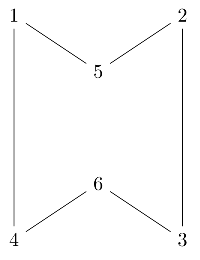
I will now show both cases, in the examples $V_1 = 1, 2, 3, 4$ and $V_2 = 5, 6$

First case:



Here both $\delta_1 = (1,2)(3,4)$ and $\delta_2 = (1,3)(2,4)(5,6)$ work as automorphisms. Then $\delta_1 * \delta_2$ is an automorphism and we have found automorphisms which stabilizes π such that vertex in any cell V_k is mapped to each vertex in V_k. Here is the other case:

Second case:



δ_1 , δ_2 and $\delta_1 * \delta_2$ works as automorphisms in this case to. Since we have solved the problem for both cases $\pi = \theta(\mathrm{Aut}(G)_\pi)$.

Now I will deal with the case a partition π with one cell V_i of size five and the rest of size one. As long as we stabilize π the only edges we need to study are those between vertices of V_i according to lemma 4.3. We know that the vertices in V_i can have zero, two or four neighbours in V_i since otherwise the induced sub-graph would have an odd number of vertices with odd degree. If they have two neighbours in V_i, the sub-graph will be C_5. For which we know that the five rotations stabilizes the edge set . If they have zero or four neighbours, we can still use the five rotations to stabilize the edge set according to lemma 4.2.

We know that in either case the five rotations work as automorphism and then the elements in the cell of size five will be in the same orbit hence $\pi = \theta(\mathrm{Aut}(G)_\pi)$.

The last two cases are included in the first two more generalized cases so they are already proved.

Now I have shown that the lemma holds for π where π fulfils any of the three conditions. But since any partition finer than π also will fulfil at least one of the conditions the lemma will hold for all those also. $\square$

4.4.4 How to prune with the indicator function

Using the indicator function as a part of the canonical labelling let us prune even more than if we would only use the automorphisms to prune. If we at any point have generated a node η such that $\Lambda(\eta)_i < \Lambda(\beta)_i$, where β is the current best guess of the canonical label, we can prune the entire branch. This is because we used a lexicographical ordering for the indicator function, which is resulting in that every terminal node on the η-branch will have a lower value for the indicator function than β and hence they are not able to be the canonical labelling.

4.4.5 The pruning part of the algorithm

Now we are ready for the last part of the algorithm, the part that handles which nodes to keep and which part of the tree to prune.

Let θ start as the ordered discrete partition, ς be the earliest terminal node, and ρ be our so far best guess of the canonical labelling, with $|\rho| = r$ and $|\varsigma| = m$. Now we define the following things,

- if π_k satisfies the conditions for π of lemma 4.4, then hh is the smallest value for which π_{hh} fulfils the requirements otherwise $hh = k$
- ht is the smallest value i such that all terminal nodes descended from ς^i have been shown to be equivalent
- the latest common ancestor of ς and η is η^h
- the latest common ancestor of ρ and η is η^{hb}
- hzb is the maximum value $1 \leq i \leq \min(k, r)$ such that $\Lambda(G, \eta^i) = \Lambda(G, \rho^i)$
- $\Lambda = \Lambda(G, \eta)$
- let v be the vertex we refine against to go from η^h to η^{h+1}

When we generate a node η, such that $\eta = [\pi_1 \ldots \pi_k]$ the following subroutine starts.

```
step 1
    If ($k > m$ or $\Lambda \neq \Lambda(G, \varsigma^k)$) and ($k > r$ or $\Lambda < \Lambda(G, \rho^k)$)
        go to B

step 2
   If $\eta$ is non terminal
          continue to generate the next node in the search tree if there are any
              if there is not any one node left to generate return $C(G) = G(\rho)$

step 3
   if($k > m$ or $\Lambda \neq \Lambda(G, \varsigma)$)
       go to step 4
   if the permutation $\gamma$ taking $\varsigma$ to $\eta$ is an automorphism
        go to A

step 4
   If($k > r$ or $\Lambda < \Lambda(G, \rho)$ or ($\Lambda = \Lambda(G, \rho)$ and $G(\eta) < G(\rho)$)
       go to B
   If($\Lambda > \Lambda(G, \rho)$ or ($\Lambda = \Lambda(G, \rho)$ and $G(\eta) > G(\rho)$)
       $\rho = \eta$ then go to B
   let $\gamma$ be the permutation taking $\rho$ to $\eta$ and go to A

A
let $\gamma$ be the found automorphism between $\eta$ and one of the two other nodes.
Add $(\text{fix}(\gamma), \text{mcr}(\gamma))$ to the storage $\tau$ and update $\zeta_{\eta_h}$ according to 1
    $\theta = theta \vee \theta(\gamma)$
   If $v$ is not in $\zeta_{\eta_h}$
       return to $\eta^h$ and then generate the next node if there are any
            if there is not any one node left to generate return $C(G) = G(\rho)$,
            $\theta(\text{Aut}(G) = \theta$
    else return to $\eta^{hb}$ and then generate the next node if there are one
        if there is not any one node left to generate return $C(G) = G(\rho)$,
        $\theta(\text{Aut}(G) = \theta$
B
  if $\Lambda \geq \Lambda(G,\rho)$ and $\eta$ is not discrete
      continue to generate the next node from $\eta$
  if $\eta$ is discrete
     If($\Lambda > \Lambda(G,\rho)$ or ($\Lambda = \Lambda(G,\rho)$ and $G(\eta) > G(\rho)$)
          $\rho = \eta$
          return to the latest $\eta^i$ for which its store $\zeta$ is not empty
              if there is not any non empty store return $C(G) = G(\rho)$,
              $\theta(\mathrm{Aut}(G) = \theta$
 If($hh < k$)
      $\theta = \theta \vee \theta(\pi_{hh})$
     add($\mathrm{fix}(\eta_{hh}), \mathrm{mcr}(\eta_{hh})$) to the storage $\tau$
 return to $\eta^i$ where $i = \min(hh-1, \max(ht-1, hzb))$ and then generate the
 next node if there are one
      if there is not any one node left to generate return $C(G) = G(\rho)$ ,
      $\theta(\mathrm{Aut}(G) = \theta$
```

4.5 Time and space complexity

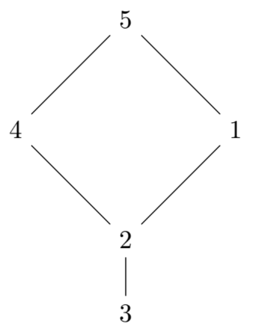
For many families of graphs the algorithm will have an polynomial running time, but there are families such that the algorithm will have an exponential running time. [6]
Let n be the number of vertices in the graph G, m the number of machine words that is required to store a vector of size n, let k be the maximum length of a node in the search tree and L the number of pairs $(\mathrm{fix}(\gamma), \mathrm{mcr}(\gamma))$ we choose as our upper limit to store in τ. Then the storage requirement for the algorithm is $2mn + 10n + m + (m + 4)k + 2mL$ machine words.

4.6 Examples

Now I will give some examples on how different parts of the algorithm works. I will do this with a couple of different graphs which are suitable for the different parts of the algorithm. I will start with an example of how the algorithm generates the starting node for the search tree.

I will do this for the following graph:

The graph G_1

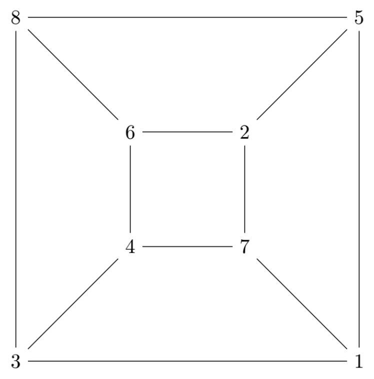


We want to calculate $R(G, Vp, V)$ which will be the start node for the search tree. We start with the partition $\pi_1 = (12345)$. First we will check if the elements have the same number of neighbours. If we do that we get that $\{2\}$ has three neighbours, $\{1, 4, 5\}$ have two neighbours and $\{3\}$ has one neighbour. We then split them up accordingly into $\pi_2 = (3|145|2)$ with the one with least neighbours first. Now we check if the vertices in each cell of π_2 has the same number of neighbours in $\{5\}$ which they does not, 1 and 4 are neighbours with 5 but 5 is not neighbour with 5. We then split them up accordingly and get the starting node to be $\pi_3 = (3|5|14|2)$. This works well to split up the vertices for graphs where the vertices has different number of neighbours, but for a regular graph it will return the unit partition.

Now I will give an example on how we build the search tree. When we build the search tree we do it in a depth first approach. We start at $R(G, Vp, V)$ and work our way down to a node as far as possible and then we backtrack to the latest node were we still need to refine. When going forward into a higher level we create a store containing the smallest non fixed cell unless the partition is fixed, then we backtrack. That cell will be the elements we want to refine against at this branch in order from lowest to highest, and when we refine against an element we remove it from the store. When we backtrack to a node we refine against the smallest element remaining in the store. If the store is empty we backtrack to the most recent ancestor to the current node. We are done if the store at the starting node is empty when we backtrack to it.

For the following graph:

The graph G_2



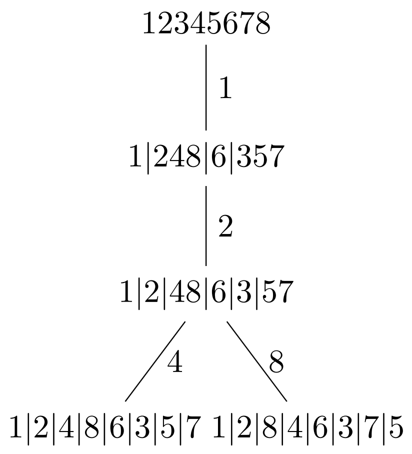
The start node of the search tree will be the unit partition since the graph is regular. We will have a store consisting of all vertices. We start with refining against 1 and working our way down to a discrete partition. Here we have reached a discrete partition.

The first terminal node of G_2

$$\begin{tikzcd}
12345678 \arrow[d, dash, "1"] \\
1|248|6|357 \arrow[d, dash, "2"] \\
1|2|48|6|3|57 \arrow[d, dash, "4"] \\
1|2|4|8|6|3|5|7
\end{tikzcd}$$

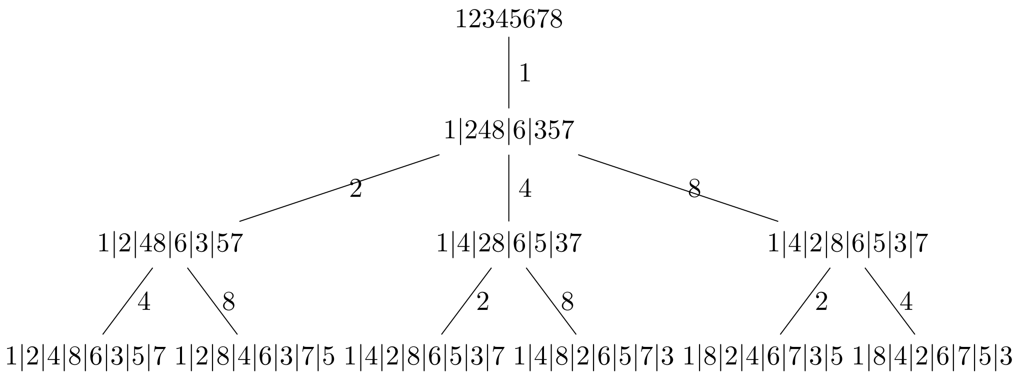
We backtrack and refine against 8 instead of 4.

The first two terminal nodes of G_2



Now we have refined against both elements at this level so we backtrack and calculate the entire branch were we refine against 4 and then the entire branch were we refine against 8.

The first branch of the search tree for G_2



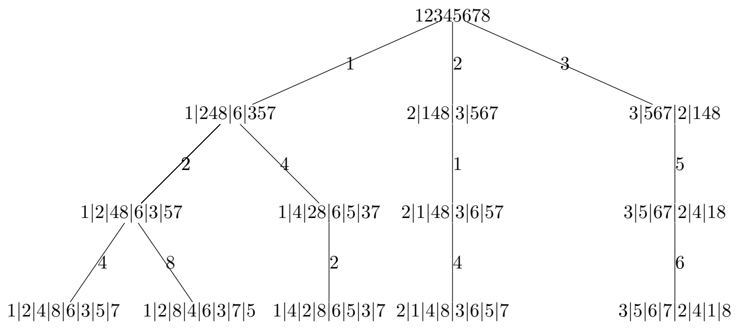
Now we have done everything for the first branch so we backtrack to the start node and we will add one branch at a time in the same way for each element in the start node in order from the smallest to the largest.

Now I will give an example of how we prune the search tree with explicit automorphisms. Whereas we before kept a store containing the smallest cell of the last partition of the current node, we also update it with the help of the automorphisms we have found so far. We will keep the store as the intersection of the smallest cell and the minimum cell representative of the automorphisms which fixes the node. We update the store at two different times. The first case is when we arrive at a node (both going forwards and backwards). The other one is when we find a automorphism between two terminal nodes. If we do that we update the store of the latest common ancestor of the two nodes which we found an automorphisms between.

When we have calculated the first two terminal nodes we can find the explicit automorphism $\delta_1 = (48)(57)$. When we then backtrack, we can prune the branch where we refine against 8 since its not a minimum cell representative of δ_1. We refine against 4 and then against 1 and find the automorphism $\delta_2 = (248)(357)$. This let us prune the branch we are on when we add it to the store of the latest common ancestor. We backtrack to the start node and refine against 2 and work our way down to the first terminal node on this branch.

Then we find the automorphism $\delta_3 = (21)(36)$. It let us prune this entire branch since 2 no longer is in the store of the start node and neither are 4 or 8. We backtrack to the start node and refine against 3. We look for the first terminal node on this branch. Then we find the automorphism $\delta_4 = (152643)(78)$. This let us prune this entire branch, and if we calculate the orbits of all the found automorphisms we get (12345678) and we can prune the rest of the search tree from the starting node.

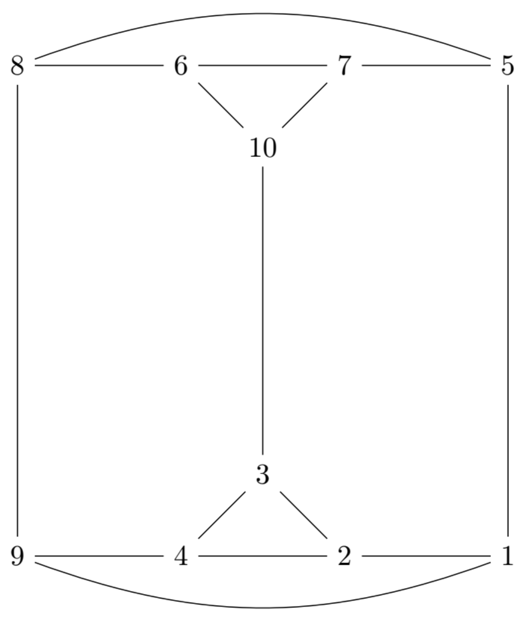
The search tree for G_2 when pruning



Now when we know how to prune the tree given automorphisms, I will show which terminal node we will keep as our best guess for the canonical labelling. As a indicator function I will use the number of edges between elements of the same cell of each partition of the node.

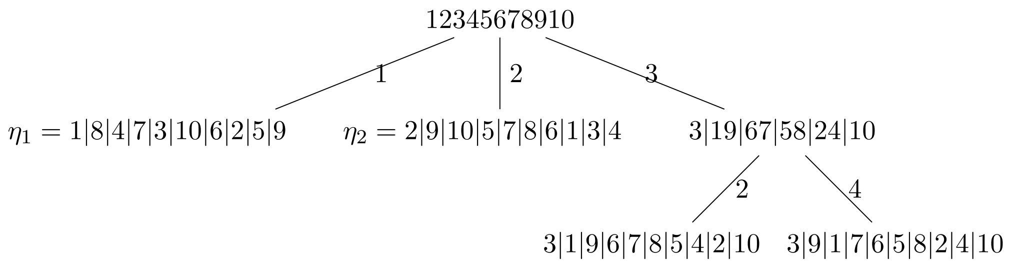
For the following graph:

The graph G_3



The starting node of the search tree will be the unit partition since the graph is three regular.

The first three branches of the search tree for G_3

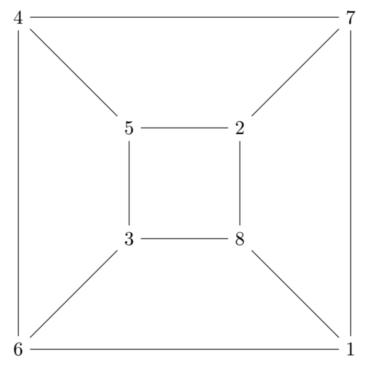


When we compare the two first generated terminal nodes η_1 and η_2, we get that they have the same value for the indicator function ((15,0)), so we need to calculate $n(G^{\delta_1})$ and $n(G^{\delta_2})$ where δ_i maps the elements of the last partition π_i in each node to their position in π_i. We then get that $n(G^{\delta_2})$ is larger and we update our best guess to π_2 but keep π_1 as our first guess.

When we generate the first node on the third branch we get that the node have indicator value (15,2) so then we know that no node on this branch can be equivalent to an earlier one and since (15,2) is lexicography before (15,0) we know that we will find a new best guess for the canonical label on this branch. Which we do when we generate the first terminal node since it have a higher indicator value then all earlier terminal nodes (15,4,0). We keep it as our best guess and keep looking through the search tree.

When we know how to handle which of the non equivalent nodes we will keep as our guess for the canonical labelling, we can handle how we find the canonical labelling. Given the graph G_2 we found that the terminal node ν which ended with the discrete partition $\pi_n = (1|2|4|8|6|3|5|7)$ maximized $C(G)$ since all terminal nodes were equivalent and this one was the earliest. Then the canonical labelling of G_2 will be G_2^{δ} where δ maps each element to their position in π_n. And then we get:

The canonical form of the graph G_2



4.7 An analysis on how the algorithm handles some families of graphs

For some of the most common graph families it is easy to analyse how the algorithm will handle them. In this part I will assume that there are room to store all found pairs $(\text{fix}(\gamma), \text{mcr}(\gamma))$ the algorithm finds. I do this to be able to show off the maximum efficiency of the algorithm .

4.7.1 How the algorithm handles K_n

How the algorithm handles the complete graph with n vertices or K_n is easily generalized for all n.

Lemma 4.5. Let G be the complete graph over n vertices, $v \in V$ and let $\pi = (V_1|\cdots|V_m)$ be in Π_*, where $v \in V_k$.

$$\text{Then } \pi \perp v = (V_1|\cdots|v|V_k/v\cdots|V_m)$$

Since G is the complete graph, every element will have the same number of neighbours with v (one) and hence we will not be able to refine more than to move v to its own cell. $\square$

This gives us a lot of information on how the search tree will look like. It will start with the unit partition, and during the refinement process we will only move any one element to its one cell which means that we will get all $n!$ possible terminal nodes in the search tree.

Lemma 4.6. Let G be the complete graph with n vertices, let $\eta = [\pi_1, \pi_2 \cdots \pi_l, \pi_m]$ where $\pi_m = (V_1 \cdots V_m)$ be any node in $T(G)$ where we are going to refine against v_j. Then the first terminal node on the branch will be $\eta_n = [\pi_1 \cdots \pi_t]$

$$\text{where } \pi_t = (V_1|V_2|\cdots|V_l|v|v_{m1}|\cdots v_{mp})$$

and $v_{m1} \cdots v_{mp}$ are the elements in V_m except v in numerical order.

Since K_n is $n-1$ regular the start node of the search tree will be the unit partition. This together with lemma 4.5 let us know that $V_1 \cdots V_l$ is fixed cells and that v is in V_m. If we now continuously use lemma 4.5 while refining against the smallest element remaining in V_m we get the wanted result. $\square$

From this we get that the first branch of the search tree for K_n looks like this:

First branch of the search tree for K_n :

$$\begin{tikzcd}
123\cdots n \arrow[d, dash, "1"] \\
1|23\cdots n \arrow[d, dash, "2"] \\
1|2|34\cdots n \arrow[d, dash, "3"] \\
\vdots \arrow[d, dash, "\text{n-1}"] \\
1|2|3|\cdots|n-1|n
\end{tikzcd}$$

Since $\text{Aut}(K_n) = S(n)$ all terminal nodes will be equivalent and the earliest terminal node will be the canonical labelling, this node will be denoted κ and will have a length of n .

If we now look at the second generated terminal node which is the first one on the branch where we refine κ^{n-1} against n instead of $n-1$ it will end with the partition $\pi_n = (1|2|3\cdots|n-2|n|n-1)$ according to lemma 4.6. We will then find the automorphism $\delta_1 = (n-1, n)$. This let us prune this entire branch since n is no longer in the minimum cell representation for $\kappa n - 1$. This will hold in general for κ^j, where $1 \leq j < n$. When we backtrack to κ^j, the only element in the store we have left to refine against will be j since we will have found automorphism $(k, k+1)$ for all $n > k > j$ and the first generated terminal node on that branch will give us the automorphism $(j, j+1)$ which will let us prune the entire branch. Then we know that the algorithm will have to generate n terminal nodes before being able to prune the rest of the search tree. This is a lot less than the in total $n!$ possible labellings but is at the same time one of the worst case scenarios for the algorithm when working with simple connected graphs since we only find automorphisms which are transpositions.

4.7.2 How the algorithm handles $K_{a,b}$

Another family for which it is easy to deduce how the algorithm handles it, is the complete bipartite graph $K_{a,b}$ where $a + b = n$. Let $A = \{v_{A1} \cdots v_{Aa}\}, B = \{v_{B1} \cdots v_{Bb}\}$ be two sets such that each part of the graph is in the same set. For $K_{a,b}$ we have two different cases

- $a = b$
- $a \geq b + 1$

The difference between the two cases is that in the first case the graph becomes a regular. This makes it so that the start node for the search tree is the unit partition instead of it being first the elements of A as a cell and then the elements of B as a cell as it would be if $a > b$. In the case of $a = b$ when we refine the starting node against an element v we will get either $v|A \setminus v|B$ or $|v|B \setminus v|A$, depending on which set v is in, since all the elements of A and B has the same neighbours. Apart from how the start node looks the algorithm handles the two cases very similarly and the efficiency will be the same.

Lemma 4.7. Let $G = K_{a,b}, v \in V$ and $\eta = [\pi_1 \cdots \pi_k] \in T(G)$ where $\pi_k = (V_1|V_2 \cdots |V_l), l \geq 2$ and $v \in V_i$ for some i between $1 \leq i \leq k$

$$\text{Then } \pi_k \perp v = (V_1|V_2 \cdots |v|V_i/v \cdots V_k)$$

Since k is larger than one we know that η is not the starting node when $a = b$. Then we know that π_k is finer than $[A|B]$. Which means that every cell of π_k either contains elements from A or B but not both. Since all the elements in the same set has the same neighbours, to refine against v will not do more then to move v to its own cell. $\square$

Let us look at the first branch of the search tree for the case $a > b$. By repeated use of lemma 4.7 we get that it look like

The first branch of $K_{a,b}$ when $a > b$

$$\begin{tikzcd}
A|B \arrow[d, dash, "v_{B1}"] \\
A|v_{B1}|B/v_{B1} \arrow[d, dash, "v_{B2}"] \\
\vdots \arrow[d, dash, "v_{b(b-1)}"] \\
A|v_{B1}|\cdots|v_{Bb} \arrow[d, dash, "v_{A1}"] \\
v_{a1}|A/v_{a1}|v_{B1}|\cdots|v_{Bb} \arrow[d, dash, "v_{A(a-1)}"] \\
v_{A1}|\cdots|v_{Aa}|v_{B1}|\cdots v_{Bb}
\end{tikzcd}$$

Let us call this node η. If we now look at the second generated terminal node we will find the automorphism $\delta = (v_{Aa}, v_{A(a-1)})$. Similarly as for K_n this result will hold in general. Let v_{Ci} (where C either stands for the set A or B which v_{Ci} is in). be the element we refined η^i against to get η^{i+1}. When we backtrack to ηi we will have found automorphisms $\delta_j = (v_{Cj}, v_{C(j+1)})$ for $i < j < |C - 1|$. Then we only want to refine against $v_{C(i+1)}$ at η^i and when we do that we find the automorphism $\delta_i = (v_{Ci}, v_{C(i+1)})$ which let us prune the entire branch. In total we need to generate n labellings which is a lot less than the $a!b!$ possible ones. If $a = b$ the first terminal node might look different if $1 \in B$. Then we will start with refining against the elements of B first instead of those in A. Though this have no effect on the algorithm since we will find automorphisms mapping each element not in C to each other before returning to the start node whilst all later nodes will be handled the same. Then we only have to refine against one element not in C, which i will call v_{D1}. When we generate the first terminal node on this branch we will find the automorphism $\delta_i = (V_{A1}, V_{B1}) \cdots (V_{Aa}, V_{Bb})$ which let us prune the entire branch we are on. This case is therefore as efficient as $a > b$ since it also needs to generate n terminal nodes.

Its easier to analyse these families of graphs then for example C_n the family of cyclic graphs. This is because of the fact that these families have many edges. They are then less dependant on the initial labelling. For both of these families of graphs the search tree and the efficiency of the algorithm does not depend on the initial labelling whilst for other families of graphs it does.

5 Discussion

The algorithm handles graphs with vertices of many different degrees fairly quickly. This is because the partition which will be the start node for the search tree will be finer than $\pi = (\forall v \in V : D(V,v) = 1 | \forall v \in V : D(V,v) = 2 \cdots | \forall v \in V : D(V,v) = n-1)$. This will make it so that we have few options to refine against at each node in the search tree when compared to for example K_n. Then the search tree will be small and we will get fewer terminal nodes to handle.

The problem for the algorithm comes when we have graphs with a large automorphism group. Then the starting node will be quite coarse and then the search tree will grow big. When the search tree grows big the algorithm have to handle many terminal nodes, the amount will be a multiple of the size of the automorphism group, and either prune them or determine if they are a better guess for the canonical labelling.

The other problem we get for a large automorphism group is that we can reach the limit of pairs of $(\mathrm{fix}(\gamma), \mathrm{mcr}(\gamma))$ which we can store. This forces us to either make the number of pairs $(\mathrm{fix}(\gamma), \mathrm{mcr}(\gamma))$ we can store higher, or we will not be able to use all found automorphisms. This will effect the efficiency of the algorithm since we will not be able to prune as much of the search tree. Then it will have to determine if all of the otherwise pruned terminal nodes could be the canonical labelling.

To get a higher efficiency for the algorithm one can choose a different way to determine which elements to refine against in the search tree, instead of just taking the smallest non fixed cell. The most efficient choice varies a lot for different families of graphs. Other options are to look at paths of length two or adjacency triangles for each vertices and from that determine which cell to refine against. For a full list of the different choices for this which comes with nauty and when they are useful, one can read the nauty manual.[4]

6 References

Unless otherwise stated all information about nauty works comes from [5]

[1] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[2] Stephen G Hartke and AJ Radcliffe. Mckay's canonical graph labeling algorithm. *Communicating mathematics*, 479:99–111, 2009.

[3] Brendan D McKay. Computing automorphisms and canonical labellings of graphs. In *Combinatorial mathematics*, pages 223–232. Springer, 1978.

[4] Brendan D McKay. nauty user's guide (version 2.5). Technical report, Technical Report TR-CS-9002, Australian National University, 2009.

[5] Brendan D McKay et al. Practical graph isomorphism. 1981.

[6] Takunari Miyazaki. The complexity of McKay's canonical labeling algorithm. 28:239–256, 1997.

[7] Ryuhei Uehara, Seinosuke Toda, and Takayuki Nagoya. Graph isomorphism completeness for chordal bipartite graphs and strongly chordal graphs. *Discrete Applied Mathematics*, 145(3):479–482, 2005.