Chapter 4

Graph Canonicalisation

The graph canonicalisation problem is about finding a canonical order of the vertices in a given graph such that the order is the same no matter how the vertices were ordered initially. As an example, consider Figure 4.1 where a graph G has an initial ordering of the vertices specified by the shown indices. If we reorder the indices of the vertices we can obtain, for example, G_1 or G_2 instead. We have not modified the graph structure so clearly $G \cong G_1 \cong G_2$, but we have implicitly modified the representation of the graph. The globally ordered adjacency lists of the graphs are shown below each graph, and we see that G_1 and G_2 have equal adjacency lists, i.e., they are representationally equal $G_1 \overset{r}{=} G_2$. They are however representationally different from G.

We could now define a total order among adjacency lists and claim that both G_1 and G_2 are "better" than G, e.g., because vertex 1 has fewer neighbours in G_1 than in G, and write $G_1 \overset{r}{<} G$. How specifically the relation $\overset{r}{<}$ is defined is unimportant for the sake of defining a canonicalisation algorithm, but we assume that it defines a total order on the relevant class of graphs. If the graphs are labelled (e.g., molecules) we assume the $\overset{r}{<}$-relation takes the labels into account.

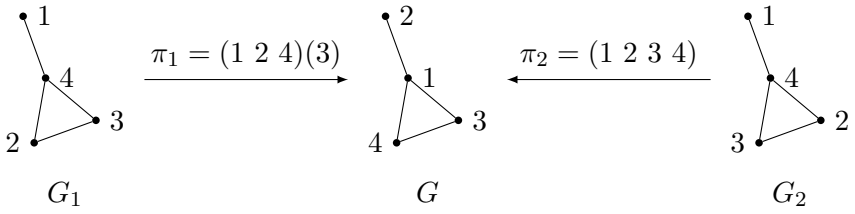


Figure 4.1: Three isomorphic graphs represented as adjacency lists. The underlying indices of the vertices are shown, and the permutations π_1 and π_2 (in cycle notation) describe the relationship between the indices of the graphs. From the adjacency lists we see that G_1 is representationally equal to G_2, and representationally different from G. The permutation $\pi' = \pi_2^{-1} \circ \pi_1 = (2\ 3)(1)(4)$ thus represents an isomorphism from G_1 to G_2.

In general for a graph G with n vertices we could theoretically enumerate all reorderings of the vertices, i.e., $n!$ graph permutations, and select any of the graphs that are smallest according to $\overset{r}{<}$ and use it as the result of canonicalisation. This is clearly a brute-force approach, which in practice is infeasible. It is currently unknown for general graph classes whether a polynomial time algorithm exists. The graph canonicalisation problem has neither been proven NP-hard in general, but if $\overset{r}{<}$ is chosen to order graphs by the lexicographical comparison of their adjacency matrices, then the problem of the finding a smallest graph is NP-hard [Babai & Luks 1983].

In this chapter we describe the individualisation-refinement paradigm, which is the basis of the currently fastest practical algorithms for graph canonicalisation, and the related problem of finding the automorphism group of a graph. The paradigm is additionally the basis of the algorithms used for canonicalisation in the chemical molecule formats SMILES and InChI (see Chapter 5). A selection of fast general algorithms are listed below, with references to descriptions of them.

- **nauty** [McKay 1981, Hartke & Radcliffe 2009, McKay & Piperno 2014b]
- **Traces** [Piperno 2008, McKay & Piperno 2014b]
- **saucy** [Darga *et al.* 2008, Katebi *et al.* 2010]
- **bliss** [Junttila & Kaski 2007]

A short overview of the core ideas is provided by [Hartke & Radcliffe 2009] and [McKay & Piperno 2014a], while the full mathematical details can be found in [McKay 1981], and parts of it in a revised version in [McKay & Piperno 2014b]. The paper [Piperno 2008] notably also includes detailed illustrated examples in its descriptions. The intention in this chapter is to give an unified intuitive understanding of the algorithmic framework, though without the proofs of correctness which can be found in [McKay 1981]. In our description we aim at precisely separating the core of the algorithm from the heuristics used for optimisation.

Throughout we denote the input graph as $G = (V, E)$ where the vertex set is the set of integers $V = \{1, 2, \ldots, n\}$. We generally use the variables u and v to refer to arbitrary vertices, but note that for ease of notation we also use $v_1, v_2, \ldots, v_k$ for an arbitrary sequence of vertices. That is, v_i is not necessarily the vertex with index i, but simply the ith vertex in the context of where it is used.

The graph may be directed or undirected, and may be labelled or not. We assume that a relation $\overset{r}{<}$ is given that induces a total order on the graph class, such that the brute-force strategy described in the beginning correctly gives a canonical form. The relation must thus include comparison of vertex and

edge labels when relevant. In Section 4.3.1 we describe how the labels can be exploited to potentially speed up a canonicalisation algorithm.

The overall idea is to start the algorithm by assuming all vertices are indistinguishable, and then iteratively use local information, e.g., vertex degrees, to partition the vertices and order them. If we reach a point where we can no longer refine the partitioning of the vertices, and some vertices are still indistinguishable, then we forcibly split a partition and again use local information for further refinement. There may be multiple choices when forcibly splitting a partition so we construct a search tree that represents all the choices.

4.1 Preliminary Definitions

4.1.1 Ordered Partitions

For representing an intermediary result of the algorithm we use an *ordered partition* of V. It is a sequence of non-empty sets $\pi = (V_1, V_2, \ldots, V_r)$ such that $V_1, V_2, \ldots, V_r$ forms a partition of V, i.e., $\bigcup_{1 \leq i \leq r} V_i = V$ and $\forall 1 \leq i < j \leq r : V_i \cap V_j = \emptyset$. Each individual V_i is called a *cell*, and cells that only contain 1 vertex are called *trivial* cells. When all cells are trivial the ordered partition is called *discrete*. In the other extreme there is an ordered partition with just a single cell equal to V. This is called the *unit partition*.

The set of all ordered partitions of V is denoted Π. If vertex $v \in V$ is in the ith cell of π we write $cell(v, \pi) = i$. We say that π_1 is *at least as fine* as π_2, and write it $\pi_1 \preceq \pi_2$ [McKay & Piperno 2014b, Section 2.1], if and only if

$$cell(u, \pi_2) < cell(v, \pi_2) \Rightarrow cell(u, \pi_1) < cell(v, \pi_1) \qquad \forall u, v \in V$$

That is, π_1 can be obtained by splitting cells of π_2 while preserving the ordering induced by π_2. When writing examples of ordered partitions we use a special notation, e.g., [1 2 | 3] which means $(\{1, 2\}, \{3\})$.

The canonicalisation algorithm starts from the unit partition and searches for a discrete partition by splitting cells when some vertices are determined to be "less" than others. A discrete partition can thus represent the canonical form in the sense that if vertex v_i is in cell i, then v_i should have index i in the canonical form. When considering a discrete partition as stored in an array, it will be a map from the canonical indices back to the original indices. For example, consider the discrete partition $\pi_1 = [2 \mid 4 \mid, 3 \mid 1]$ as an array of the vertices of the input graph G. At index 1 the vertex 2 is located, indicating it is the first vertex in the canonical form. At index 2 the vertex 4 is located, likewise indicating it is the second vertex in the canonical form. Rewriting π_1 as a permutation in cycle notation we obtain $\pi_1 = (1\ 2\ 4)(3)$. This example is depicted in Figure 4.1.

4.1.2 Permutations

In the following sections we need several concepts from computational permutation group theory, and we refer to [Seress 2003] for a comprehensive treatment of this topic.

Recall that we regard V as the set of integers $\{1, 2, \ldots, n\}$, and let γ be a permutation of V. The image of a vertex v under the permutation γ is denoted v^γ. The application of two successive permutations γ_1, γ_2 on v is written as $(v^{\gamma_1})^{\gamma_2}$ or simply $v^{\gamma_1\gamma_2}$, though note that permutation application is not commutative. The consequence of this notation is that the composition of permutations $\gamma_1\gamma_2$ is interpreted as the application first of γ_1 and then γ_2. We also use the conventional parenthesis notation with explicit composition operators, i.e., $(\gamma_2 \circ \gamma_1)(v) = \gamma_2(\gamma_1(v)) = v^{\gamma_1\gamma_2}$. The inverse of a permutation γ is written as γ^{-1}.

The set of all permutations of V, along with the permutation composition operator, is also known as the symmetric group on n elements, S_n. We say that a permutation $\gamma \in S_n$ *fixes* a vertex $v \in V$, if $v^\gamma = v$. Explicit permutations will generally be written in cycle notation, without fixed elements. For a subset $X \subseteq V$ we define the permutation of X with $\gamma \in S_n$ as $X^\gamma = \{x^\gamma \mid x \in X\}$. For a set of permutations A the group *generated* by A is the smallest group $\langle A \rangle$ that includes A. This group can be created by computing the closure of A under permutation composition. For a vertex $v \in V$ and a permutation group S' we say that the *orbit* of v under S', written $v^{S'}$, is the set of vertices which is the image of v under all permutations in S', i.e., $v^{S'} = \{v^\gamma \mid \gamma \in S'\}$.

In Figure 4.1 we saw how reordering of the vertices can be seen as a permutation of the vertex indices. We extend the notation of permutation application to graphs, such that G^γ, with $\gamma \in S_n$, denotes the permutation of the indices of the vertices in G, and assume the underlying representation to change accordingly. For Figure 4.1, if we assume that the canonicalisation algorithm described below is applied to G, then it may calculate discrete partitions corresponding to π_1 and π_2. During the algorithm it will then further calculate the graphs $G_1 = G^{\pi_1^{-1}}$ and $G_2 = G^{\pi_2^{-1}}$ as candidates for the canonical form.

Recall that a graph automorphism is an isomorphism from a graph to itself, which can be represented by a specific permutation of the indices. This is illustrated in Figure 4.1 where $G_1 \overset{r}{=} G_2$, and the permutation $\pi' = \pi_2^{-1} \circ \pi_1 = (2\ 3)$ is an automorphism on G_1. We are however interested in the corresponding automorphism on G instead. This will be explored in Section 4.3.3. The *automorphism group* of G, denoted $\mathrm{Aut}(G)$, is the subgroup of S_n with all automorphisms of G.

We also extend the application of a permutation to ordered partitions. For a permutation $\gamma \in S_n$ and and ordered partition $\pi = (V_1, V_2, \ldots, V_r)$ we define π^γ to be $(V_1^\gamma, \ldots, V_r^\gamma)$.

4.1.3 Definition of Canonicalisation

In practice we do not necessarily want to compute the actual canonical from of a graph, but rather just the needed permutation of the indices. If we let $\mathcal{G}$ denote the set of graphs, and $\mathcal{S}$ the set of all permutations, a canonicalisation function is then a function $C\colon \mathcal{G} \to \mathcal{S}$ taking a graph G as argument and returning a permutation of the vertex indices. Thus for $\sigma = C(G)$, we can obtain the canonical form as G^σ. The function must fulfil the property described in [McKay & Piperno 2014b, Property C2], that for any permutation $\gamma \in S_n$, where we compute $\sigma' = C(G^\gamma)$, it must hold that

$$G^\sigma \overset{r}{=} G^{\gamma\sigma'}$$

That is, if we canonicalise any permutation of the input graph (which is isomorphic to G) and construct the canonical form, then the two canonical forms are representationally equal. This property is a special case of what is called *isomorphism invariance* of a function. In general we say that a function f involving the vertex set V is isomorphism invariant if a permutation of the vertices results in a corresponding permutation of the output [Piperno 2008, McKay & Piperno 2014b]. This property is required for most of the procedures we use in the canonicalisation algorithm.

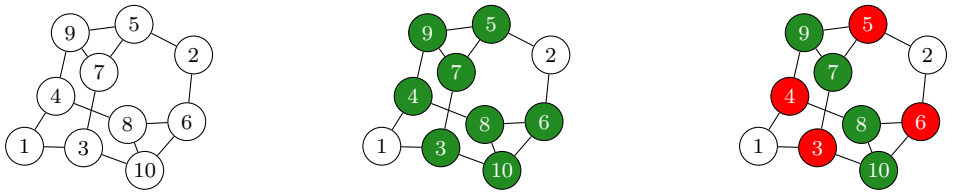
4.2 The Core Algorithm

The core components of the individualisation-refinement approach are partition refinement and vertex individualisation. They will then be used to define a search tree where the canonical form correspond to one of the "best" leaves.

4.2.1 Partition Refinement

In the beginning of the canonicalisation algorithm we assume all vertices are indistinguishable, represented by the unit partition. The idea is now to find properties of the vertices that can be used to iteratively refine the partition, for example degree information as we illustrate here. Note that the choices for refinement can be made completely independent on how the graph relation $\overset{r}{<}$ is defined.

We arbitrarily decide that lower degree is "better" than higher degree, and we can thus trivially refine the unit partition by partitioning the vertices by their degree and ordering the cells from lowest to highest vertex degree. Using the graph in Figure 4.2a as example we start with the unit partition and arrive at the ordered partition illustrated in Figure 4.2b. We can now iterate the degree argument in a more general sense: the vertices in the first cell (white) are adjacent to only some of the vertices in the second cell (green). In general for an ordered partition $\pi = (V_1, \ldots, V_r)$ we can consider two cells V_s and V_t and split V_t according to the values $d(v, V_s)$ for all $v \in V_t$. We say that we



(a) [1 2 3 4 5 6 7 8 9 10] (b) [1 2 | 3 4 5 6 7 8 9 10] (c) [1 2 | 7 8 9 10 | 3 4 5 6]

Figure 4.2: Partition refinement, starting from the unit partition (a), using ascending vertex degree as ordering. (b) the result of shattering the first (and only cell) with itself. (c) the result after shattering the second cell with the first cell of the partition in (b) . This partition is now equitable with respect to the chosen refinement procedure, as no more shattering will refine the partition. The example graph was found in [Piperno 2008].

use V_s to *shatter* V_t [Hartke & Radcliffe 2009]. In Figure 4.2a we had the special case where we used V to shatter itself. Shattering the second cell in Figure 4.2b with the first cell we arrive at Figure 4.2c. Note that in this third ordered partition we can no longer find pairs of cells where shattering results in any cell splits, i.e., $d(v, V_s) = d(u, V_s)$ for all $u, v \in V_t$ for all pairs of cells V_s, V_t. The partition is then said to be *equitable* [McKay & Piperno 2014b].

This particular refinement procedure, shown in [McKay & Piperno 2014b, Algorithm 1], is also known as 1-dimensional Weisfeiler-Lehman refinement [Piperno 2008], which can be seen as a generalisation of Hopcroft's algorithm for DFA minimisation [McKay 1981, Hopcroft 1971]. Other refinement functions can be used instead, e.g., k-dimensional Weisfeiler-Lehman refinement. In general, let $\mathcal{G}$ denote the set of all graphs, then we say that a refinement function must be an isomorphism invariant function $R\colon \mathcal{G} \times \Pi \to \Pi$ that makes the partition strictly finer, or does nothing. Formally, for all graphs $G \in \mathcal{G}$, ordered partitions $\pi \in \Pi$, and permutations $\gamma \in S_n$ we require

$$R(G, \pi) \preceq \pi$$
$$R(G^\gamma, \pi^\gamma) = R(G, \pi)^\gamma$$

This leaves a lot of freedom for defining R, and in the extreme case we find the identity function $R(G, \pi) = \pi$ as a valid choice. In the other extreme it is also valid to use another canonicalisation algorithm as a refinement function, which would guarantee a discrete partition, but does not reduce the problem at hand.

4.2.2 Vertex Individualisation and Target Cell Selection

Assume we are given an ordered partition π where the refinement function can no longer split any cells, and the partition is not discrete. The idea is now to

introduce artificial asymmetry into it by forcibly splitting the cell, and later consider all such artificial splits.

For a graph $G \in \mathcal{G}$, a non-discrete partition $\pi \in \Pi$, and a vertex $v \in V$ belonging to a non-trivial cell of π, we define a strictly finer partition $\pi \downarrow v$ by *individualising* the vertex v. Let π be on the form $(V_1, V_2, \ldots, V_{q-1}, V_q, V_{q+1}, \ldots, V_r)$ with $v \in V_q$. Then

$$\pi \downarrow v = (V_1, V_2, \ldots, V_{q-1}, \{v\}, V_q \backslash \{v\}, V_{q+1}, \ldots, V_r)$$

is the ordered partition where v has been individualised into its own cell, and that cell has arbitrarily been chosen to be smaller than the remainder of V_q. For completeness we also define individualisation for vertices in trivial cells, simply as the identity operation $\pi \downarrow v = \pi$.

After individualisation, if the partition is still not discrete, we can once again use the refinement procedure to obtain a possibly even finer partition. Note that while there is no guarantee that the refinement procedure splits any cells, the individualisation step always splits a cell, unless the partition is already discrete. As every cell must be non-empty there can be at most n cells, and therefore if we repeatedly refine and individualise we eventually obtain a discrete partition.

In the final algorithm we consider the individualisation of every vertex in a specific cell. To find that cell a function called the *target cell selector* is introduced. It is an isomorphism invariant function $Q \colon \mathcal{G} \times \Pi \to 2^V$, that given a graph G and a non-discrete ordered partition $\pi = (V_1, V_2, \ldots, V_r)$, selects a non-trivial cell V_q of π. For example, Q can simply select the first non-trivial cell of π. This is an isomorphism invariant choice as it does not depend on the location of any vertices, but only the structure of π. The target cell selector could also find the first largest non-trivial cell, or the first smallest non-trivial cell. The Traces program uses a yet more complicated target cell selector [McKay & Piperno 2014b, Section 3.2].

For ease of notation we use $\pi_{(v_1, v_2, \ldots, v_k)}$ to denote the ordered partition obtained after repeated rounds of refinement, target cell selection, and vertex individualisation. For the empty sequence we define

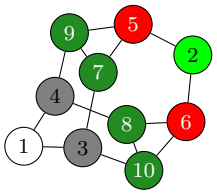
$$\pi_{()} = R(G, \pi)$$

next, if the partition is not discrete, for any $v_1 \in Q(G, \pi_{()})$ we define

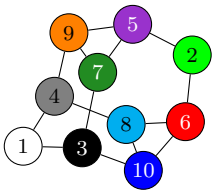
$$\pi_{(v_1)} = R(G, \pi_{()} \downarrow v_1))$$

and generally

$$\begin{aligned} v_{k+1} &\in Q(G, \pi_{(v_1, v_2, \ldots, v_k)}) \\ \pi_{(v_1, v_2, \ldots, v_k, v_{k+1})} &= R(G, \pi_{(v_1, v_2, \ldots, v_k)} \downarrow v_{k+1}) \end{aligned}$$



(a) $\pi_{(1)} = [1 \mid 2 \mid 7\ 8\ 9\ 10 \mid 5\ 6 \mid 3\ 4]$



(b) $\pi_{(1,7)} = [1 \mid 2 \mid 7 \mid 10 \mid 8 \mid 9 \mid 6 \mid 5 \mid 4 \mid 3]$

Figure 4.3: Vertex individualisation and refinement of the ordered partition from Figure 4.2c, using the target cell selector that selects the first non-trivial cell. (a) Individualisation of vertex 1, and subsequent refinement. The first non-trivial cell is $\{7, 8, 9, 10\}$. (b) Individualisation of vertex 7, and subsequent refinement. The partition is discrete and is thus a candidate for creating the canonical form.

As an example, consider the graph from Figure 4.2 and the partition $\pi_{()} = [1\ 2 \mid 7\ 8\ 9\ 10 \mid 3\ 4\ 5\ 6]$ we obtained through refinement. Using the target cell selector that returns the first non-singleton cell we have both vertex 1 and 2 for individualisation. Figure 4.3a shows the result of individualising vertex 1 and the subsequent refinement (see Figure 4.4 for the individualisation of 2). Using the same target cell selector we now have a choice between vertex 7, 8, 9, and 10. Figure 4.3b shows the result when individualising vertex 7. The obtained partition $\pi_{(1,7)}$ is discrete and is a candidate for creating the canonical form. As described in Section 4.1.2 we can interpret the partition as map from the potentially canonical graph back to the input graph. The map is a bijection so we can create the inverse map $\pi^{-1}_{(1,7)}$, which maps vertices in the input graph into the vertices of the potentially canonical graph. The candidate graph is therefore $G^{\pi^{-1}_{(1,7)}}$.

4.2.3 Canonicalisation as a Tree Search

The vertex individualisations are used to introduce artificial asymmetry when the refinement procedure can not split any more cells. However, performing an arbitrary vertex individualisation is not isomorphism invariant so in order to find the canonical form we must consider all individualisations in the chosen cell. Starting from an initial ordered partition this exploration of choices induces a tree of partitions, where the leaves correspond to discrete partitions that are all candidates for constructing the canonical form. An example is shown in Figure 4.4.

Formally, given a graph $G \in \mathcal{G}$ and an initial ordered partition $\pi \in \Pi$ we denote the search tree as $\mathcal{T}(G, \pi)$. Each node in the tree is determined uniquely by the sequence of individualised vertices [McKay & Piperno 2014b, Section 2.3], so we can simply say that the root node is the empty sequence $\tau_r = ()$, which represents the partition $\pi_{\tau_r} = \pi_{()}$. Let $\tau = (v_1, v_2, \ldots, v_k)$ be a node of $\mathcal{T}(G, \pi)$, representing the partition $\pi_\tau = \pi_{(v_1, v_2, \ldots, v_k)}$. If π_τ is discrete then τ is a leaf of $\mathcal{T}(G, \pi)$. Otherwise, let $W = Q(G, \pi_\tau)$ be the target cell of

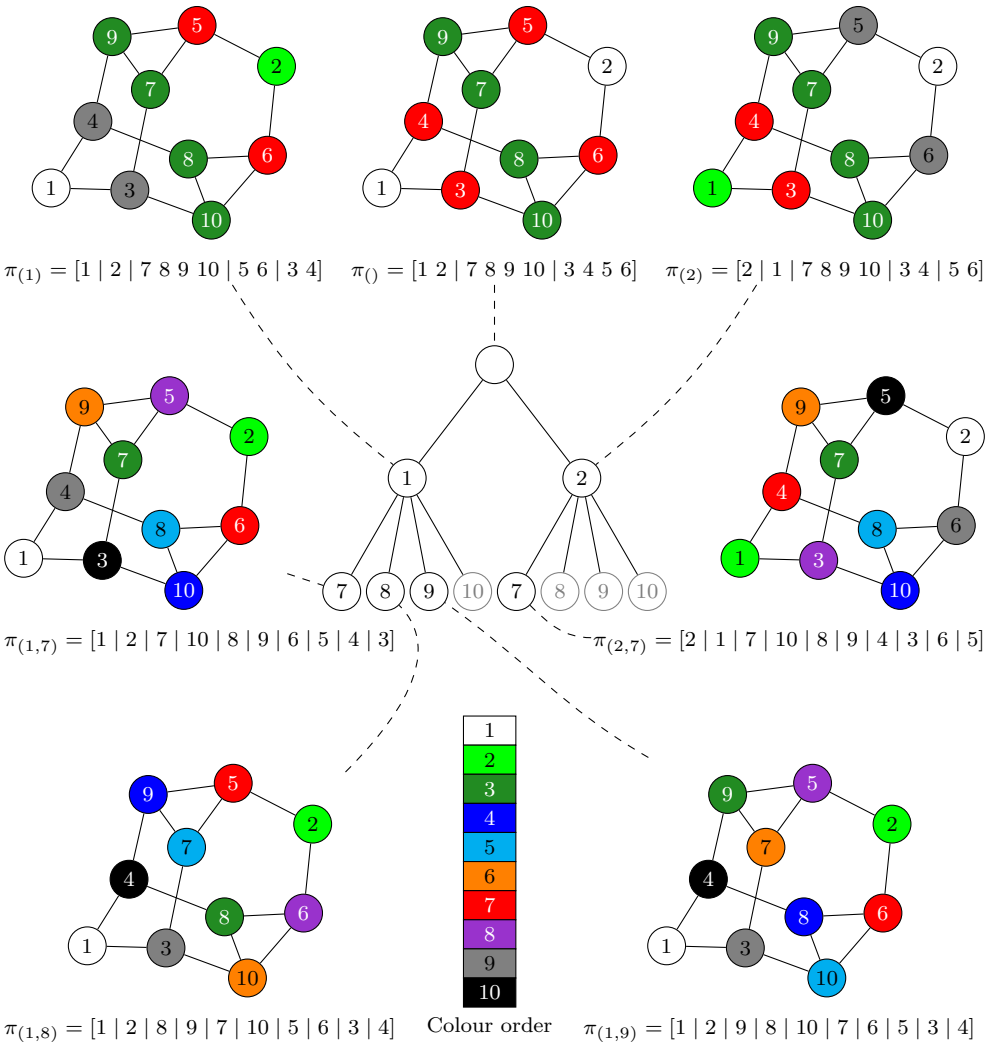


Figure 4.4: A search tree $\mathcal{T}(G, \pi)$ starting with the refinement of the unit partition in the root. Each node in the tree represents a sequence of vertex individualisations, where the latest vertex being individualised is shown in the nodes. For most tree nodes the corresponding partition is shown along with the coloured graph it represents. In the coloured graphs the vertices are labelled with the vertex indices from the input graph, and coloured with "numbers" corresponding to the potential canonical vertex indices. Note that the coloured graphs in the leaves of the left half of the tree (the children of $\tau = (1)$) are all isomorphic. This is also true among the children in the right half of the tree (the children of $\tau = (2)$. However between the halves of the tree, the graphs are not isomorphic. They greyed out nodes correspond to nodes pruned from automorphism discovery (see Section 4.3.3), when depth-first traversal of the tree is used. The example is heavily inspired by [Piperno 2008, Figure 3], though here using different functions for refinement and target cell selection.

π_τ, then the children of τ are

$$\{\tau_{(v_1,v_2,\ldots,v_k,u)} \mid u \in W\}$$

That is, τ has a child representing each choice of vertex individualisation within the selected target cell. In Figure 4.4 most of the tree for our example graph is visualised. Each node of the tree is labelled with the newly individualised vertex. In the following we as a shorthand use $\mathcal{L}(G, \pi)$ to denote the set of leaf nodes of $\mathcal{T}(G, \pi)$.

Recall that we defined a canonicalisation function as taking a graph as an argument and returning the permutation of vertex indices needed to construct the canonical form. Using the search tree defined above we can now define the basic canonicalisation algorithm in the individualisation-refinement approach: Let $R\colon \mathcal{G} \times \Pi \to \Pi$ be a fixed refinement function, and let $Q\colon \mathcal{G} \times \Pi \to 2^V$ be a fixed target cell selector. Further let $\overset{r}{<}\colon \mathcal{G} \to \mathcal{G}$ be a fixed relation that induces a total order of $\mathcal{G}$. The basic canonicalisation function $C\colon \mathcal{G} \to \mathcal{S}$ can then be calculated with the following procedure.

1. Let $G = (V, E) \in \mathcal{G}$ be the input graph to canonicalise.
2. Start from the unit partition $\pi = (V)$, and traverse the tree $\mathcal{T}(G, \pi)$.
3. Find a leaf node τ_c corresponding a $\overset{r}{<}$-smallest graph:
$$\tau_c = \underset{\tau \in \mathcal{L}(G,\pi)}{\arg\min} \left\{ G^{\pi_\tau^{-1}} \right\}$$
4. Let the result be the permutation $\pi_{\tau_c}^{-1}$ mapping vertices of G into the canonical vertices.

In a concrete implementation we naturally also need to define an algorithm to calculate the tree. For example, nauty, saucy, and bliss generates the tree in depth-first order, while Traces uses breadth-first generation combined with *experimental paths* from newly calculated nodes down to a leaf.

4.3 Algorithm Variations and Search Tree Pruning

The description above leaves room for large variations in concrete algorithms, and thereby variations in how fast those algorithms run in practice. For example, we obtain the brute-force algorithm simply by selecting the identity function as the refinement function, i.e., $R(G, \pi) = \pi$. In addition to variations of the core components there are techniques that prune branches from the search while it is being generated, thereby skipping evaluation of the refinement function, which in practice is the most time-consuming component.

The following sections define several optimisation techniques, of which some most of them change the canonical form. For a better understanding

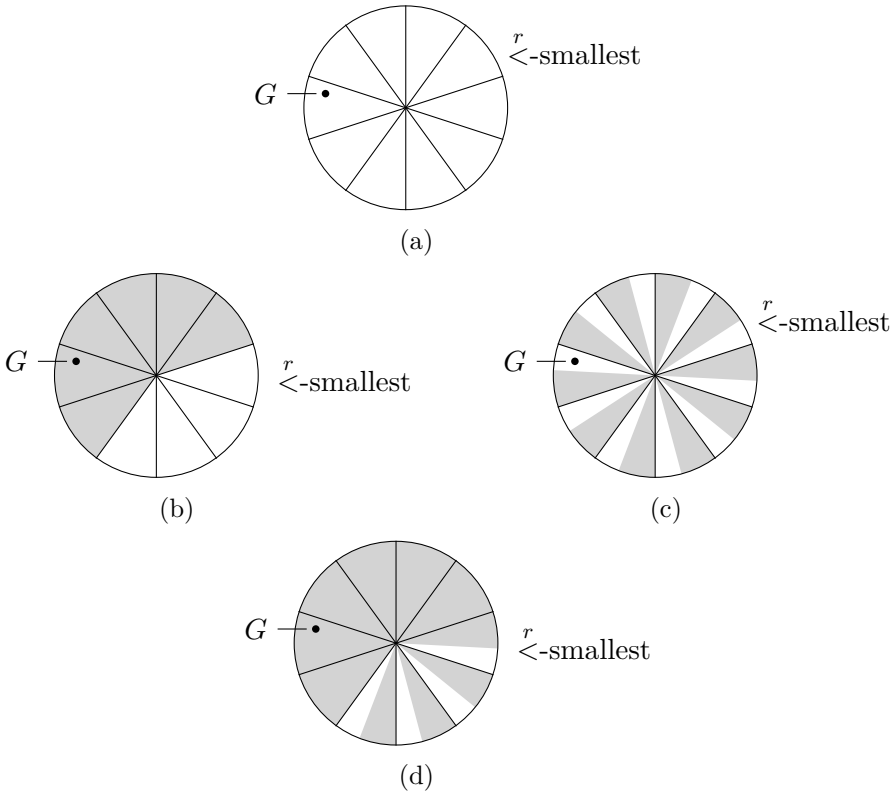


Figure 4.5: Abstract depiction of the set of all permutations of a graph G. Each slice of the set is a subset of permutations that are representational equal. One of the subsets correspond to the $\overset{r}{<}$-smallest representation. (a) If the identity refinement function is used the algorithm explores all $n!$ permutations, and the canonical form is only determined by the $\overset{r}{<}$-relation. (b) With a non-trivial refinement function, some equivalence classes of representations (depicted in grey) will not be considered. The canonical form may now be different, if the global $\overset{r}{<}$-smallest equivalence class was pruned. (c) Using graph automorphisms it is possible to prune among the equivalent graphs, in all classes (see Section 4.3.3). (d) In an efficient implementation both types of pruning would most likely be used.

of how they change the core algorithm we first introduce a different view on canonicalisation. Let G be a graph with n vertices, to be canonicalised in the sense that a "best" permutation from the permutation group S_n must be found. Consider all permuted graphs G^γ for $\gamma \in S_n$, and group them by representational equality as illustrated in Figure 4.5. Using the brute-force approach with the identity function for refinement the search tree will contain all graphs, as indicated in Figure 4.5a. The canonical form will be determined purely by the $\overset{r}{<}$-relation on graphs. A refinement function does however not need to preserve this minimum graph, as depicted in Figure 4.5b.

For example the $\overset{r}{<}$-relation can order vertices by descending degree and the refinement function can order them by ascending degree. Unless the graph is regular this will produce a different canonical form than the brute-force algorithm.

A different type of optimisation is based on detecting automorphisms in the graph and using them to prune redundant subtrees. This pruning situation is depicted in Figure 4.5c, and explore in detail in Section 4.3.3.

4.3.1 Exploiting Vertex and Edge Labels

The core algorithm works both for labelled and unlabelled graphs as the final graph comparison with $\overset{r}{<}$ is assumed to compare labels. However, the labels can be incorporated earlier to provide stronger refinement.

Vertex Labelled Graphs

Let the input graph $G \in \mathcal{G}$ be vertex-labelled with elements from a set Ω_V with the labelling function $l_V \colon V \to \Omega_V$. Additionally let the operators $\overset{r}{<}$ and $\overset{r}{=}$ be defined for pairs of elements Ω_V, in a manner consistent with isomorphism checking between graphs of $\mathcal{G}$.

Instead of starting the algorithm with the unit partition we can construct a finer partition π' by partitioning and sorting V by the labels. That is, we construct π' such that for all $u, v \in V$

$$\begin{aligned} &cell(u, \pi') = cell(v, \pi') && \text{if } l_V(u) \overset{r}{=} l_V(v) \\ &cell(u, \pi') < cell(v, \pi') && \text{if } l_V(u) \overset{r}{<} l_V(v) \end{aligned}$$

Note that using π' instead of the unit partition may change the canonical form calculated by the algorithm.

In the literature of the mentioned algorithms the incorporation of vertex labels is done implicitly by canonicalising both a graph and an ordered partition, instead of just a graph as described here.

Edge Labelled Graphs

Edge labels can be incorporated in at least two different ways. One method is to construct a vertex labelled, bipartite graph $G' = (V \cup E, E')$ where vertices $V \cup E$ are labelled both with their labels in G and with a tag specifying their membership in V and E. The new graph G' is then canonicalised where the labels can be incorporated as described above.

We also suggest another method which does not require the construction of an auxiliary graph, however it may not be feasible (or practical) for all types of labels. Let Ω_E be the set of edge labels, and assume we use the refinement function based on vertex degrees described above. Instead of simply

counting the total degree we can count the occurrence of each unique label. E.g., for chemical molecules we can use a 4-vector to count the vertex degree with respect each bond type. In general the requirement for such generalised counters is that they have a total order defined.

4.3.2 Node Invariants

We can define a pruning scheme based on computing a isomorphism invariant value $\phi(\tau)$, in a totally ordered set, for each tree node τ in the following way (see also [Junttila & Kaski 2007, Section 3.3]). For a node τ in the search tree let $\tau_1, \tau_2, \ldots, \tau_l$ denote the path from the root node $\tau_r = \tau_1$ to $\tau = \tau_l$. Then let $\overline{\phi}(\tau) = (\phi(\tau_1), \phi(\tau_2), \ldots, \phi(\tau_l))$ be the sequence of node invariants for the path, starting from the root. Given a node τ at depth l and a node τ' at depth l' we can now lexicographically compare the first $\min\{l, l'\}$ elements of $\overline{\phi}(\tau)$ and $\overline{\phi}(\tau')$. If they are different we can decide that the node with the smallest invariant prefix is better, and prune the subtree rooted at the node with the larger invariant prefix.

In the literature various invariants have been proposed, e.g., counting the number of cells in the partitions, or the sequence of cell sizes. The name of Traces stems from the introduction of a new invariant which contains the sequence of positions of every cell split. That is, if

$$\pi = (V_1, V_2, \ldots, V_{t-1}, V_t, V_{t+1}, \ldots, V_r)$$

is split into

$$\pi = (V_1, V_2, \ldots, V_{t-1}, V_{t'}, V_{t''}, V_{t+1}, \ldots, V_r)$$

then the position of the split is $\sum_{1 \leq i \leq t'} |V_i|$. When considering partitions to be stored in arrays, the split position is exactly the index of the first vertex in the cell $V_{t''}$. As cells are split during the refinement procedure it is possible with this invariant to abort a partition refinement if it is evaluated to be worse than the currently best trace of cell splits. The pruning by node invariants is also described in general by items (A) and (B) in [McKay & Piperno 2014b, Section 2.4].

4.3.3 Automorphism Discovery

Consider again the search tree in Figure 4.4 and note that the two partitions

$$\pi_{(1,7)} = [1 \mid 2 \mid 7 \mid 10 \mid 8 \mid 9 \mid 6 \mid 5 \mid 4 \mid 3]$$
$$\pi_{(1,8)} = [1 \mid 2 \mid 8 \mid 9 \mid 7 \mid 10 \mid 5 \mid 6 \mid 3 \mid 4]$$

give representationally equal graphs. That is, if $G' = G^{\pi_{(1,7)}^{-1}}$ and $G'' = G^{\pi_{(1,8)}^{-1}}$ then $G' \overset{r}{=} G''$. Therefore there is a trivial isomorphism from G' to G'' that

maps the ith vertex of G' to the ith vertex of G''. We can recast this into an automorphism α on G, which can be described as $\alpha(\pi_{(1,7)}(v)) = \pi_{(1,8)}(v), \forall v \in V$ (see also [McKay 1981, 2.18]). The automorphism is thus

$$\begin{aligned} \alpha \circ \pi_{(1,7)} &= \pi_{(1,8)} && \Leftrightarrow \\ \alpha &= \pi_{(1,8)} \circ \pi_{(1,7)}^{-1} \\ \alpha &= (3\ 4)(5\ 6)(7\ 8)(9\ 10) \end{aligned}$$

Geometrically this automorphism mirrors the graph in the axis through vertex 1 and 2. Such a morphism is called an *explicit automorphism*, as we construct it from explicitly by comparing leaves of the search tree. As described in [McKay & Piperno 2014b, Section 2.4] the set of all explicit automorphisms found during the algorithm generates the automorphism group of G. In some cases it is also possible to find *implicit* automorphisms without comparing leaves, when non-discrete partitions have a certain structure, e.g., see [McKay 1981, 2.24] and [McKay & Piperno 2014b, Sections 3.5 and 3.6].

Any available automorphism can be used to prune the the search tree. Consider again Figure 4.4 and the partition $\pi_{(1)} = [1 \mid 2 \mid 7\ 8\ 9\ 10 \mid 5\ 6 \mid 3\ 4]$, and assume we have discovered the automorphism α as described above. Note that α fixes vertex 1, which is the vertex being individualised to reach $\pi_{(1)}$. When we now individualise respectively vertex 9 and 10 we get

$$\begin{aligned} \pi_{(1)} \downarrow 9 &= [1 \mid 2 \mid 9 \mid 7\ 8\ 10 \mid 5\ 6 \mid 3\ 4] \\ \pi_{(1)} \downarrow 10 &= [1 \mid 2 \mid 10 \mid 7\ 8\ 9 \mid 5\ 6 \mid 3\ 4] \end{aligned}$$

Notice though that the permutation of $\pi_{(1)} \downarrow 9$ with automorphism α gives exactly the partition $\pi_{(1)} \downarrow 10$. The refinement function will use the structure of the graph to split cells, but as an automorphism exactly preserves the graph structure the two resulting partitions must induce the same graph representation. We can therefore skip the subtree $\tau = (1, 10)$ (which happens to just be a leaf), as it contains the same leaves as $\tau' = (1, 9)$. The formal proof of this property requires several new definitions and additional theorems. They can be found in [McKay 1981, McKay & Piperno 2014b].

The general pruning scheme is as follows. Let $A \subseteq Aut(G)$ be a subset of known automorphisms on G, and $\tau = (v_1, v_2, \ldots, v_k)$ an internal node of $\mathcal{T}(G, \pi)$. The children of τ will be given by the set of vertices in the target cell $W = Q(G, \pi_\tau)$. Now construct $A' \subseteq A$ as the set of known automorphisms that fixes each $v_i, 1 \leq i \leq k$. That is, all permutations in A' will map each of the already individualised vertices to them self. For each $w \in W$ we can calculate the orbit of w under the permutations A', as the closure of w when applying A'. In our example this means 7 and 8 are in the same orbit when using only α, and 9 and 10 are in the same orbit. The set W is in this manner partitioned into subsets $W_1, W_2, \ldots, W_p$, and it suffices to explore just one child from each of these subsets.

In Figure 4.4 we have greyed out several nodes due to the automorphism pruning. From the discussion above it is clear why node $(1, 10)$ is pruned. However, notice that α also fixes vertex 2, and the same automorphism can thus be used to prune nodes $(2, 10)$ and $(2, 8)$. The last pruned node, $(2, 9)$, is due to the discovery of an automorphism between $\pi_{(1,7)}$ and $\pi_{(1,9)}$, which fixes both vertex 1 and 2.

The original description of nauty [McKay 1981] used both implicit and explicit automorphisms to perform pruning. However, it only used a list of these automorphisms A, but a permutation group is closed under composition. Therefore all permutations constructed from combinations of permutations in A are also automorphisms. When calculating the automorphisms A' that fixes a sequence of vertices, the algorithm thus may miss some of the combined permutations. For example, two known automorphisms could be $\gamma_1 = (1\ 2)(3\ 4)$ and $\gamma_2 = (1\ 2)(5\ 6)$, of which neither fixes vertex 1 and 2. So when pruning children in a tree node descending from individualisation of vertex 1 and 2 it will not be possible to use γ_1 and γ_2 for pruning. If we calculated their composition $\gamma_1 \circ \gamma_2 = (3\ 4)(5\ 6)$ we see that it would be a usable. The automorphism group of a graph may be very large, and it is therefore not feasible to compute the composition closure of the known automorphisms in general. To alleviate this problem the Traces algorithm in practice introduced the use of Schreier-trees [Seress 2003] that can efficiently represent the closure, and facilitate orbit calculations in subgroups with fixed vertices.