ELTE Eötvös Loránd University

Faculty of Science

Graph canonization algorithms

Math project III. report

Szabolcs Ákos Nagy

Applied mathematics MSc

Supervisor:

Péter Madarasi

ELTE Institute of Mathematics

Department of Operations Research

Budapest
2024

1 Summary of the project

The goal of the project is to analyze and understand various graph canonization algorithms and to introduce new methods to potentially improve upon them. The main motivation behind this is that canonical labelings can be used to generate non-isomorphic graphs in an efficient manner. Still, the number of possible graphs, even with respect to isomorphism makes computations infeasible for large n, so our focus is mainly on canonizing graph of size at most 20.

2 Graph canonization

Let us consider a graph labeling process that, given a graph G on n vertices, labels the vertices from 1 to n. We call such a process a *canonization* if, for any two isomorphic graphs, the newly labeled graphs are identical.

We call this labeling the *canonical labeling* or *canonical form* of the graph, with respect to the given canonization process. Different canonization algorithms might assign different labelings to the same graph, but for any one, the resulting graph should always be the same within one isomorphism class. One of the main inhibiting factors in computing a canonical form is that we cannot utilize the starting labeling in a meaningful way. The outcome of the algorithm cannot rely on the order in which the vertices are examined.

An example of an outcome relying on examination order would be when a BFS tree is built. Graphs that are isomorphic can have non-isomorphic BFS-trees, simply because matching vertices are placed into the queue of reached nodes in a totally different order based on their labels. Since we need the output graph to be identical for any two isomorphic input graphs, we cannot let such things to happen. Every time we have to choose what order to examine vertices in, we need to make sure the final outcome will not be affected.

2.1 Computing a canonical labeling

A brute-force canonization algorithm can be given quite simply: establish a fixed ordering on all labeled graphs (e.g., the lexical ordering of their adjacency matrices), then check all $n!$ possible permutations of $V(G)$, picking the resulting form that is first according to the fixed order. It is easy to see that the resulting graph will be the same for isomorphic starting graphs, but also that the runtime is $\Omega(n!)$, which is not ideal.

However, the basic idea of choosing the lexicographically smallest form from a specific pool of forms is still useful: by utilizing graph properties that are invariant to isomorphism, such as the degrees of vertices, the number of different graph forms can be reduced significantly. One of the most sophisticated forms of this idea is currently McKay's and Piperno's nauty project [4].

2.2 Brief overview of McKay's algorithm

The search-tree The main idea presented by McKay in [3] is the effective construction of a search-tree $T(G)$, the leaves of which give the potential labelings of the graph, out of which we will choose the canonical form. This is achieved by maintaining for each $t \in V(T(G))$ an ordered partition Π_t, that is, a partition whose cells are in a specified order.

The root of the tree contains the unit partition, where all vertices are in one cell, and every child in the tree contains a finer version of their parent's partition, so denoting the parent of v as $p(v)$, then we have for all non-root tree node t:

$$\forall V \in \Pi_t : \exists W \in \Pi_{p(t)} : V \subseteq W$$

The leaves of the tree will then in turn be trivial ordered partitions, where every vertex is in its own cell, which gives an explicit ordering of the vertices, and therefore a permutation on them, and a potential labeling of the graph.

If we make sure that, when deciding what children each node should have going down the search-tree, we do so in a manner that is isomorphism-invariant with respect to the current "labeling" the ordered partition gives (vertices of G are labeled from 1 to n starting at the earliest cell and going up, with arbitrary order within cells), then the resulting permutations given by the leaves of the tree will give the same graph forms for any two isomorphic starting graphs. Therefore, the lexicographically smallest one will be the same for each one, and a good choice for a canonical labeling.

Choosing children One such way of choosing children partitions of t would be to simply pick the first non-trivial cell V of Π_t and creating a child for each $u \in V$, in which the partition is the same as Π_t, but with V replaced by $\{u\}$ and $V \setminus \{u\}$ in that order. This method of splitting off a single element of a cell will be referred to as an *individualization* step. It is not difficult to see that doing this will result in the leaves of the tree simply giving each possible permutation, in line with the brute-force method mentioned above.

One of the main ideas of the search-tree canonization algorithm is to decrease the number of children by using isomorphism-invariant graph properties to refine its partition as much as possible before individualizing needs to happen. In McKay's algorithm, this is achieved with the use of *equitable* partitions.

Refinement In this algorithm, a partition Π is considered equitable if, for any two cells $V, W \in \Pi$ (these can be the same), we have $d(v_1, W) = d(v_2, W)$ for all $v_1, v_2 \in V$, where $d(v, W)$ denotes the number of neighbors of v that are in W. We will indicate this with $eq(V, W)$, that is, When this holds for the cells V and W, we will denote it as $eq(V, W) = 1$, and $eq(V, W) = 0$ otherwise.

Now we can define the refinement step as follows: before giving t its children, we check whether Π_t is equitable. If there are two cells, V and W for which $eq(V, W) = 0$, then we split V into several cells $V_1, V_2, \dots V_k$ where $d(v_1, W) = d(v_2, W)$ for any $v_1, v_2 \in V_i$, for all $i \in [k]$, putting these new cells in ascending order according to this common number of neighbors.

We can keep doing this until an equitable partition is reached. In fact, as outlined in [3], there exists a unique coarsest equitable partition Π_t^* which will be reached in this manner. The order of the cells of Π_t^* in the stored ordered partition usually relies on the order in which we examine cell-pairs V, W for $eq(V, W)$, sorting newly split cells as previously described. Provided that the starting partitions are isomorphic, this process is indeed an isomorphism-invariant way to order the cells of Π_t^*. Once the ordered partition is computed, we may replace Π_t with Π_t^* and generate children by individualizing the vertices of its first non-trivial cell.

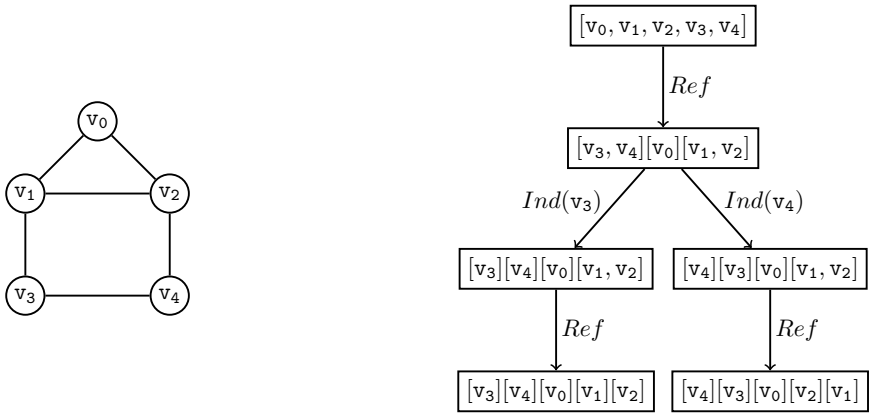


Figure 1: An example of a search-tree with cell refinement

See Figure 1 for an example of reducing partitions with the help of equitability. First we have the unit partition, which is not equitable, since $d(v_0, V) = 2 \neq 3 = d(v_1, V)$, so the partition is $\{v_0, v_3, v_4\}, \{v_1, v_2\}$. Were are not done refining though, since in this partition, $d(v_0, \{v_1, v_2\}) = 2 \neq 1 = d(v_3, \{v_1, v_2\})$, so the bigger cell is once again split, giving us $\{v_3, v_4\}, \{v_0\}, \{v_1, v_2\}$. This is in fact equitable, and the refinement procedure stops. Individualization is utilized before refinement is attempted again.

Notice how performing these splits in no way requires us to know the order in which the vertices of V appear, and relies only on isomorphism-invariant properties. Therefore, the leaves of the search-tree yields identical labeled graphs for isomorphic starting graphs. A more detailed and precise explanation of the correctness of this approach can be found in [3]. This refinement step on its own immensely decreases the size of the search-tree for the vast majority of graphs. Though, for graphs that are highly symmetric, a lot of "individualizing" child generations may still need to be made before refinement can significantly reduce the partitions.

Automorphism-pruning The problem posed by highly symmetric graphs can be somewhat mitigated with the help of the many automorphisms of the graph. The idea of McKay's pruning process

is that when we are individualizing vertices, there is no need to create children for vertices that are in one orbit with one for which we already have, with respect to the current partition. More specifically, if we are individualizing a $V \in \Pi_t$ cell with vertices $v_1, v_2 \in V$, and there exist an automorphism ϕ of G such that $\phi(\Pi) = \phi(\Pi)$ and $\phi(v_1) = \phi(v_2)$, then it is not difficult to verify that the entire resulting branches starting from these two children would result in the exact same graph forms obtained from the leaves, with ϕ being able to map the leaves reached from the v_1 branch to the v_2 branch, therefore, it is enough to individualize v_1.

An efficient way of computing whether such and automorphism exists in a graph is not known, so the way this is handled is that graph form given by the first reached leaf is stored, and any leaf that gives the same form going forward gives us an automorphism, the permutation that turns the permutation of the first leaf into the permutation of the new one. Then, we always check whether the above reduction can be made with any of the automorphisms already found. Shown below is how many search-tree nodes are visited in total during the canonization of certain graph families, and how much time is spent on them. k-labeled means that all labeled graphs on k vertices are canonized, and k-unlabeled means the same for unlabeled graphs.

Graph type	No pruning # nodes	Pruning # nodes	No pruning runtime	Pruning runtime
5-labeled	5 124	4 502	48 ms	43 ms
6-labeled	139 995	127 915	1,2 s	990 ms
7-labeled	6 741 158	6 428 164	203 s	183 s
7-unlabeled	7 326	5 874	117 ms	109 ms
8-unlabeled	59 508	52 137	1,8 s	1,62 s
9-unlabeled	841 363	792 173	64 s	60 s

2.3 What could potentially be improved

Modified refinement While the equitability-ensuring refinement procedure in McKay's algorithm is effective in making the initial partition finer, it is relatively arbitrary in the sense that any cell-symmetry condition that we can ensure in an isomorphism-invariant manner would also provide an adequate way to refine the partition. One approach to improving the search-tree algorithm is to make the condition of "equitability" stricter or different, trying to achieve a finer partition before individualization becomes necessary. This semester, one such stricter way of refining cells was explored. When taking this approach, something to keep in mind is that while the size of the search-tree might decrease, the time spent on refining each partition and assuring that they satisfy the stricter equitability condition could outweigh the time saved by the decrease in amount of tree nodes. Over-committing to refinement calculations might prove counterproductive.

Invariant calculation Another method of getting a finer cell before individualization is to establish a family of isomorphism-invariant functions for all graphs, that is, functions $f_G : V(G) \times \mathcal{P}_{V(G)} \to \mathbb{R}$ (where $\mathcal{P}_X$ denotes the set of ordered partitions of X) for which, given two isomorphic graphs G_1, G_2 with isomorphic ordered partitions Π_1, Π_2, then for any isomorphism ϕ for which $\phi(G_1) = \phi(G_2)$ and $\phi(\Pi_1) = \phi(\Pi_2)$, we have for any $v \in V(G_1)$:

$$f_{G_1}(v, \Pi_1) = f_{\phi(G_1)}(\phi(v), \phi(\Pi_1)) = f_{\phi(G_1)}(\phi(v), \Pi_2)$$

It is clear that when applying $f_.$ to isomorphic graphs with isomorphic ordered partitions, the isomorphic vertices will receive the same values between the graphs. Therefore, computing $f_.$ for each vertex and ordering the vertices based on the received values will give the same order for isomorphic graphs, so refining the partition by leaving vertices with the same $f_.$ value in one cell, ordered by that value will leave to the same ordered partitions in the search-trees of isomorphic graphs.

At first glance it may seem like vertices in one orbit of the graph will always have the same $f_.$ value, which would not be particularly helpful in refining the partition, but keep in mind that the ordered partition we already have can be used in the computation of $f_.$, so after the individualization of vertices, orbits may be separated into several pieces by $f_.$, based on relation to vertices that have already been split off.

3 Distant neighbors

The base refine step of McKay's algorithm uses degree information in the current partition to differentiate between vertices of cells. If the immediate neighbors are not in the same cells for all vertices

of a given cell, then that one is split apart to fix this issue. As more and more individualization steps are made, neighborhoods of cells become more and more defined, and vertices can start splitting apart from one another based on how "far away" they are from the individualized vertices.

Thus, one idea to make refinement more effective in splitting the partition is to get ahead of the individualization step in recognizing differences between vertices based on their "distance". Similar concepts have been explored before, such as in [7], where vertices are differentiated based directly on their distance from partition cells, but we are examining and using "vertex-distances" in a different manner.

3.1 k-equitability

One may look at McKay's equitability condition in the following sense: an ordered partition Π is equitable if, for any two cells $V, W \in \Pi$, the number of 1-long walks starting at v and ending in W is the same for all $v \in V$. Let us say that in this case, Π is 1-equitable.

This "1-long walks" thought is just a different way of writing down the degrees of vertices, but it opens the possibility of doing the same thing for numbers higher than 1. We might just as easily say that we require the above property, but also for 2-long, 3-long, all the way up to k-long walks. When this condition is satisfied for two specific cells V and W, that is, $d_2(v, W)$ is the same for all $v \in V$, we will denote it as $eq_k(V, W)$. When this stricter k-equitability condition holds, that is, $eq_k(V, W)$ holds for all cells $V, W \in \Pi$, we say that the partition is k-equitable.

It should be emphasized that we are talking about *walks*, and not paths. Deciding whether a k-long path exists between two vertices at all is an NP-complete problem, but the same thing for walks is rather simple. We simply have to check what the neighborhood of the neighborhood of (so on...) the neighborhood of v is, which trivially takes at most mk steps. It is not difficult to verify that, much in the same way as the regular 1-equitability condition, k-equitability is isomorphism-invariant for any k.

Example At first, one might intuit that if 1-equitability is satisfied, then this perhaps also implies k-equitability for higher k, but this simply is not the case.

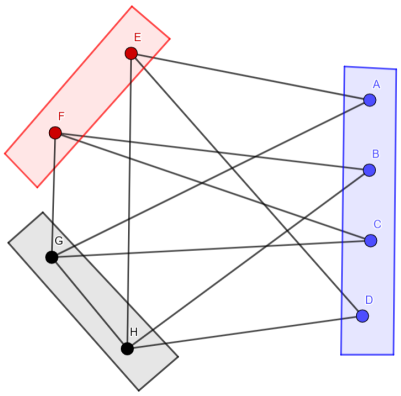


Figure 2: Example of a 1-equitable, but not 2-equitable partition

Consider the graph in Figure 2. The vertices are "labeled" A through D, and the partition has 3 cells: $\Pi = \{V_1, V_2, V_3\}$, with $V_1 = \{A, B, C, D\}, V_2 = \{E, F\}, V_3 = \{G, H\}$. The edges are as seen in the Figure. It is not difficult to verify that this partition is equitable. We can see that the for any V_i and V_j, we have $d(v, V_j) = M(i, j)$ for all $v \in V_i$, with $M = \begin{bmatrix} 0 & 1 & 1 \\ 2 & 0 & 1 \\ 2 & 1 & 1 \end{bmatrix}$.

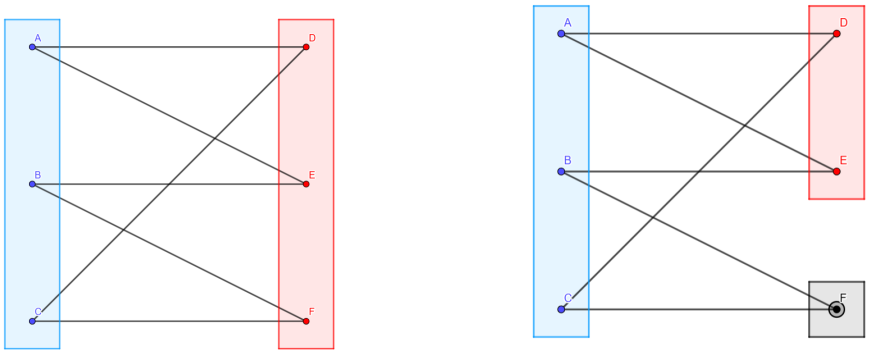
Therefore, Π is 1-equitable, but in fact, not 2-equitable. One can check that vertex C has a 2-long walk to G, as well as H. There is, however, no 2-long walk from A to G, therefore, $eq_2(V_1, V_3) = 0$. In this example, the coarsest finer 2-equitable partition would be $\Pi^* = \{\{A, B\}, \{C, D\}, \{E, F\}, \{G, H\}\}$.

3.2 Ensuring 2-equitability

For now, let us focus on the case of $k = 2$. First, one might attempt to use the regular refinement to get a 1-equitable partition, then look for any cells V, W that break the condition of 2-equitability.

However, this is not an ideal approach in the sense that we have no direct pointers as to which cells could be problematic.

When checking for 1-equitability after individualization, it is enough to start by checking that $eq(V, W) = 1$ still holds when taking W as one of the two new cells, as all other cells $Z \in \Pi$ still have the same elements, for which we already know that $d(v, Z)$ is the same, splitting the starting cells does not change this. Thus, the newly created cells are the only "dangerous" cells that may ruin the property.



(a) An equitable partition

(b) Partition after split

Figure 3: Individualization, two new dangerous cells

For example, see Figure 3. In 4a, we have an equitable partition, both classes connect to the other one twice. However, after the individualization of the vertex F in 3b, the elements of the cell on the left connect to the cells on the right with different multiplicity:

$d(v, W)$	A	B	C
$\{D, E\}$	2	1	1
$\{F\}$	0	1	1

Notice how every vertex still clearly has the same number of neighbors in the cell on the left, since it has not changed. Thus, it cannot be W in a violating $eq(V, W) = 0$ pair.

This still holds true later down the line in a single refinement, that is, whenever a cell is split because it has elements with cell-neighborhoods of different multiplicity, it is enough to add the newly formed cells to the pool of dangerous cells, and then check whether $eq(V, W) = 1$ still holds for any $V \in \Pi$ and dangerous W. Note how this exact strategy still works when checking for k-equitability, with the exact same reasoning for correctness.

By waiting for the 1-equitability condition to be satisfied, we would have to re-evaluate every cell that, in its current form, was deemed dangerous during the computation of the 1-equitable partition. Instead, it is more efficient to check for higher conditions right beside the regular one: separate elements of V not just based on $d(v, W)$, but also $d_2(v, W)$. To do this, we need an efficient way to count the number of 2-distance neighbors each vertex has in a given cell. To achieve this, we need to calculate a "second neighbor" version of the adjacency matrix, denoted here as A_2, where $A_2(i, j)$ is 1 when there exists a 2-long walk from i to j.

From here, it is not difficult to modify the regular cell-splitting step to ensure 2-equitability: Instead of just calculating $d(v, W)$ for all $v \in V$ in the current V candidate, we calculate $d_2(v, W)$ as well. Then, we sort all elements of the cell lexicographically based firstly by d, then d_2, splitting them into several cells when appropriate based on how many different pairs we have calculated.

Like in the original algorithm, radix sort is ideal here, given how the number of different values is relatively low, and the provided stability between the sorting of the two calculated values makes our lives a lot easier as opposed to having to assure manually that the appropriate vertices come after one another in the correct order. This effectively doubles the amount of time we spend on each part of the refinement step, since we check two indicator vectors for elements of W in every split check: the first *and* second neighbors of v.

3.3 Ensuring k-equitability

We can similarly extend the idea for further neighborhoods: whenever the cells V, W are checked for $eq_k(V, W)$, we calculate a vector $x_v \in \mathbb{N}^{\Pi}$ for all $v \in V$, where $x_v(i) = d_i(v, W)$. Then, we

lexicographically sort the elements of V based on their vectors, with the new cells defined by vertices with the same vectors.

This not only requires us to calculate A_2 through A_k at the start of the canonization, but every time two cells are examined during refinement, we have to calculate the k-long vector for each vertex, essentially taking k times as long as it would have to ensure just 1-equitability.

4 Complete graph invariants

In line with the previously mentioned invariant-calculations, an alternative approach to graph canonization is not with the explicit calculation of a labeling, but with the computation of a certain isomorphism-invariant h, which is assured to be different for non-isomorphic graphs, that is, $h(G_1) = h(G_2)$ exactly when the two graphs are isomorphic. Such an h is called a *complete graph invariant*. Here the $h(G)$ value is considered the "canonical form" of G, much in the same way that the canonically labeled graph was earlier. It is often not trivial to acquire an actual canonical labeling from this h value, especially in polynomial time, but they can still be useful nonetheless.

4.1 Tree-coding

One of the most well-known examples of this is Read's tree-coding algorithm [6], which "labels" every vertex of a tree with a string of ones and zeroes in polynomial time. Here, the hash value is the tree, with its vertices labeled with the calculated strings. These labels can be the same for several vertices, so this is no way a canonical labeling according to our earlier definition, but it can be proven that the resulting labeled trees are identical exactly when the starting trees are identical.

The idea of the algorithm is to start off with the initial label 01 on each vertex, then as a general step, use isomorphism-invariant graph information (leaf vertices, strings of neighbors) to modify it. In each iteration, we remove the leaves of G, finalizing their string, and keep going. We do this until no more vertices remain.

Utilizing tree-coding The above algorithm relies heavily on the fact that there are leaves in every iteration, therefore it only works on trees (and arborescences), so one might try to modify it in order to be applicable to general graphs, and to actually give a proper labeling when we need one.

Firstly, the main way such a faux-labeling can be used is to split the vertices of G into cells based on their received "label", treating the strings as the previously discussed $f_\cdot$ values, and continuing the search-tree algorithm from there. One idea to compute such a partition with the previous algorithm is the following: in case of a graph with bounded or low tree-width, for which we can calculate an isomorphism-invariant tree-decomposition, we can canonize the nodes of the tree-decomposition, and then label vertices of G the same as the lexicographically smallest node of the heaps in which they appear.

One such way of computing a tree-decomposition has been explored by Lokshtanov [2], with the decomposition in question serving as a complete graph invariant. This, however, involves an FPT scheme, which for a fixed k either concludes that the tree-width of the graph is over k, or gives such a tree-labeling, with width $\mathcal{O}(k^4)$. This is not very viable on the level we are examining graph canonization on, but is something to keep in mind for canonization of large graphs, especially ones with known tree-widths.

4.2 Coding of non-trees

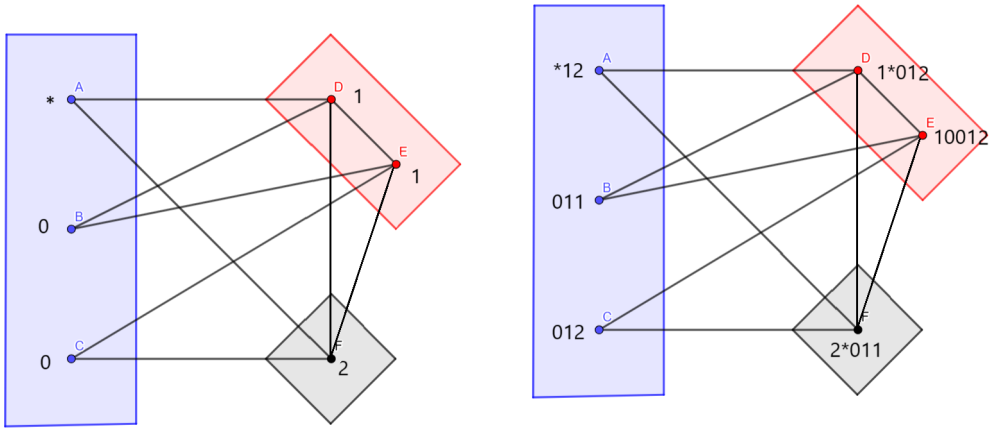
Jüttner and Madarasi [1] have constructed a method of vertex string-hashing similar to that of Read, which can be used on general graphs. It too runs in polynomial time, and calculates a string of numbers for each vertex as a faux-labeling, although it is not quite as simple to use this to determine isomorphism between general graphs. Moreover, this algorithm can be slightly modified to be able to include an initial partition of the vertices, which means that the previously outlined sort-partitioning can be utilized several times in the search-tree, after each individualization.

The algorithm follows the sentiment of Read's algorithm, as well as the previously brought up ensuring of k-equitability, in the sense that we propagate the initial strings along neighbors, so vertices with different k-neighborhoods will have different strings by the end of the k-th iteration of the algorithm.

Initially, let us set the string of each vertex to be the cell in which it resides in the current partition. Then for each vertex v, we will do the following: We first replace the label of v with a special unique label "$*$", and then, we iterate a concatenation step.

In this step, for every $z \in V$ vertex, we take the neighbors of z, sort their labels lexicographically, and concatenate them to the end of the label of z. For this, the algorithm considers the labels of these neighbors as they appeared at the start of the iteration step, so the order in which we do this does not affect the label each vertex ends up with after all concatenations are done. Once we have these concatenated labels, we sort them lexicographically, and replace them with their placement in the order (this ensures that the labels do not become too long), and another concatenation step can take place. See Figure 4 for an example of what this step looks like.

We do this for however many k iterations that we want, or until different labels stop showing up, which is at most $V(G)-1$ cycles. After all concatenation cycles are finished, we take the current label of every vertex, sort them lexicographically, concatenate them together, and finally write that string onto v as its final label. We do this for every $v \in V$, always considering the labelings each vertex had at the beginning of the algorithm. This way, we will eventually end up with a final string for every vertex, this is going to be the faux-labeling based on which we can form our new ordered partition.



(a) Initial labels in the calculation for A

(b) Labels after one concatenation cycle, trivial partition already achieved

Figure 4: Example of label concatenation during computation of the final label of A

Notice how doing this with only $k = 1$ concatenation cycles for each vertex, then the resulting partition will be finer exactly when the initial partition is not equitable, since the initial labels of each vertex within one cell being the same is exactly what it means both for the partition to be equitable, and for the final strings to be equal after one concatenation cycle on each vertex. While this algorithm essentially ensures an even further generalization of k-equitability, it also involves a lot more calculation. Nevertheless, if we find an efficient way of calculating only what we want from this labeling, such as only calculating it in certain cells when we know we might be able split there, it might still be useful in the computation of a canonical labeling.

5 Stationary distributions

In this section, we introduce a different approach to decreasing the size of the search-tree, taking the route of invariant calculation. The base idea is the following: consider a Markov-chain, the elements of which are the vertices of the graph, and one can step from one element to another with positive probability if there is an edge between them.

If these probabilities are defined in an isomorphism-invariant manner, and the stationary distribution of this Markov-chain is unique, then it is also isomorphism-invariant, meaning that separating the vertices of the graph based on their distribution value is a viable way of making the initial partition finer before the cycle of refining and individualizing steps above.

5.1 The Markov-chain

The most straightforward idea of constructing an isomorphism-invariant Markov-chain from the graph G is the following: the elements of the ground set are the vertices of G, and $p(u,v) = \frac{1}{d(u)}$ for all $u, v \in V(G)$, so we may step to any neighbor of u with equal probability.

Unfortunately, this is not quite sufficient for us, as there could easily be several distinct stationary distributions on this Markov-chain, and therefore we may end up computing completely distinct ones for different starting graphs, even if they are isomorphic. Thankfully, we can assure the uniqueness of the stationary distribution with relatively little modification of the construction.

It is a well-known fact of stochastic processes that when a Markov-chain is irreducible and aperiodic, its stationary distribution is unique [5]. With the above graph-to-Markov-chain transformation, this would entail that G is connected, and that there is a $n_0 \in \mathbb{N}$ such that for any $u, v \in V$ and $n \geq n_0$, there exists an n-long path of positive-probability steps from u to v. Both of these can be achieved by simply adding another "connecting" vertex z to the graph, which is connected to all vertices in the graph, including itself. Let us call this new graph G'. It is easy to see that G' is connected, and that $n_0 = 2$ satisfies the second condition, since we can step into z in one step, then just stay there using its loop for as many as we want before terminating in the goal vertex.

It is important to verify that this does not violate the isomorphism-invariance of the calculations. We need to assure that adding this extra vertex yields isomorphic graphs exactly when the starting graphs are isomorphic. Notice that with starting graphs G and H, the resulting G' and H' received this way are only ever isomorphic if G and H were isomorphic to begin with, since the "connecting" vertex z is the only one with maximal degree in both of them, so any isomorphism between the bigger graphs will pair up the z-equivalents between the two, and the rest of the isomorphism will still be one when applied to only the smaller graphs. Clearly, if G and H are isomorphic, G' and H' are as well.

5.2 Computing the distribution

Let P be the transition matrix of the Markov-chain constructed in the above way from G', where $P(i,j)$ is the probability of stepping from vertex i to j. Then, we are looking for a non-zero row vector μ such that $\mu P = \mu$, or, brought to a more conventional linear equation form a column vector for which $(P^T - I)\mu = 0^{n+1}$. Note that the unique stationary distribution is in fact a proper μ, but also any scalar-multiple of this, as well as the 0-vector. We can make it easier for an equation solver to find the desired non-zero solution by ensuring that the elements of μ add up to 1, that is, $1^{n+1^T}\mu = 1$. So, all in all, we need to find a $\mu \in \mathbb{R}^{n+1}$ such that:

$$\begin{bmatrix} P^T - I \\ \mathbb{1} \end{bmatrix} \mu = \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix}.$$

Here, the linear equation system has one less variable than there are conditions, which may look like it could lead to a lack of solutions, but keep in mind that we know for a fact that exactly one such μ exists. After μ is calculated, using it to get a finer starting partition is simple: sort the vertices of G based on the distribution value that were attributed to them by μ, and the cells will contain vertices with the same μ value. This will lead to isomorphic ordered partitions with isomorphic starting graphs.

5.3 Utilizing partitions

Once the basic methodology is established, the thought naturally occurs: can we use distributions to refine the partition more than once? Let us say that we already have some Π partition, through some combination of refining and individualizing steps. Is there a yet still isomorphism-invariant way to modify the above method such that the resulting distribution is finer than what we got before?

It turns out that this is indeed achievable by simply modifying the probabilities of the Markov-chain. We could make it so that when checking to see where to step, we step toward vertices in different cells with different probabilities. One way of going about this is to add parallel edges to G' for the purpose of skewing the probabilities of the Markov-chain, based on the lexical cell distance of nodes, that is, given ordered partition $\Pi = \{V_1, V_2, \dots V_r\}$ with $r \geq 2$ (since we are no longer in the starting partition), for every $v_i \in V_i$ and $v_j \in V_j$, there are $|i - j| + 1$ parallel edges between v_i and v_j if $v_i v_j \in E(G')$. As for the "connecting" vertex, z one might similarly say that there are i edges connecting z and v_i for $v_i \in V_i$.

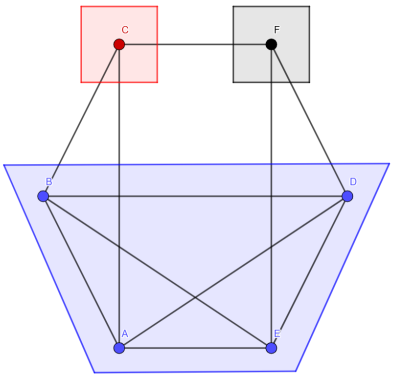


Figure 5: Partition that can be further refined with the help of the Distribution

See Figure 5 for an example where this helps us refine a partition. The partition consists of the cells $V_1 = \{A, B, D, E\}, V_2 = \{C\}, V_3 = \{F\}$. Here, we can set the probabilities of the steps Markov-chain by just the edges, that is, when standing in vertex v, we can step along every edge starting at v with equal probability, resulting in transition matrix P. On the other hand, we can consider "inserting parallel edges" in the previously described manner, and then setting the probabilities accordingly, resulting in transition matrix P'. Including the extra z connecting vertex, these matrices will be the following (rounded to two decimal places):

$$P = \begin{bmatrix} 0 & 0.2 & 0.2 & 0.2 & 0.2 & 0 & 0.2 \\ 0.2 & 0 & 0.2 & 0.2 & 0.2 & 0 & 0.2 \\ 0.25 & 0.25 & 0 & 0 & 0 & 0.25 & 0.25 \\ 0.2 & 0.2 & 0 & 0 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0 & 0.2 & 0 & 0.2 & 0.2 \\ 0 & 0 & 0.25 & 0.25 & 0.25 & 0 & 0.25 \\ 0.14 & 0.14 & 0.14 & 0.14 & 0.14 & 0.14 & 0.14 \end{bmatrix}, P' = \begin{bmatrix} 0 & 0.16 & 0.33 & 0.16 & 0.16 & 0 & 0.16 \\ 0.16 & 0 & 0.33 & 0.16 & 0.16 & 0 & 0.16 \\ 0.29 & 0.29 & 0 & 0 & 0 & 0.29 & 0.14 \\ 0.14 & 0.14 & 0 & 0 & 0.14 & 0.43 & 0.14 \\ 0.14 & 0.14 & 0 & 0.14 & 0 & 0.43 & 0.14 \\ 0 & 0 & 0.22 & 0.33 & 0.33 & 0 & 0.11 \\ 0.1 & 0.1 & 0.2 & 0.1 & 0.1 & 0.3 & 0.1 \end{bmatrix}$$

Calculating the distribution with the formula in 5.2 with P, we will get the partition $\{A, B, D, E\}, \{C, F\}$, even coarser than what we already have, but substituting P with P', we get $\{A, B\}, \{D, E\}, \{C\}, \{F\}$, so we have successfully refined the partition.

More meticulous care could be taken to differentiate between cell-connections ever further, but the main goal is to ensure that when there is an option to step into different cells, these chances are different. This, just like everything before, will lead to ordered partitions that are isomorphic, given that the initial ones were as well.

5.4 Results

While our initial hope was that this would result in refining the partition into cells that the regular, degree-information refinement step of McKay's algorithm could not, but it turns out that this was an overly optimistic expectation. In our tests, not one single time did the stationary distribution give a finer partition than what the refinement procedure would give. In fact, after due consideration, we proved that for any starting partition Π, the finer partition given by the stationary distribution of an isomorphism-invariant Markov-chain on $V(G')$, in which we always step from the current position into a every vertex of a given cell with equal probability (and z is always in its own cell) is either the unique coarsest finer equitable partition Π^*, or is coarser than it.

The basic idea of the proof is taking the cells of Π^* as the elements of another Markov-chain, and setting step-probabilities based on how many neighbors the vertices in a cell have in other cells. Extending the Markov-chain with z as previously, setting the probabilities in the same way as in the original G', we will receive a Markov-chain with a unique stationary distribution, which can be used to construct a stationary distribution μ on G' with partition Π^* with the same probability-settings as we had with Π. This means that μ is in fact also a stationary distribution on G' with Π.

5.5 Utilizing distant neighbors

Recall that our Markov-chain does not necessarily need to be constructed in the specific way we did, the only requirement for the construction is to be isomorphism-invariant with respect to partitions. Recall how we calculated the matrix of second neighbors A_2 during testing for 2-equitability. We can just as easily say that in G', there is also a certain number of edges between two vertices if there is a 2-long walk between them, the amount depending on which cells the two vertices are in.

Calculating the stationary distribution of this stationary distribution did, in fact, give partitions that were finer than what we would have gotten with the regular refinement step. Recall when we previously discussed how some partitions are 1-equitable, but not 2-equitable. This Markov-chain, essentially including 2-long walks as edges, contains the information necessary for the distribution to "recognize" this non-2-equitableness, given that we choose the step-probabilities appropriately.

Note how the previous reasoning can also be used to demonstrate how this distribution will be a coarser version of the coarsest finer 2-equitable partition, but this time, the only part of the calculation that takes longer is setting the values of the matrix in the linear equation. Take note also of the fact that this whole line of thought can be extended to k-long walk Markov-chain steps and coarser-than-k-equitable partitions, where the only part that takes k times longer is, once again, setting the values in the matrix.

Keep in mind that when always trying to differentiate between unique kinds of Markov-chain steps, such as steps between different cells, steps representing walks of different length and so on with differing probabilities, the number of floating points we need to calculate in order to remain proper in our solution increases, or, if we attempt to calculate everything using rational numbers, the denominators can climb all the way up to n^k, which can get out of hand if we are not careful. This can either be solved by either rounding results when the denominators get too high, or by sacrificing distinction between different probabilities, both at the expense of information contained within the resulting distribution. Below are shown computational results of the algorithm, using this distribution to create new cells before regular equitability checking was used. Here, we consider the time taken to calculate the distribution to is negligible, that is, an efficient linear equation solver is available.

Graph type	No pruning # nodes	Pruning # nodes	No pruning runtime	Pruning runtime
5-labeled	5 124	4 502	46 ms	39 ms
6-labeled	139 935	127 915	990 ms	987 ms
7-labeled	6 739 073	6 434 494	185 s	181 s

We can see that this results in slight improvement in both search-tree size and runtime, however, in the case where automorphism pruning is used, it only manages the calculate the labelings by visiting just as many, in fact, more search tree nodes than without the distribution, with little improvement in runtime. This happens because when pruning the tree, branches can be dismissed when an appropriate automorphism is found, and sorting the vertices by distribution value can result in leaves being reached in different order, and automorphism can sometimes, like here, can only be used later to prune the tree.

Considering also that calculating the distribution does actually take a non-negligible amount of time, this shows us that while often more effective at refining individual partitions than regular equitability-insurance, this type of distribution-calculation as a means to accelerate canonization is not better than regular automorphism-pruning.

6 Closing thoughts, goals for the future

In this semester, we introduced and implemented a generalization of McKay's equitability condition in the form of k-equitability, and a new method of computing an isomorphism-invariant on the vertices of a graph in the form of the stationary distribution. While significant reduction to the runtime of the algorithm or the size of the search-tree has not been attained, the mentioned methods which have the potential of being used in the computation of complete graph invariants.

Our future goals include implementing cell-refinement improvements related to k-long walks for general k, consisting of k-equitability ensuring and the Markov-chain including k-distance neighbor information. We also intend to look into more ways to achieve a finer partitioning of the vertices of a graph, and to determine a canonical labeling from such.

References

[1] Alpár Jüttner and Péter Madarasi. A graph isomorphism invariant based on neighborhood aggregation, 2023. URL: https://arxiv.org/abs/2301.09187, arXiv:2301.09187.

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

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

[4] Brendan D. McKay and Adolfo Piperno. Nauty and traces. URL: https://pallini.di.uniroma1.it/.

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

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

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