Tight Lower and Upper Bounds for the Complexity of Canonical Colour Refinement

Christoph Berkholz[1], Paul Bonsma[2], and Martin Grohe[1]

[1] RWTH Aachen University, Germany
{berkholz,grohe}@informatik.rwth-aachen.de
[2] University of Twente, The Netherlands
p.s.bonsma@ewi.utwente.nl

Abstract. An assignment of colours to the vertices of a graph is *stable* if any two vertices of the same colour have identically coloured neighbourhoods. The goal of *colour refinement* is to find a stable colouring that uses a minimum number of colours. This is a widely used subroutine for graph isomorphism testing algorithms, since any automorphism needs to be colour preserving. We give an $O((m+n)\log n)$ algorithm for finding a *canonical* version of such a stable colouring, on graphs with n vertices and m edges. We show that no faster algorithm is possible, under some modest assumptions about the type of algorithm, which captures all known colour refinement algorithms.

1 Introduction

Colour refinement (also known as *naive vertex classification*) is a very simple, yet extremely useful algorithmic routine for graph isomorphism testing. It classifies the vertices by iteratively refining a colouring of the vertices as follows. Initially, all vertices have the same colour. Then in each step of the iteration, two vertices that currently have the same colour get different colours if for some colour c they have a different number of neighbours of colour c. The process stops if no further refinement is achieved, resulting in a *stable colouring* of the graph. To use colour refinement as an isomorphism test, we can run it on the disjoint union of two graphs. Any isomorphism needs to map vertices to vertices of the same colour. So, if the stable colouring differs on the two graphs, that is, if for some colour c, the graphs have a different number of vertices of colour c, then we know they are nonisomorphic, and we say that colour refinement *distinguishes* the two graphs. Babai, Erdös, and Selkow [2] showed that colour refinement distinguishes almost all graphs (in the $G(n, 1/2)$ model). In fact, they proved the stronger statement that the stable colouring is discrete on almost all graphs, that is, every vertex gets its own colour. On the other hand, colour refinement fails to distinguish any two regular graphs with the same number of vertices, such as a 6-cycle and the disjoint union of two triangles.

Colour refinement is not only useful as a simple isomorphism test in itself, but also as a subroutine for more sophisticated algorithms, both in theory and practice. For example, Babai and Luks's [1,3] $O(2^{\sqrt{n \log n}})$-algorithm — this is still the best known worst-case running time for isomorphism testing — uses colour refinement as a subroutine, and most practical graph isomorphism tools (for example, [9,6,8,12]), starting with McKay's "Nauty" [9,10], are based on the *individualisation refinement* paradigm. The basic idea of these algorithms is to recursively compute a *canonical labelling* of a given graph, which may already have an initial colouring of its vertices, as follows. We run colour refinement starting from the initial colouring until a stable colouring is reached. If the stable colouring is discrete, then this already gives us a canonical labelling (provided the colours assigned by colour refinement are canonical, see below). If not, we pick some colour c with more than one vertex. Then for each vertex v of colour c, we modify the stable colouring by assigning a fresh colour to v (that is, we "individualise" v) and recursively call the algorithm on the resulting vertex-coloured graph. Then for each v we get a canonically labelled version of our graph, and we return the lexicographically smallest among these. (More precisely, each canonical labelling of a graph yields a canonical string encoding, and we compare these strings lexicographically.) To turn this simple procedure into a practically useful algorithm, various heuristics are applied to prune the search tree. They exploit automorphisms of the graph found during the search. However, crucial for any implementation of such an algorithm is a very efficient colour refinement procedure, because colour refinement is called at every node of the search tree.

Colour refinement can be implemented to run in time $O((n+m)\log n)$, where n is the number of vertices and m the number of edges of the input graph. To our knowledge, this was first been proved by Cardon and Crochemore [5]. Later Paige and Tarjan [11, p.982] sketched a simpler algorithm. Both algorithms are based on the partitioning techniques introduced by Hopcroft [7] for minimising finite automata. However, an issue that is completely neglected in the literature is that, at least for individualisation refinement, we need a version of colour refinement that produces a *canonical colouring*. That is, if f is an isomorphism from a graph G to a graph H, then for all vertices v of G, v and $f(v)$ should get the same colour in the respective stable colourings of G and H. However, neither of the algorithms analysed in the literature seem to produce canonical colourings. Very briefly, the reason is that these algorithms use bucketing techniques for indexing vectors with an initial segment of the natural numbers that make sure that different vectors get different indices, but do not assign indices in the lexicographical or any other canonical order. This issue can be resolved by sorting the vectors lexicographically, but this causes a logarithmic overhead in the running time. We resolve the issue differently and avoid the logarithmic overhead, thus obtaining an implementation of colour refinement that computes a canonical stable colouring in time $O((n+m)\log n)$. Ignoring the canonical part, our algorithmic techniques are similar to known results: like [11] and various other papers, we use Hopcroft's strategy of 'ignoring the largest new cell', after splitting a cell [7]. Our data structures are similar to those described by Junttila and Kaski [8]. Nevertheless, since [8] contains no complexity analysis, and [11] omits various (nontrivial) implementation details, it seems that the current paper gives the first detailed description of an $O((m+n)\log n)$ algorithm that uses this strategy. On a high level, our algorithm is also quite similar to McKay's *canonical* colour refinement algorithm [9, Alg. 2.5], but with a few key differences which enable an $O((n + m)\log n)$ implementation. McKay [9] gave an $O(n^2 \log n)$ implementation using adjacency matrices.

Now the question arises whether colour refinement can be implemented in linear time. After various attempts, we started to believe that it cannot. Of course with currently known techniques one cannot expect to disprove the existence of a linear time algorithm for the standard (RAM) computation model, or for similar general computation models. Instead, we prove the complexity lower bound for a broad class of partition-refinement based algorithms, which captures all known colour refinement algorithms, and actually every reasonable algorithmic strategy we could think of. This model can alternatively be viewed as a "proof system". We use the following assumptions. (See Sections 2 and 4 for precise definitions.) Colour refinement algorithms start with a unit partition (which has one *cell* $V(G)$), and iteratively refine this until a stable colouring is obtained. This is done using *refining operations*: choose a union of current partition cells as *refining set* R, and choose another (possibly overlapping) union of partition cells S. Cells in S are split up if their neighbourhoods in R provide a reason for this. (That is, two vertices in a cell in S remain in the same cell only if they have the same number of neighbours in every cell in R.) This operation requires considering all edges between R and S, so the number of such edges is a very reasonable and modest lower bound for the complexity of such a refining step; we call this the *cost* of the operation. We note that a naive algorithm might choose $R = S = V(G)$ in every iteration. This then requires time $\Omega(mn)$ on graphs that require a linear number of refining operations, such as paths. Therefore, all fast algorithms are based on choosing R and S smartly (and on implementing refining steps efficiently).

For our main lower bound result, we construct a class of instances such that any possible sequence of refining operations that yields the stable partition has total cost at least $\Omega((m + n)\log n)$. Note that it is surprising that a tight lower bound can be obtained in this model. Indeed, cost *upper* bounds in this model would not necessarily yield corresponding algorithms, since firstly we allow the sets R and S to be chosen nondeterministically, and secondly, it is not even clear how to refine S using R in time proportional to the number of edges between these classes. However, as we prove a lower bound, this makes our result only stronger. We formulate the lower bound result for undirected graphs and non-canonical colour refinement, so that it also holds for digraphs, and canonical colour refinement. Our proof also implies a corresponding lower bound for the coarsest relational partitioning problem considered by Paige and Tarjan [11]. Because of space constraints, some details have been omitted.

2 Preliminaries

For an undirected (simple) graph G, $N(v)$ denotes the set of neighbours of $v \in V(G)$, and $d(v) = |N(v)|$ its degree. For a digraph, $N^+(v)$ and $N^-(v)$ denote the out- and in-neighbourhoods, and $d^+(v) = |N^+(v)|$ resp. $d^-(v) = |N^-(v)|$

the out- and in-degree, respectively. A partition π of a set V is a set $\{S_1, \ldots, S_k\}$ of pairwise disjoint nonempty subsets of V, such that $\cup_{i=1}^{k} S_i = V$. The sets S_i are called *cells* of π. The *order* of π is the number of cells $|\pi|$. A partition π is *discrete* if every cell has size 1, and *unit* if it has exactly one cell. Given a partition π of V, and two elements $u, v \in V$, we write $u \approx_\pi v$ if and only if there exists a cell $S \in \pi$ with $u, v \in S$. We say that a set $V' \subseteq V$ is π*-closed* if it is the union of a number of cells of π. In other words, if $u \approx_\pi v$ and $u \in V'$ then $v \in V'$. For any subset $V' \subseteq V$, π *induces* a partition $\pi[V']$ on V', which is defined by $u \approx_{\pi[V']} v$ if and only if $u \approx_\pi v$, for all $u, v \in V'$.

Let $G = (V, E)$ be a graph. A partition π of V is *stable* for G if for every pair of vertices $u, v \in V$ with $u \approx_\pi v$ and $R \in \pi$, it holds that $|N(u) \cap R| = |N(v) \cap R|$. If G is a digraph, then $|N^+(u) \cap R| = |N^+(v) \cap R|$ should hold. For readability, all further definitions and propositions in this section are formulated for (undirected) graphs, but the corresponding statements also hold for digraphs (replace degrees/neighbourhoods by out-degrees/out-neighbourhoods). One can see that if π is stable and $d(u) \neq d(v)$, then $u \not\approx_\pi v$, which we will use throughout.

A partition ρ of V *refines* a partition π of a subset S of V if for every $u, v \in S$, $u \approx_\rho v$ implies $u \approx_\pi v$. (Usually we take $S = V$.) If ρ refines π, we write $\pi \preceq \rho$. If in addition $\rho \neq \pi$, then we also write $\pi \prec \rho$. Note that $\preceq$ is a partial order on all partitions of V.

Definition 1. *Let G be a graph, and let π and π' be partitions of $V(G)$. For vertex sets $R, S \subseteq V(G)$ that are π-closed, we say that π' is obtained from π by a* refining operation (R, S) *if*

- *for every $S' \in \pi$ with $S' \cap S = \emptyset$, it holds that $S' \in \pi'$, and*
- *for every $u, v \in S$: $u \approx_{\pi'} v$ if and only if $u \approx_\pi v$ and for all $R' \in \pi$ with $R' \subseteq R$, $|N(u) \cap R'| = |N(v) \cap R'|$ holds.*

Note that if π' is obtained from π by a refining operation (R, S), then $\pi \preceq \pi'$. We say that the operation (R, S) is *effective* if $\pi \prec \pi'$. In this case, at least one cell $C \in \pi$ is *split*, which means that $C \notin \pi'$. Note that an effective refining operation exists for π if and only if π is unstable. In addition, the next proposition says that if the goal is to obtain a (coarsest) stable partition, then applying any refining operation is safe.

Proposition 2 (*). [1] *Let π' be obtained from π by a refining operation (R, S). If ρ is a stable partition with $\pi \preceq \rho$, then $\pi \preceq \pi' \preceq \rho$.*

A partition π is a *coarsest* partition for a property P if π satisfies P, and there is no partition ρ with $\rho \prec \pi$ that also satisfies property P.

Proposition 3 (*). *Let $G = (V, E)$ be a graph. For every partition π of V, there is a unique coarsest stable partition ρ that refines π.*

[1] In the full version, (detailed) proofs are given for statements marked with a star.

3 A Fast Canonical Color Refinement Algorithm

Consider a method for obtaining a sequence $S_1^G, \dots, S_{k(G)}^G$ of subsets of $V(G)$, for any (di)graph G. This method is called *canonical* if for any two isomorphic (di)graphs G and G', and any isomorphism $h : V(G) \to V(G')$, it holds that $k(G) = k(G')$, and $u \in S_i^G$ implies $h(u) \in S_i^{G'}$, for any $u \in V(G)$ and $i \in \{1, \dots, k(G)\}$. In a slight abuse of terminology, we also call the sequence canonical, if the method for obtaining it is clear from the context. For instance, for simple undirected graphs G, if we define D_d to be the set of vertices of degree d, for $d \in \{0, \dots, n-1\}$, $n = |V(G)|$, then $D_0, \dots, D_{n-1}$ is a canonical sequence, because every isomorphism maps vertices to vertices of the same degree. (In other words: degrees are *isomorphism invariant.*) In this section we give a fast algorithm for obtaining a canonical coarsest stable partition of $V(G)$, for any digraph G. This is an *ordered partition* of V, which is a sequence of sets $C_1, \dots, C_k$ such that $\{C_1, \dots, C_k\}$ is a partition of V. To obtain the most general result, we formulate the algorithm for digraphs.

High-level Description and Correctness Proofs The input to our algorithm is a digraph $G = (V, E)$, with $V = \{1, \dots, n\}$. For every vertex $v \in V$, the sets of out-neighbours $N^+(v)$ and in-neighbours $N^-(v)$ are given. Throughout, the algorithm maintains an ordered partition $\pi = C_1, \dots, C_k$ of V, starting with the unit partition. This partition is iteratively refined using operations of the form (R, V), where $R = C_r$ for some $r \in \{1, \dots, k\}$. We will show that when the algorithm terminates, no effective refining operations are possible on the resulting partition. So the resulting partition is the unique coarsest stable partition of G.

We now explain how to maintain a canonical order for the partition $\pi = C_1, \dots, C_k$. To this end, indices $i \in \{1, \dots, k\}$ are called *colours*, and the cells C_i are also called *colour classes* of the current partition. The partition π is then also viewed as a *colouring* of the vertices with colours $1, \dots, k$. To canonically choose the next *refining colour* r, we maintain a canonical sequence (stack) S_{refine} of colours that should still be used as refining colour. When a new refining colour r should be chosen, we select r to be the last colour added to S_{refine} (i.e. r is popped from the stack). For a given refining colour class $R = C_r$ and any $x \in V$, call $d_r^+(x) := |N^+(x) \cap R|$ the *colour degree* of x. Then every colour $s \in \{1, \dots, k\}$ will be split up according to colour degrees. More precisely, for a given refining colour r, we partition every cell C_s into new cells $C_{\sigma_1}, \dots, C_{\sigma_p}$, such that for $x \in C_{\sigma_i}$ and $y \in C_{\sigma_j}$: (i) $i = j$ if and only if $d_r^+(x) = d_r^+(y)$, and (ii) if $i < j$ then $d_r^+(x) < d_r^+(y)$. In other words, the new colours are ordered canonically according to their colour degrees. Since we wish to have nonempty sets in our partition, we choose $\sigma_1 = s$, and $\sigma_i = k + i - 1$ for all $2 \le i \le p$, and then update the number of colours k. To obtain a canonical colouring, it is also important to split up the colours $s \in \{1, \dots, k\}$ in increasing order.

It remains to explain how newly introduced colours are added to the stack S_{refine} in a canonical way. Initially, S_{refine} contains colour 1, and whenever new colours are introduced during the splitting of a colour class C_s, these are pushed onto the stack S_{refine}, in increasing order. There is however one exception: if we have already used the vertex set $S = C_s$ as refining colour class before, and this set is split up into new colours $C_{\sigma_1}, \dots, C_{\sigma_p}$, then it is not necessary to use all of these new colours as refining colours later. Indeed, for every $i \in \{1, \dots, p\}$ and every $x, y \in V(G)$, if $|N^+(x) \cap S| = |N^+(y) \cap S|$ and $|N^+(x) \cap C_j| = |N^+(y) \cap C_j|$ holds for every $j \neq i$, then it also holds that $|N^+(x) \cap C_i| = |N^+(y) \cap C_i|$, since $\{C_{\sigma_1}, \dots, C_{\sigma_p}\}$ is a partition of S. Hence we may select an $i \in \{1, \dots, p\}$, and only add the colours $\{1, \dots, p\} \setminus \{i\}$ to the stack S_{refine}. To obtain a good complexity, we choose i such that $|C_{\sigma_i}|$ is maximised, in order to minimise the sizes of the refining colour sets used later during the computation. (This is Hopcroft's trick [7].) To be precise, for a given s, we canonically choose b to be the minimum colour degree that maximises $|\{x \in C_s \mid d_r^+(x) = b\}|$, and add all newly introduced colours to the stack, in increasing order, except the new colour that corresponds to b. On the other hand, if s was already on the stack S_{refine}, then this argument does not apply, so we have to add every new colour to the stack. The algorithm terminates when the stack S_{refine} is empty, and returns the final ordered partition $C_1, \dots, C_k$.

Lemma 4. *For any digraph G, the above algorithm computes a canonical sequence $C_1, \dots, C_k$, such that $\{C_1, \dots, C_k\}$ is the coarsest stable partition of G.*

Proof sketch: The resulting partition $\pi = \{C_1, \dots, C_k\}$ is refined by the coarsest stable partition ω of G because it is obtained from the unit partition by using refining operations (Proposition 2). It is then equal to ω since it is stable. This follows since using the argument given above, one can verify that the following invariant is maintained: if there exist colour classes C_r and C_s such that the refining operation (C_r, C_s) is effective, then the stack S_{refine} contains a colour r' such that the refining operation $(C_{r'}, C_s)$ is effective. Since the stack S_{refine} is empty when the algorithm terminates, stability follows. The final sequence is canonical since at every point during the computation, both the stack S_{refine} and the current ordered partition $C_1, \dots, C_k$ are canonical sequences. This holds because informally, the new colours that we assign to vertices, and the order in which new colours are added to the stack, are completely determined by isomorphism-invariant values such as colour degrees with respect to sets from a canonical sequence. □

Implementation and Complexity Bound. We now describe a fast implementation of the aforementioned algorithm. The main idea of the complexity proof is the following: one *iteration* consists of popping a refining colour r from the stack S_{refine}, and applying the refining operation (R, V), with $R = C_r$. Below we show that one such iteration takes time $O(|R| + D^-(R) + k_i \log k_i)$, where $D^-(R) = \sum_{v \in R} d^-(v)$ and k_i is the number of new colours that are introduced during iteration i. Next, we observe that for every vertex $v \in V(G)$, if $R_1^v, \dots, R_q^v$ are the refining colour classes C_r with $v \in C_r$ that are considered throughout the computation, in chronological order, then for all $i \in \{1, \dots, q-1\}$, $|R_i^v| \geq 2|R_{i+1}^v|$ holds. This holds because whenever a set $S = C_s$ is split up into $C_{\sigma_1}, \dots, C_{\sigma_p}$, where S has been considered earlier as a refining colour (so it is not in S_{refine} anymore), then for all new colours σ_i that are added to the stack

S_{refine}, $|C_{\sigma_i}| \leq \frac{1}{2}|S|$ holds (since the largest colour class is not added to S_{refine}). Note that if a colour class C_{σ_i} is subsequently split up before it is considered as refining colour, the bound of course also holds. It follows that every $v \in V(G)$ appears at most $\log_2 n$ times in a refining colour class. Then we can write

$$\sum_R |R| + D^-(R) \leq \sum_{v \in V(G)} (1 + d^-(v)) \log_2 n = (n+m) \log_2 n,$$

with $m = |E(G)|$, where the first summation is over all refining colour classes $R = C_r$ considered during the computation. In addition, the total number of new colours that is introduced is at most n, since every colour class, after it is introduced, remains nonempty throughout the computation. So we may write $\sum_i k_i \log k_i \leq \sum_i k_i \log n \leq n \log n$. Combining these facts shows that the total complexity of the algorithm can be bounded by $O((n+m)\log n) + O(n \log n) = O((n+m)\log n)$.

It remains to describe an implementation such that the complexity of one iteration i of the while-loop, where refining colour class $R = C_r$ is considered, can be bounded by $O(|R| + D^-(R) + k_i \log k_i)$. The colour classes C_i are represented by doubly linked lists. For all lists, we maintain the length.

The first challenge is how to compute the colour degrees $d_r^+(v)$ efficiently for every $v \in V(G)$, with respect to the refining colour r. For this we use an array cdeg$[v]$, indexed by $v \in \{1, \ldots, n\}$. We use the following invariant: at the beginning of every iteration, cdeg$[v] = 0$ for all v. Then we can compute these colour degrees by looping over all in-neighbours w of all vertices $v \in R$, and increasing cdeg$[w]$. At the same time, we compute a list C_{adj} of colours i that contain at least one vertex $w \in C_i$ with cdeg$[w] \geq 1$, and for every such colour i, we compute a list A_i of all vertices w with cdeg$[w] \geq 1$. None of these lists contain duplicates. This can all be done in time $O(|R| + D^-(R))$, assuming that at the beginning of every iteration, every A_i is an empty list, C_{adj} is an empty list, and flags are maintained for colours to keep track of membership in C_{adj}. With the same complexity, we can reset all of these data structures at the end of every iteration.

Next, we address how we can consider all colours that split up in one iteration, in canonical (increasing) order. To this end, we compute a new list C_{split}, which represents the subset of C_{adj} containing all colours that actually split up. By ensuring that all colours in C_{split} split up, we have that $|C_{\text{split}}| \leq k_i$, and therefore we can afford to sort this list. This can be done using any list sorting algorithm of complexity $O(k_i \log k_i)$, such as merge sort. To compute which colours split up, we compute for every colour in $i \in C_{\text{adj}}$ the maximum colour degree maxcdeg$[i]$ and minimum colour degree mincdeg$[i]$. The value maxcdeg$[i]$ can easily be computed while computing the colour degrees. We have mincdeg$[i] = 0$ if $|A_i| < |C_i|$. Otherwise, we can afford to compute mincdeg$[i]$ by iterating over $A_i = C_i$.

Finally, we need to show how a single colour class $S = C_s$ can be split up, and how the appropriate new colours can be added to the stack S_{refine} in the proper order, all in time $O(D_R^+(S))$. Here $R = C_r$ denotes the refining colour class, and

$D^+_R(S) = \sum_{v \in S} |N^+(v) \cap R|$. Note that when summing over all $s \in C_{\text{split}}$, this indeed gives a total complexity of at most $O(D^-(R))$. Firstly, for every relevant d, we compute how many vertices in C_s have colour degree d. These values are stored in an array numcdeg$[d]$, indexed by $d \in \{0, \ldots, \text{maxcdeg}[s]\}$. (Note that maxcdeg$[s] \leq D^+_R(S)$, so we can afford to initialise an array of this size.) Using this array numcdeg, we can easily compute the (minimum) colour degree b that occurs most often in S, which corresponds to the new colour that is possibly not added to S_{refine}. Using numcdeg, we can also easily construct an array f_{newcol}, indexed by $d \in \{0, \ldots, \text{maxcdeg}[s]\}$, which represents the mapping from colour degrees that occur in S to newly introduced colours, or to the current colour s. Finally, we can loop over A_s in time $O(D^+_R(S))$, and move all vertices $v \in A_s$ from C_s to C_i, where $i = f_{\text{newcol}}[\text{cdeg}[v]]$ is the new colour that corresponds to the colour degree of v. With a proper implementation using doubly linked lists, this can be done in constant time for a single vertex. Note that looping over A_s suffices, because if there are vertices in C_s with colour degree 0, then these keep the same colour, and thus do not need to be addressed. This fact is essential since the number of such vertices may not be bounded by $O(D^+_R(S))$.

This shows how the algorithm can be implemented such that one iteration takes time $O(|R| + D^-(R) + k_i \log k_i)$. Combined with the above analysis, this shows that the algorithm terminates in time $O((n+m)\log n)$. So with Lemma 4, we obtain:

Theorem 5. *For any digraph G on n vertices with m edges, in time $O((n + m)\log n)$ a canonical coarsest stable partition can be computed.*

In individualisation refinement algorithms, one branch is as follows [9,6,8,12,10]: whenever a stable but non-discrete colouring is obtained, some new vertex v is 'individualised' by assigning it a new unique colour, v is added to S_{refine}, and the process continues. Observe that the $O((n + m)\log n)$ bound holds for this entire process.

4 Complexity Lower Bound

The *cost* of a refining operation (R, S) is $\text{cost}(R, S) := |\{(u, v) \mid u \in R, v \in S\}|$. This is basically the number of edges between R and S, except that edges with both ends in $R \cap S$ are counted twice.

Definition 6. *Let $G = (V, E)$ be a graph, and π be a partition of V.*

- *If π is stable, then cost$(\pi) := 0$.*
- *Otherwise, cost$(\pi) := \min_{R,S}$ cost$(\pi(R, S))$ + cost(R, S), where the minimum is taken over all effective refining operations (R, S) that can be applied to π, and where $\pi(R, S)$ denotes the partition resulting from the operation (R, S).*

A refining operation (R, S) on π is *elementary* if both $R \in \pi$ and $S \in \pi$. The following observation is useful: since non-elementary refining steps can be split up into elementary refining steps with the same total cost, we may also take the minimum over all effective *elementary* refining operations.

We can now formulate the main result of this section.

Theorem 7. *For every integer $k \geq 2$, there is a graph G_k with $n \in O(2^k k)$ vertices and $m \in O(2^k k^2)$ edges, such that cost$(\alpha) \in \Omega((m+n)\log n)$, where α is the unit partition of $V(G_k)$.*

Note that this theorem implies a complexity lower bound for all partition-refinement based algorithms for colour refinement, as discussed in the introduction. We use the following key observation to prove the theorem. For a partition π of V, denote by π_∞ the coarsest stable partition of V that refines π.

Proposition 8 (*). *Let π and ρ be partitions of V such that $\pi \preceq \rho \preceq \pi_\infty$. Then cost$(\pi) \geq$ cost(ρ).*

We say that a partition π of V *distinguishes* two sets $V_1 \subseteq V$ and $V_2 \subseteq V$ if there is a set $R \in \pi$ with $|R \cap V_1| \neq |R \cap V_2|$. This is used often for the case where $V_1 = N(u)$ and $V_2 = N(v)$ for two vertices u and v, to conclude that if π is stable, then $u \not\approx_\pi v$. If $V_1 = \{x\}$ and $V_2 = \{y\}$, then we also say that π *distinguishes x from y.*

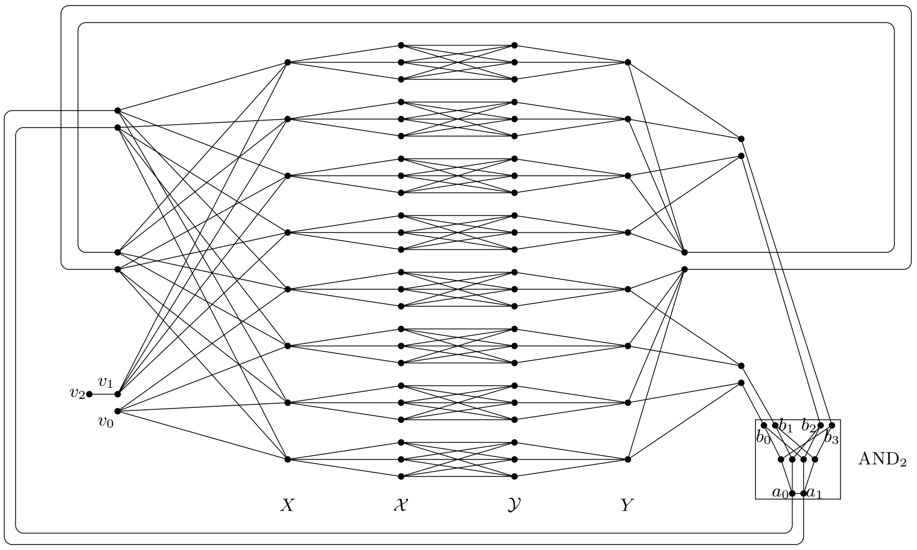

Fig. 1. The Graph G_3

Construction of the Graph For $k \in \mathbb{N}$, denote $\mathcal{B}_k = \{0, \ldots, 2^k - 1\}$. For $\ell \in \{0, \ldots, k\}$ and $q \in \{0, \ldots, 2^\ell - 1\}$, the subset $\mathcal{B}^\ell_q = \{q2^{k-\ell}, \ldots, (q+1)2^{k-\ell} - 1\}$ is called the *q-th binary block of level ℓ*. Analogously, for any set of vertices with indices in $\mathcal{B}_k$, we also consider binary blocks. For instance, if $X = \{x_i \mid i \in \mathcal{B}_k\}$, then $X^\ell_q = \{x_i \mid i \in \mathcal{B}^\ell_q\}$ is called a binary block of X. For such a set X, a partition π of X *into binary blocks* is a partition where every $S \in \pi$ is a binary block. A key fact for binary blocks that we will often use is that for any ℓ and q, $\mathcal{B}^\ell_q = \mathcal{B}^{\ell+1}_{2q} \cup \mathcal{B}^{\ell+1}_{2q+1}$.

For every integer $k \geq 2$, we will construct a graph G_k. In its core this graph consists of the vertex sets $X = \{x_i \mid i \in \mathcal{B}_k\}$, $\mathcal{X} = \{x^j_i \mid i \in \mathcal{B}_k, j \in [k]\}$, $\mathcal{Y} = \{y^j_i \mid i \in \mathcal{B}_k, j \in [k]\}$ and $Y = \{y_i \mid i \in \mathcal{B}_k\}$. Every vertex x_i is adjacent to x^j_i for all $j \in [k]$ and every y_i is adjacent to all y^j_i. Furthermore, for all i, j_1, j_2 there is an edge between $x^{j_1}_i$ and $y^{j_2}_i$. (For $\mathcal{X}$, *binary blocks* are subsets of the form $\mathcal{X}^\ell_q := \{x^j_i \mid i \in \mathcal{B}^\ell_q, j \in [k]\}$, and for $\mathcal{Y}$ the definition is analogous.)

We add gadgets to the graph to ensure that any sequence of refining operations behaves as follows. After the first step, which distinguishes vertices according to their degrees, X and Y are cells of the resulting partition. Next, X splits up into two binary blocks X^1_0 and X^1_1 of equal size. This causes $\mathcal{X}$ to split up accordingly into $\mathcal{X}^1_0$ and $\mathcal{X}^1_1$. One of these cells will be used to halve $\mathcal{Y}$ in the same way. This refining operation (R, S) is expensive because $[R, S]$ contains half of the edges between $\mathcal{X}$ and $\mathcal{Y}$. Next, Y can be split up into Y^1_0 and Y^1_1. Once this happens, there is a gadget AND$_1$ that causes the two cells X^1_0, X^1_1 to split up into the four cells X^2_q, for $q = 0, \ldots, 3$. Again, this causes cells in $\mathcal{X}, \mathcal{Y}$ and Y to split up in the same way and to achieve this, half of the edges between $\mathcal{X}$ and $\mathcal{Y}$ have to be considered. The next gadget AND$_2$ ensures that if both cells of Y are split, then the four cells of X can be halved again, etc. In general, we design a gadget AND$_\ell$ of level ℓ that ensures that if Y is partitioned into $2^{\ell+1}$ binary blocks of equal size, then X can be partitioned into $2^{\ell+2}$ binary blocks of equal size. By halving all the cells classes of X and Y $k = \Theta(\log n)$ times (with $n = |V(G_k)|$), this refinement process ends up with a discrete colouring of these vertices. Since every iteration uses half of the edges between $\mathcal{X}$ and $\mathcal{Y}$ (which are $\Theta(m)$), we get the cost lower bound of $\Omega(m \log n)$ (with $m = |E(G_k)|$).

We now define these gadgets in more detail. For every integer $\ell \geq 1$, we define a gadget AND$_\ell$, which consists of a graph G together with two *out-terminals* a_0, a_1, and an ordered sequence of $p = 2^\ell$ *in-terminals* $b_0, \ldots, b_{p-1}$. For $\ell = 1$, the graph G has $V(G) = \{a_0, a_1, b_0, b_1\}$, and $E(G) = \{a_0b_0, a_1b_1\}$. For $\ell = 2$, the graph G is identical to the construction of Cai, Fürer and Immerman [4], but with an edge a_0a_1 added (see Figure 1). The out-terminals a_0, a_1 and in-terminals $b_0, \ldots, b_3$ are indicated. For $\ell \geq 3$, AND$_\ell$ is obtained by taking one copy G^* of an AND$_2$-gadget, and two copies G' and G'' of an AND$_{\ell-1}$-gadget, and adding four edges to connect the two pairs of in-terminals of G^* with the pairs of out-terminals of G' and G'', respectively. As out-terminals of the resulting gadget we choose the out-terminals of G^*. The in-terminal sequence is obtained by concatenating the sequences of in-terminals of G' and G''. For any AND$_\ell$-gadget G with in-terminals $b_0, \ldots, b_{2^\ell-1}$, the *in-terminal pairs* are pairs b_{2p} and b_{2p+1}, for all $p \in \{0, \ldots, 2^{\ell-1} - 1\}$. We now state the key property for *AND*$_\ell$-gadgets, which can be verified for $\ell = 2$, and then follows inductively for $\ell \geq 3$. We say that ρ *agrees with* ψ if $\rho[S] = \psi$.

Lemma 9 (*). *Let G be an AND$_\ell$-gadget with in-terminals $B = \{b_0, \dots, b_{2^\ell-1}\}$ and out-terminals a_0, a_1. For any partition ψ of B into binary blocks, the coarsest stable partition ρ of $V(G)$ that refines ψ agrees with ψ. Furthermore, ρ distinguishes a_0 from a_1 if and only if ψ distinguishes all in-terminal pairs.*

The graph G_k is now constructed as follows. Start with vertex sets $X, \mathcal{X}, \mathcal{Y}$ and Y, and edges between them, as defined above. For every $\ell \in \{1, \dots, k-1\}$, we add a copy G of an AND$_\ell$-gadget to the graph. Denote the in- and out-terminals of G by a_0, a_1 and $b_0, \dots, b_{2^\ell-1}$, respectively.

- For $i = 0, 1$ and all relevant q: we add edges from a_i to every vertex in $X^{\ell+1}_{2q+i}$.
- For every i, we add edges from b_i to every vertex in Y^ℓ_i.

Finally, we add a *starting gadget* to the graph, consisting of three vertices v_0, v_1, v_2, the edge v_1v_2, and edges $\{v_0x_i \mid i \in \mathcal{B}^1_0\} \cup \{v_1x_i \mid i \in \mathcal{B}^1_1\}$. See Figure 1 for an example of this construction. (In the figure, we have expanded the terminals of AND$_2$ into edges, for readability. This does not affect the behavior of the graph.)

Cost Lower Bound Proof Intuitively, at level ℓ of the refinement process, the current partition contains all blocks $\mathcal{X}^{\ell+1}_q$ of level $\ell+1$ and for all $0 \le q < 2^\ell$, either $\mathcal{Y}^\ell_q$ or the two blocks $\mathcal{Y}^{\ell+1}_{2q}$ and $\mathcal{Y}^{\ell+1}_{2q+1}$. In this situation one can split up the blocks $\mathcal{Y}^\ell_q$ into blocks $\mathcal{Y}^{\ell+1}_{2q}$ and $\mathcal{Y}^{\ell+1}_{2q+1}$ using either refinement operation $(\mathcal{X}^{\ell+1}_{2q}, \mathcal{Y}^\ell_q)$ or $(\mathcal{X}^{\ell+1}_{2q+1}, \mathcal{Y}^\ell_q)$. These operations both have cost $2^{k-(\ell+1)}k^2$, and refining all the $\mathcal{Y}^\ell_q$ cells in this way costs $2^{k-1}k^2$. Once $\mathcal{Y}$ is partitioned into binary blocks of level $\ell+1$, we can partition $\mathcal{X}$ into blocks of level $\ell+2$ (using the AND$_\ell$-gadget), and proceed the same way. Since there are k such refinement levels, we can lower bound the total cost of refining the graph by $2^{k-1}k^3 = \Omega(m \log n)$ and are done. What remains to show is that applying the refinement operations in this specific way is the only way to obtain a stable partition. To formalise this, we introduce a number of partitions of $V(G_k)$ that are stable with respect to the (spanning) subgraph $G'_k = G_k - [\mathcal{X}, \mathcal{Y}]$, and that partition $\mathcal{X}$ and $\mathcal{Y}$ into binary blocks. (For disjoint vertex sets S,T, we denote $[S, T] = \{uv \in E(G) \mid u \in S, v \in T\}$.) So on G_k, these partitions can only be refined using operations (R, S), where R is a binary block of $\mathcal{X}$ and S is a binary block of $\mathcal{Y}$.

Definition 10. *For any $\ell \in \{0, \dots, k-1\}$, and nonempty set $Q \subseteq \mathcal{B}_\ell$, by $\tau_{Q,\ell}$ we denote the partition of $\mathcal{X} \cup \mathcal{Y}$ that contains cells*

- *$\mathcal{X}^{\ell+1}_q$ for all $q \in \mathcal{B}_{\ell+1}$,*
- *$\mathcal{Y}^\ell_q$ for all $q \in Q$, and both $\mathcal{Y}^{\ell+1}_{2q}$ and $\mathcal{Y}^{\ell+1}_{2q+1}$ for all $q \in \mathcal{B}_\ell \setminus Q$.*

$\pi_{Q,\ell}$ denotes the coarsest stable partition for $G'_k = G_k - [\mathcal{X}, \mathcal{Y}]$ that refines $\tau_{Q,\ell}$.

Since $\pi_{Q,\ell}$ is stable on G'_k, any effective refining operation (with respect to G_k) should involve the edges between $\mathcal{X}$ and $\mathcal{Y}$. Using Lemma 9, it can be shown that $\pi_{Q,\ell}$ agrees with $\tau_{Q,\ell}$, and therefore any effective *elementary* refining operation has the following form:

Lemma 11 (*). *Let (R,S) be an effective elementary refining operation on $\pi_{Q,\ell}$. Then for some $q \in Q$, $R = \mathcal{X}_{2q}^{\ell+1}$ or $R = \mathcal{X}_{2q+1}^{\ell+1}$, and $S = \mathcal{Y}_q^{\ell}$. The cost of this operation is $k^2 2^{k-(\ell+1)}$.*

This motivates the following definition: for $q \in Q$, by $r_q(\pi_{Q,\ell})$ we denote the partition of $V(G_k)$ that results from (either of) the above refining operation(s).

Proof sketch of Theorem 7: Let G_k be the graph described above and $\pi_{Q,\ell}$ be the partitions of $V(G_k)$ from Definition 10. First, we note that using Lemma 9, it can be shown that a partition is not stable until it is discrete on $X \cup Y$. So the coarsest stable partition ω of G refines all partitions $\pi_{Q,\ell}$. For ease of notation, we define $\pi_{\emptyset,\ell} := \pi_{\mathcal{B}_{\ell+1},\ell+1}$. By Lemma 11, any effective elementary refinement operation on a partition $\pi_{Q,\ell}$ has cost $2^{k-(\ell+1)}k^2$, and results in $r_q(\pi_{Q,\ell})$ for some $q \in Q$. Denote $Q' = Q \setminus \{q\}$. Note that $r_q(\pi_{Q,\ell})$ agrees with $\tau_{Q',\ell}$ on $\mathcal{X} \cup \mathcal{Y}$. It can actually be shown that $r_q(\pi Q, \ell) \preceq \pi_{Q',\ell}$. So we may now apply Proposition 8 to conclude that $\mathrm{cost}(\pi_{Q,\ell}) \geq 2^{k-(\ell+1)}k^2 + \min_{q \in Q} \mathrm{cost}(\pi_{Q\setminus\{q\},\ell})$. By induction on $|Q|$ it then follows that $\mathrm{cost}(\pi_{\mathcal{B}_\ell,\ell}) \geq 2^{k-1}k^2 + \mathrm{cost}(\pi_{\mathcal{B}_{\ell+1},\ell+1})$ for all $0 \leq \ell \leq k-1$. Hence, by induction on ℓ, $\mathrm{cost}(\pi_{\mathcal{B}_0,0}) \geq 2^{k-1}k^3$, which lower bounds $\mathrm{cost}(\alpha)$. It can be verified that $n \in O(2^k k)$ and $m \in O(2^k k^2)$, so $\log n \in O(k)$. This shows that $\mathrm{cost}(\alpha) \in \Omega((m+n)\log n)$. □

References

1. Babai, L.: Moderately exponential bound for graph isomorphism. In: Gécseg, F. (ed.) FCT 1981. LNCS, vol. 117, pp. 34–50. Springer, Heidelberg (1981)
2. Babai, L., Erdös, P., Selkow, S.: Random graph isomorphism. SIAM Journal on Computing 9, 628–635 (1980)
3. Babai, L., Luks, E.: Canonical labeling of graphs. In: Proc. STOC 1983, pp. 171–183 (1983)
4. Cai, J., Fürer, M., Immerman, N.: An optimal lower bound on the number of variables for graph identifications. Combinatorica 12(4), 389–410 (1992)
5. Cardon, A., Crochemore, M.: Partitioning a graph in $O(|A| \log_2 |V|)$. Theoretical Computer Science 19(1), 85–98 (1982)
6. Darga, P., Liffiton, M., Sakallah, K., Markov, I.: Exploiting structure in symmetry detection for CNF. In: Proc. DAG 2004, pp. 530–534 (2004)
7. Hopcroft, J.: An n log n algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
8. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Proc. ALENEX 2007, pp. 135–149 (2007)
9. McKay, B.: Practical graph isomorphism. Congressus Numerantium 30, 45–87 (1981)
10. McKay, B.: Nauty users guide (version 2.4). Computer Science Dept., Australian National University (2007)
11. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM Journal on Computing 16(6), 973–989 (1987)
12. Piperno, A.: Search space contraction in canonical labeling of graphs. arXiv preprint arXiv:0804.4881v2 (2011)