Almost-symmetries of Graphs

Igor L. Markov

Department of EECS, The University of Michigan, Ann Arbor, MI 48109-2121
imarkov@eecs.umich.edu

Abstract. Many successful uses of symmetries in discrete computational problems rely on group-theoretical properties. Almost-symmetries can be more numerous, but do not have the same properties. In this work, we study almost-symmetries of graphs, proposing data structures for representing them, algorithms for finding them, and logic predicates for symmetry-breaking.

1 Introduction

A structure-preserving reversible transformation of a structured object is often called a symmetry (or *automorphism*), with prime examples being (1) a permutation of vertices of a given graph, that maps edges to edges and preserves vertex labels, (2) a permutation of variables in a system of constraints, that preserves the system, (3) a permutation of possible values of some variables, such as the simultaneous negation of two Boolean (or two integer) variables, (4) a permutation of input and output variables of a Boolean function that leaves the function invariant. Computational applications often involve finite domains. For example, identifying a pair of symmetric variables x and y in an equation allows one to introduce the additional constraint $x \leq y$ so as to reduce the amount of searching by up to 25%, or closer to 50% for non-Boolean variables. Combining many such symmetries sometimes reduces the complexity of search, proofs or refutations from exponential to polynomial, both in provable lower bounds [9] and empirical performance [1].

Identifying and using a greater variety of symmetries often improves computational efficiency. Hence, it is natural to relax the notion of symmetry and deal with *almost-symmetries* which retain useful properties, occur more often and may have greater impact. In this work we consider a class of almost-symmetries, their representation, discovery and usage. Our approach is to relax the notion of graph symmetries (automorphisms) to account for vertices whose colors may change and whose edges may appear or disappear. In particular, our algorithms can discover a small number edge additions or removals that can make a given graph more symmetric. We show that almost-symmetries can be compactly represented by unordered lists of unordered lists of permutations, from which almost-symmetry-breaking predicates can be constructed, analogous to the well-known technique for full-fledged symmetries.

2 Almost-symmetries and Their Properties

Many existing notions of symmetry can be and often are modeled by graph automorphisms, i.e., vertex permutations that map edges to edges. As a matter of convenience (but not generalization), vertices may bear colors, so that blue vertices cannot map into red vertices.

Almost-symmetries can be defined either as transformations that violate some conditions for being symmetries, or as symmetries of slightly modified objects [7]. For example, adding or removing one constraint can make the overall set of constraints more symmetric. In this work, we pursue the latter approach and consider two possibilities: (1) vertices with undefined color (or a set of possible colors), (2) edges that can be added or removed. Formally speaking, the second case can be reduced to the first by representing every edge and every non-edge by a pair of fake edges and a fake vertex of appropriate color. If an original edge is allowed to disappear, its fake vertex can assume either color. While this reduction can introduce gross inefficiency in practical algorithms, it is convenient to illustrate key concepts. Another reduction that does not require loss of generality is to assume that vertices of variable (chameleon) color can take on any specific color without further restrictions. To relax the original color-related limitation, a chameleon vertex can be mapped to a chameleon vertex or a vertex of any regular color. Additionally, a vertex of a regular color can be mapped to a chameleon vertex. Since almost-symmetries are permutations from S_n,

- products of almost-symmetries are unambiguous,
- the identity permutation is an almost-symmetry,
- each almost-symmetry has a unique inverse.

Unfortunately, almost-symmetries are not closed under the compositional product. For example, consider a 3-vertex graph with no edges, the permutation π_1 that swaps the blue vertex v_1 with the chameleon vertex v_2, and π_2 that swaps v_2 with the red vertex v_3. The product $\pi_1 \cdot \pi_2$ maps the blue vertex v_1 to the red vertex v_3, which is forbidden. We cannot fix all products, but we can require that all powers of an almost-symmetry be almost-symmetries.

We further restrict almost-symmetries to those automorphisms of the underlying unlabeled graph $\mathcal{G}^*$ with the property that each cycle contains vertices of no more than one regular color.

With their compositional product defined only partially, almost-symmetries do not form *groups*, *semi-groups*, *monoids* or *groupoids*. In particular, consider the products $(\pi_1 \cdot \pi_1) \cdot \pi_2$ and $\pi_1 \cdot (\pi_1 \cdot \pi_2)$. Since $\pi_1 \cdot \pi_1 = (\ )$, the former is defined $(= \pi_2)$, but $(\pi_1 \cdot \pi_2)$ is not. Almost-symmetries do not form *cosets* because any coset containing the identity permutation ( ) must be a subgroup.

Note that for any given color-based almost-symmetry of a graph, there is *a specialization of colors* for the graph's chameleon vertices that turns the almost-symmetry into a regular symmetry. Indeed, each cycle can include vertices at most one regular color, to which all chameleon vertices in this cycle can be

specialized. If *all* vertices in the cycle are chameleon-colored, specialize all of them to any existing color.

Now consider all possible color specializations of the chameleon vertices (partitions of the vertex set into as many cells as we have regular colors). In each case we obtain a regular labeled graph with a group of symmetries, and each almost-symmetry is contained in at least one of those groups. Thus

The set of almost-symmetries is a union of subgroups of S_n.

In an *irredundant* union-of-subgroups expression no subgroups can be skipped. Greedy removal of redundant subgroups from an expression ensures irredundancy, but not the smallest size. Even solving the (implicit) set-covering problem for given subgroups may not produce a union with fewest subgroups possible.[1]

Observe that all relevant subgroups are contained in the automorphism group $Aut(\mathcal{G}^*)$ of the unlabeled (colorless) graph $\mathcal{G}^*$ and contain all automorphisms of $\mathcal{G}^*$ whose cycles do not mix chameleon vertices with regular vertices. Such permutations form a subgroup that can be recovered in two steps: (1) construct the labeled graph $\mathcal{G}^\sharp$ by specializing all chameleon vertices to a color that has not been used before, (2) finding $Aut(\mathcal{G}^\sharp)$. The algebraic structure described above is visualized in Figure 1. In practice the intersection of subgroups can be larger than $Aut(\mathcal{G}^\sharp)$, e.g., consider two disconnected vertices v_1 (blue) and v_2 (chameleon), for which $Aut(\mathcal{G}^\sharp) = \{(\ )\}$ but the only non-trivial almost-symmetry (12) generates a larger subgroup.

Solutions to the graph almost-automorphism problem can be represented by subgroup generators arranged in potentially-overlapping unordered lists — one list per subgroup of S_n in the union-of-subgroups structure.

[1] Consider $\mathbb{Z}_2 \times \mathbb{Z}_2$, the symmetry group of the letter H represented by the union of its three two-element subgroups.

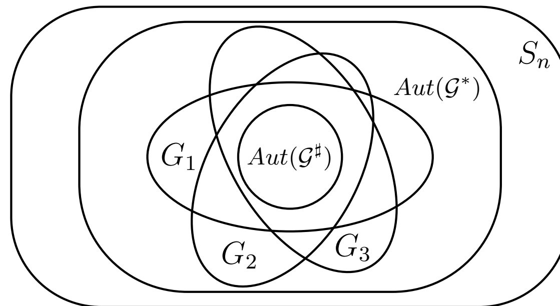

Fig. 1. Algebraic structure in almost-symmetries — a set-union of subgroups: $Aut(\mathcal{G}^\sharp) \subseteq G_1 \cup G_2 \cup G_3 \subseteq Aut(\mathcal{G}^*)$. All shapes represent groups or subgroups, and illustrate containment relations geometrically.

3 Finding Almost-Symmetries

To find almost-symmetries and represent them compactly by lists of lists of permutations, we extend a common graph-automorphism algorithm, used in solvers NAUTY [10] and SAUCY [8], to handle color-based almost-symmetries. Our goal here is to capture all almost-symmetries by a small set of colorings, i.e., vertex partitions. Fortunately, existing algorithms are based on partition refinement,

First, vertices are partitioned by degree. Further, an *immediate refinement* step is based on external adjacencies, e.g., two vertices in the same cell m_1 cannot be symmetric if one is adjacent to a vertex cell m_2 and the other is not adjacent to any vertex in m_2 [8, Figure 1]. After such refinements are exhausted Hopcroft's procedure, the algorithm resorts to (*traditional*) branching by, conceptually, picking a non-singleton cell and mapping its lowest-indexed vertex v_i to another vertex v_j [10, 8]. This may trigger another round of immediate partition refinement — neighbors of v_i can only map to neighbors of v_j. The overall algorithm proceeds by alternating between branching and refinement until all vertices in some cells are mapped to other vertices (or themselves), which allows one to test the resulting permutation for being a symmetry of the original graph [8, Figure 2]. Confirmed symmetries are accumulated,[2] and the algorithm backtracks to explore other branches of the search tree. With appropriate pruning [10], the algorithm ignores all branches leading only to symmetries expressible as compositions of accumulated symmetries. This ensures that group generators at the output are irredundant and can implicitly express an exponential number of symmetries in polynomial space.

We extend the above algorithm by interleaving traditional branching and partition refinement with branching on colors of chameleon vertices. To minimize the number of different subgroups in the union-of-subgroups structure, we delay such branching. The algorithm repeatedly applies prioritized rules:

1. Since all almost-symmetries of a given graph $\mathcal{G}$ are in $Aut(\mathcal{G}^*)$, apply an existing graph-automorphism algorithm to $\mathcal{G}^*$ (i.e., ignore vertex colors) until it needs branching or terminates.
2. Any cell with vertices of more than one regular color, but no chameleon vertices, must be split immediately. Cells containing chameleon vertices cannot be split based on internal vertex colors.
3. Apply partition refinement based on adjacencies to cells of regular colors.

4A. If a cell contains only chameleon vertices, then specialize all vertices to one arbitrary color.

4B. If a cell contains chameleon vertices and vertices of one regular color, then specialize all chameleon vertices to this regular color.

5. In a non-trivial cell without chameleon vertices, invoke traditional branching in the hope that some cells with chameleon vertices will be refined through

[2] For graphs in engineering applications it is relatively rare to reject potential symmetries at this stage, but such *bad leaves* are common for highly symmetric Cayley graphs. With no bad leaves, the algorithm runs in polynomial time.

adjacencies. This rule does not use traditional branching in cells with unspecified chameleon vertices due to difficulties with color assignment, implied in some, but not all branches.

6. Branch on chameleon vertices in cell j with the smallest branching factor:

– Specialize all k_j chameleon vertices in cell j at once — otherwise splitting and traditional branching will not work (Rules 2 and 5).
– Assign only c_j regular colors used in cell j.
– Select j to minimize branching factor $c_j^{k_j}$.

Symmetry generators accumulated since the last branching on chameleon vertices can only be used in that branch, hence we output a new subgroup upon returning from the lowest-level chameleon branch.

Rule 2, 3 (immediate refinement) and Rules 4A, 4B (dominant colors) perform constraint propagation, interleaved with two types of branching (Rules 5 and 6). Pruning by accumulated symmetry ensures that generators of each subgroup are irredundant. For many graphs, constraint propagation alone will specialize all chameleon vertices, and for most randomly-generated graphs no branching will be invoked at all. However, even when almost-symmetries form a group, chameleon branching may be necessary, as shown in Figure 2. This example also shows that our algorithm may produce redundant unions-of-subgroups and therefore needs post-processing. However, in general, delayed branching on chameleon vertices and the minimization of branching factors lead to more compact union-of-subgroup expressions. The core algorithm above allows a number of engineering improvements, e.g., its decoupled branching on chameleon vertices can honor color-based constraints and preferences. Another major extension (not described here due to space limitations) is to explicitly track chameleon edges.

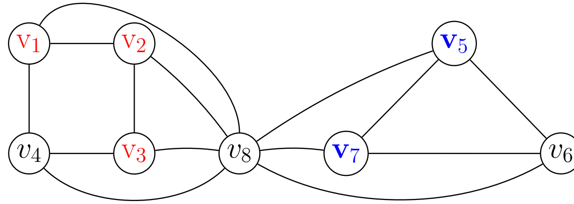

Fig. 2. A graph with three red vertices (v_1, v_2, v_3), two blue vertices ($\mathbf{v}_5$, $\mathbf{v}_7$) and three chameleon vertices (v_4, v_8, v_6). Rule 1 separates v_8 from other vertices, and Rule 4A colors v_8 red. The remaining cell cannot be split (Rule 2), and traditional branching is not allowed in it (Rule 5). Therefore, Rule 6 must be applied with branching factor 4 — on two vertices (v_4,v_6) with two colors. Three of those branches produce non-trivial almost-symmetries after traditional branching (Rule 5), but one of the resulting subgroups contains two others. Indeed, every almost-symmetry becomes a symmetry when v_4 is colored red and v_6 is colored blue.

4 Almost-symmetry-breaking

Consider the DNF-SAT instance $ac + bc$ as example. This formula has one non-trivial permutational symmetry (ab), with SBP $(a \leq b) = a' + b$. We can also swap *either* (a with c) *or* (b with c), but not both. The obvious almost-symmetry-breaking predicate $(a \leq c) + (b \leq c) = (a' + c) + (b' + c) = a' + b' + c$ removes the non-solution 110 allowed by $a' + b$.

To generalize this technique to larger instances, we need to develop theory analogous to the case of full-fledged symmetries. When modeling symmetries on graphs, the group isomorphism $H_{sym} \simeq Aut(\mathcal{G})$ is key — it allows one to pull back group generators (important in symmetry-breaking) from graphs to non-graph objects. Therefore, we now define isomorphism of almost-symmetries so that descriptions of almost-symmetries in terms of generators always map to valid descriptions.

An isomorphism of almost-symmetries is a one-to-one mapping γ such that $\forall$ almost-symmetries π_1, π_2, their product $\pi_1 \cdot \pi_2$ is defined if and only if the product $\gamma(\pi_1) \cdot \gamma(\pi_2)$ is defined, in which case we require that $\gamma(\pi_1 \cdot \pi_2) = \gamma(\pi_1) \cdot \gamma(\pi_2)$.

Isomorphism-of-symmetries proofs for graph constructions (that model constraints and objective functions) [1, 3] all extend to almost-symmetries. Omitting details, each such graph construction defines an isomorphism of containing S_k groups (for k initial variables and k graph vertices), and this mapping remains an isomorphism on every subgroup in the union. Hence, if either of the two permutations $\gamma(\pi_1) \cdot \gamma(\pi_2)$ and $\pi_1 \cdot \pi_2$ is in a valid subgroup, then so is the other.

When defining SBPs, the key issue is not to prohibit *all* good solutions. To aid in this, we build global SBPs from known lex-leader SBPs [2] for every generator of almost-symmetries. In particular, generator SBPs can be conjoined within subgroups because the respective almost-symmetries can be freely composed. When the union of subgroups is derived from a disjunctive constraint, we can OR *all* subgroup SBPs because the lex-smallest assignment satisfying a disjunctive term will satisfy one of subgroup SBPs. Yet, for a general CSP, a unique overall solution may violate all subgroup SBPs, which suggests additional conditions per subgroup G_i. We propose to pick lex-leaders only among those assignments that enable almost-symmetries in G_i.

For almost-symmetries $g_{i1}, g_{i2}, \ldots, g_{im}$ from subgroup G_i, we build their SBP as $\Psi(G_i) := (\Phi_{G_i} \Rightarrow (\wedge_j \psi(g_{ij}))$ where $\psi(g_{1j})$ is a lex-leader SBP for the permutation g_{ij} and Φ_{G_i} is a pre-condition. *The overall SBP for $\cup_i G_i$ then $\wedge_i \Psi(G_i)$.*

For function f, Φ_{G_i} is (ideally the weakest) specialization of don't-cares that turns $g_{ij}, \forall j$ into symmetries. For a constraint graph, if turning g_{ij} into symmetries requires adding (or removing) an edge, then Φ_{G_i} expresses the new constraint represented by this edge (or its negation). When a good precondition is hard to build, we can skip G_i by assuming $\Phi_{G_i} = 0, \Psi(G_i) = 1$.

References

1. F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetry", *IEEE Trans. on CAD*, Sep. 2003, pp. 1117-1137.
2. F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Efficient SymmetryBreaking for Boolean Satisfiability," in Proc. *IJCAI* '03, pp. 271-282.
3. F. A. Aloul, A. Ramani, I. L. Markov and K. A. Sakallah, "Symmetry-Breaking for Pseudo-Boolean Formulas", in Proc. *ASPDAC* '04, pp. 884-887.
4. I. P Gent, B. M. Smith, "Symmetry Breaking in Constraint Programming", in Proc. *ECAI*'00, pp. 599-603.
5. I. Gent, W. Harvey, T. Kelsey, "Groups and Constraints: Symmetry breaking during search", in Proc. *CP* '02, LNCS 2470, pp. 415-430, Springer.
6. I.P. Gent et al, "Conditional Symmetry Breaking," in Proc. *CP* '05, pp. 256-270.
7. P. Gregory and A. Donaldson, "Concrete Applications of Almost-Symmetry", in *Workshop on Almost-Symmetry in Search*, pp. 1-5, Comp. Sci. TR-2005-201, Univ. of Glasgow, 2005.
8. P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting Structure in Symmetry Detection for CNF", in Proc. *DAC* '04, pp. 530-534. `http://vlsicad.eecs.umich.edu/BK/SAUCY/`
9. B. Krishnamurthy, "Short Proofs For Tricky Formulas", *Acta Informatica*, vol. 22, pp.327–337, 1985.
10. B. D. McKay, "Practical Graph Isomorphism", *Congressus Numerantium* 30('81), pp. 45-87.

Appendix A: Exp-sized Union of Subgroups

Consider $n+2$ disconnected vertices: v_1 is red, v_2 is blue, and n vertices are chameleon. All color specializations are indexed by $j = 0..2^n - 1$ such that 1s in the binary expansion of j correspond to red vertices. With $\#j$ red vertices and $n - \#j$ blue vertices, $G_j \simeq S_{1+\#j} \times S_{1+n-\#j}$. No element of G_j for any j maps $v_1 \mapsto v_2$, hence this is an invariant of $\cup_j G_j$. However, $\forall\, i \neq j$, the minimal subgroup containing G_i and G_j contains a permutation that maps $v_1 \mapsto v_2$. Hence in any union-of-subgroups expression for almost-symmetries of the graph in question, each G_j must be contained in a separate term. Therefore, any list of lists of generators representing such an expression requires $\Omega(2^n)$ space.

Appendix B: Handling Chameleon Edges

We will now outline preliminary results on extending the algorithm from Section 3 to handle chameleon edges, using the same overall framework that interleaves branching and partition refinement. The simplest approach is to first branch on all chameleon edges and then continue the existing algorithm. This is indeed a correct overall algorithm, but will generate unions with the greatest possible number of subgroups. Further improvements need to delay branching as much as possible, identify chameleon edges that can be specialized without branching, and

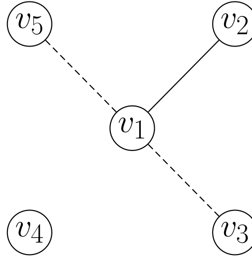

Fig. 3. A graph with one regular and two chameleon edges illustrating difficulties in ensuring that powers of almost-symmetries are, too, almost-symmetries. Consider a clockwise 45°-rotation about the center.

let partition-refinement prune unpromising branches. For example, if we know that each of the two vertices v_i and v_j can only be mapped to itself, the chameleon edge connecting them can be specialized arbitrarily, without branching.

Some of the difficulties in generalizing partition refinement to chameleon edges are largely the same as for chameleon vertices. Namely, the relation "a given edge/vertex can be mapped to another given edge/vertex by an almost-symmetry" is not transitive and therefore is not an equivalence relation. The former prevents free composition of almost-symmetries and the latter means that almost-symmetries do not have orbits in the same sense that symmetries do. This is what complicates splitting a cell with one blue, one yellow and one chameleon vertex.[3] In the presence of chameleon edges, each vertex with χ_i incident chameleon edges may have a range of (χ_i+1) possible degrees depending on how those edges specialize. In the graph from Figure 3, vertices v_2 and v_4 have degrees 1 and 0 respectively, however vertices v_3 and v_5 may each have degrees 0 or 1 independently, and vertex v_1 may have degrees 1, 2 or 3. At this point, we see that v_4 cannot map to v_1 and v_2, but we cannot split v_4 away vertices because v_4 can map to v_3, while v_3 can map to v_2. Thus, we have to branch on both chameleon edges from the start, and soon discover that the resulting four-subgroup union cannot be simplified — the initial branching factor could not have been reduced.

To fathom the scale of additional changes required to improve the naive algorithm, we list several fundamental differences between chameleon edges and vertices.

1. If chameleon edges are present, graph vertices do not have fixed degrees, which undermines the initial degree partition in Rule 1.
2. Existing partition-refinement algorithms do not work in the presence of chameleon edges, which defeats Rule 3.

[3] Color information is used in traditional branching, but partition refinement can often draw more implications from color information that eventually prune additional branches.

3. Unlike specializations of chameleon vertices, specializations of multiple chameleon edges are *not* vertex partitions. This obstructs possible analogues of Rules 4A and 4B for chameleon edges.

The problem we have identified so far with the initial degree-partition (Rule 1) can be addressed effectively as follows (Rule $1'$). After having computed the *degree interval* of each vertex, consider the corresponding *interval graph* — it uses the original vertex set, but an edge between v_i and v_j is established if and only if the intervals labeling these vertices intersect. Without chameleon vertices, every vertex has a fixed degree, and the interval graph is a disjoint union of cliques (one per possible vertex degree). The cells of the degree-partition are connected components of the interval graph, and can be found without building the interval graph explicitly or traversing all of its edges. Chameleon vertices can make the interval graph a lot more complex, but vertices in different connected components still cannot be mapped to each other, and connected components can still be found without building the interval graph explicitly or traversing all of its edges. The proposed **Rule $1'$** is unconditional, just as Rule 1 it replaces. It builds an interval graph (or a reduced version discussed below) and uses its connected components as an initial vertex partition.

The graph in Figure 3 has a connected interval graph, but one can build a non-trivial example based on the graph in Figure 2, whose interval graph consists of a 7-clique and the disconnected vertex v_8. Pick three edges incident to v_8 and color them chameleon. This will leave v_8 with at least four incident edges — more than any other vertex may have. The remaining 7 vertices may still all have degree 3, suggesting that the interval graph has not been affected. The interval graph defined so far can be reduced by removing some edges. Namely, when two vertices connected by one chameleon edge have degree intervals $[m_1, d]$ and $[d, m_2]$ respectively, these vertices cannot simultaneously have the same degree (e.g., vertices v_1 and v_3 in Figure 3). From now on, we shall use this *reduced interval graph* (RIG). However, without new branch-pruning rules for chameleon edges, similar to rules Rules 4A and 4B, it is not useful to build an initial partition before specializing chameleon edges.

We now use the following idea.

> *Label each pair of vertices by* non-edge, edge or chameleon. *Given a vertex-partition, induce a partition on the sets of vertex pairs (edge-partition) where each edge-cell corresponds to a pair of vertex-cells. Two pairs of vertices in different edge-cells cannot be mapped to each other.*

The concept of edge-partition facilitates analogues (but *not replacements*) of Rules 3, 4A and 4B.

3B Edge-cells that contain edges and non-edges, but no chameleon edges, should be split immediately.

4C If an edge-cell contains only chameleon edges, then specialize all of them to non-edges (or, if desired, specialize all to edges).

4D If an edge-cell contains chameleon edges, but all other vertex pairs in it are non-edges (edges), then specialize all chameleons to non-edges (edges).

These rules can be incorporated into the process of inducing an edge-partition. An implementation can ignore edge-cells consisting entirely of non-edges, which improves efficiency for sparse graphs, and only keep track of the remaining two types of cells: (i) edges only, (ii) edges, non-edges and chameleons.

Rule 4D can be illustrated using our running example with the graph from Figure 2 where three edges incident to v_8 are changed into chameleons. The initial vertex partition separates v_8 from 7 remaining vertices. In the induced edge-partition, one particular edge-cell consists of vertex pairs $(v_8, v_j), j = 1..7$ and qualifies for Rule 4D. As a result, all chameleons are specialized to edges, and the degree intervals of some vertices get truncated. In general, the latter can cause the deletion of edges from the RIG and split some vertex cells. This does not happen in our example, where we have specialized all chameleon edges without branching and can now fall back on Section 3.

We have not yet extended vertex-based partition-refinement in Rule 3 to work with chameleon edges, which means that Rule 3 can only be applied after all chameleon edges are specialized. However, we can already formulate the first non-trivial algorithm for finding almost-symmetries with chameleon edges and vertices. The algorithm first builds a RIG and finds its connected components to build an initial degree partition. If some vertex-cells can be split by colors as in Rules 2 and/or 3, this is done and all edges in the RIG that connect newly-split cells are removed (we will abbreviate *split RIG* as *SRIG*). The presence of chameleon edges blocks partition refinement in Rule 3 so far. If there is more than one vertex-cell, the algorithm induces a non-trivial edge-partition. If any of Rules 3B, 4C and 4D trigger, some degree intervals can be adjusted, possibly splitting vertex-cells. Some cells may be eligible for Rules 2, 3, 4A and/or 4B. If the vertex-partition changes, the edge-partition must be refined, and so on. When this cycle stops, branching on chameleon edges can be considered, especially that the branching factor is only 2 per edge (if we want to minimize the number of subgroups, we should delay this branching further, as discussed below).

As it turns out, maintaining edge-partitions also allows us to repair Rule 3. The first step is to reword immediate partition-refinement without chameleon edges in terms of edge-partitions. Namely, each vertex is incident to some edge-cells, regardless of its color, and a vertex-cell can be split so that only vertices incident to exact same edge-cells remain in the same [sub-]cell. This is equivalent to the original partition-refinement by adjacency because edges incident to a given vertex-cell can only be differentiated by their incident vertex-cells on the other side.

The second step is to check that the two-step partition-refinement still works if we calculate incidence to edge-cells through edges and chameleon-edges (without distinguishing them at this point). The two-step refinement remains valid because edge-partition remains an equivalence relation even when chameleon edges are present. We *replace* Rule 3 with **Rule 3′** by swapping two-step partition-refinement for immediate partition-refinement.

The improved algorithm, starts with Rule $1'$, attempts to refine the vertex-partition by Rule 2, then induces an edge-partition and applies Rules 3B, 4C and 4D, then tries color-related rules 4A and 4B, followed by Rule $3'$ and partition refinement. When such combined refinement stops, we first seek non-singleton cells without chameleon vertices and not incident to chameleon edges. In such cells we invoke traditional branching, which is likely to trigger combined refinement again. At some point, we may have to branch on either chameleon edges or vertices – this branching is deliberately decoupled from traditional branching to (i) reduce the number of subgroups, and (ii) allow honoring chameleon-related constraints. Since the branching factor of chameleon edges (=2) is smaller while branching may have a greater impact, chameleon-edge branching is preferable *apriori*. Rather than branch on all remaining chameleon edges, we seek small sets of chameleon edges such that specializing them may split some connected components of the SRIG, at least in some branches. A good heuristic is to find small vertex-cuts in SRIG components and branch on chameleon vertices incident to those vertices — the idea is to truncate vertex degree intervals enough to disconnect the cuts.

Future work. Algorithmic extensions outlined in this appendix appear to address key challenges in handling chameleon edges and suggest that implementation efforts proceed by modifying existing software. Here the main goal would be to add new functionality without affecting the current performance of NAUTY and/or SAUCY in cases without chameleon vertices and edges. Note, however, that efficient data structures used for vertex-partition refinement need to be reworked to support edge-partitions and two-step variant of this classical algorithm.

Closing remarks. While the handling of chameleon vertices and chameleon edges appear so different, they can be reduced to each other via transformations with $O(V^3)$ overhead. First note that vertex colors can be simulated with vertex degrees if we add large numbers of fake edges connecting existing vertices to new vertices of degree 1. Suppose M is the maximal vertex degree in the original graph ($M < V-1$), then to simulate color k we can add $kM < (V-1)^2$ edges to each vertex of that color (more economical constructions are likely possible).

To simulate chameleon vertex colors by chameleon edges, the newly-added edges can be chameleon. Another transformation allows one to simulate chameleon edges by chameleon vertices — remove all edges, add a new vertex v_{ij} for each pair of existing vertices v_i,v_j and establish pairs of edges: (v_{ij}, v_i) and (v_{ij}, v_j). Now artificially increase the degree of newly added vertices to $M+1$ by adding fake edges. If the edge e_{ij} existed in the original graph, color v_{ij} pink, else color it violet. Similarly, chameleon edges can be modeled by chameleon vertices.

The two transformations just described allow us to bound the conceptual differences between chameleon edges and chameleon vertices, but are not necessarily useful in practice.