Павлов Дмитрий Алексеевич

КАНОНИЧЕСКАЯ НУМЕРАЦИЯ ГРАФОВ И БИБЛИОТЕКА NAUTY

Аннотация

Проводится краткий обзор алгоритмов, среди которых лидирует алгоритм Брендана МакКея, реализованный им в библиотеке nauty. Излагаются главные принципы этого алгоритма: уточнение разбиений вершин и использование автоморфизмов для сокращения поиска. Приводится псевдокод всего алгоритма в упрощённом виде.

Ключевые слова: канонический код, каноническая нумерация, изоморфизм графов, автоморфизмы графов, nauty.

Допустим, мы храним и поддерживаем базу данных молекул, или путей метаболизма [1] в биологических клетках, или вовсе базу абстрактных графов, для какой-либо задачи по комбинаторике [2]. В любом случае речь идёт о графах с дополнительными свойствами.

При помещении в базу очередной записи необходимо проверить, что аналогичной записи в базе ещё нет. Выражаясь математически, нужно убедиться в отсутствии графов, *изоморфных* добавляемому. Наиболее грубый (и наименее быстрый) способ выглядит так: мы перебираем всю базу и осуществляем проверку изоморфизма каждого графа из базы с новым образцом.

Проверка изоморфизма двух графов – непростая задача. Очевидно, что она принадлежит к классу NP, но принадлежность её к классам P или NP-полных до сих пор под вопросом. Пока что для этой задачи придуман собственный класс сложности в NP под названием GI [3]. Но даже если у нас будет эффективная процедура для проверки изоморфизма графов, в любом случае, на проверку присутствия нового образца уйдут многие минуты.

Возможно, перед лицом этой опасности и были изобретены канонические коды графов [4]. *Канонический код* – это строковое представление графа, не зависящее от порядка нумерации вершин, своего рода хеш-код без промахов. Если иметь такой код, никогда не потребуется проверять непосредственно изоморфизм графов, – достаточно сравнивать две строки. Коды одинаковы тогда и только тогда, когда графы изоморфны.

Таким образом, перед добавлением графа в базу мы вычислим его канонический код и затем осуществим поиск этого кода (строки) среди имеющихся. Поиск строк, в отличие от изоморфных графов, осуществляется быстро. Проблема решена.

Следует отметить, что канонические коды – не единственное средство. Существует немало кодов, которые вычисляются за полиномиальное время и не являются «каноническими» в том смысле, что могут совпадать у неизоморфных графов. Однако, если коды разные, то графы неизоморфны. Пользуясь этим, можно осуществить отсеивание (screening) большей части неизоморфных графов, и проверять изоморфизм только для оставшихся.

Очевидно, что задача вычисления канонического кода не проще, чем задача проверки изоморфизма, но она интереснее, и канонические коды удобнее в использовании: на один граф вызывается ровно одна специальная процедура.

Любой канонический код состоит из (i) определённой схемы кодирования графа и (ii) процедуры переупорядочения вершин. Нас не интересует схема кодирования: это может быть обход в глубину, или в ширину, или развёрнутая в строку мат-

рица смежности, или просто списки вершин и рёбер... Как бы то ни было, кодирование графа подразумевает какой-то порядок его вершин. Наша задача – упорядочить вершины так, чтобы результат не зависел от изначального порядка. Это называется *канонической нумерацией* (canonical numbering, canonical labeling).

Похоже, что пионерами этой задачи являются Арлазаров, Зуев, Усков и Фараджев, написавшие в 1973 году статью «Алгоритм приведения неориентированных графов к каноническому виду» [5]. Насколько большую роль сыграли их идеи – отдельная тема, выходящая за рамки данной заметки; мы останемся в реалиях сегодняшнего дня.

Реалии, однако, мало изменились за последние 15 лет. Австралийский профессор Брендан МакКей (Brendan McKay) в 1980-х годах написал статью «Practical Graph Isomorphism» [6] и соответствующий программный пакет под названием nauty [7]. Долгое время никому не удалось превзойти nauty по быстродействию, хотя для некоторых частных случаев более эффективные алгоритмы всё же появляются. Особого упоминания заслуживают библиотеки bliss [8] и saucy2 [9], которые на больших разреженных графах уверенно обгоняют nauty. Но эти алгоритмы фактически не более чем nauty с модификациями.

Печально другое: мало кто из программистов сумел хотя бы подойти близко к пониманию того, как работает nauty. Если мы посмотрим на современные программные пакеты, то окажется, что они либо напрямую используют эту библиотеку: Magma, MuPAD, GRAPE, Planarity, Cyberaide, igraph (использует bliss), либо используют более примитивные алгоритмы OpenBabel, Marvin. В любом случае об осмыслении nauty речь не идёт. Приятным исключением является InCHI (http://www.iupac.org/inchi/), в которой присутствует алгоритм, явно сделанный с оглядкой на nauty. Настоящая популярность к алгоритму МакКея придёт тогда, когда в каждом пакете он будет реализован по-своему.

ПЕРЕСТАНОВКИ И УПОРЯДОЧЕННЫЕ РАЗБИЕНИЯ

Говоря здесь о *разбиениях*, мы подразумеваем упорядоченные разбиения множества вершин графа, то есть представления этого множества вершин в виде упорядоченного набора его попарно непересекающихся непустых подмножеств. [*a b c* | *d e*] и [*a c b* | *e d*] – одно и то же разбиение, но [*d e* | *a b c*] от него отличается. Подмножества [*a b c*] и [*d e*] называются *ячейками* (cells).

Если каждая ячейка разбиения π_1 является подмножеством какой-либо ячейки разбиения π_2, то π_1 называется *подразбиением* π_2, а π_2 – *надразбиением* π_1. Также говорят, что π_1 *мельче* π_2, а π_2 *крупнее* π_1. Мы подразумеваем, что при подразбиении сохраняется порядок ячеек. Например, [*a b* | *c* | *d e*] является подразбиением [*a b c* | *d e*] и [*a b* | *c d e*], но не [*c* | *a b d e*].

Разбиение, в котором все подмножества состоят из одного элемента, называется *дискретным* (discrete partition). Дискретное разбиение очевидным образом образует *перестановку* (permutation) вершин. Например, дискретное разбиение [*b* | *c* | *a* | *d* | *e*] образует перестановку (*b*, *c*, *a*, *d*, *e*).

Разбиение, состоящее из одной ячейки, называется *элементарным* (unit partition). Любое разбиение является подразбиением элементарного.

Покажем процедуру перебора всех перестановок вершин, основанную на переборе разбиений. При своей крайней примитивности она является базой всего алгоритма канонической нумерации. Для получения всех перестановок в процедуру следует передать элементарное разбиение (см. листинг 1).

На этом месте у читателя может возникнуть вопрос: зачем нужны разбиения, если наша цель – перебор перестановок? Да, для перебора всех перестановок не нужно вводить понятие разбиений. Но на самом деле все перестановки нам не нужны, а нужна одна перестановка – каноническая, которую мы выбираем как «лучшую» среди какого-то класса перестановок (не обязательно всех). Очевидно, что, чем меньше этот класс, тем лучше, но есть условие: состав

Листинг 1

```
function AllPermutations(π = [V0|...|Vk])
   if π — дискретное разбиение:
      yield π
   else:
      выбрать Vi ∈ π : |Vi| ≠ 1
      for v ∈ Vi:
         AllPermutations([V0|...|{v}|Vi\v|...|Vk])
```

этого класса перестановок не должен зависеть от исходной нумерации вершин. Именно здесь нам поможет идея разбиений: мы будем не отбрасывать перестановки, а уточнять разбиения, участвующие в переборе.

ГОДНЫЕ РАЗБИЕНИЯ

Мы называем разбиение π *годным* (equitable), если для него выполняется следующее свойство: для любых двух ячеек $V_1, V_2 \in \pi$ (не обязательно различных) и для любых $v_1, v_2 \in V_1$ выполняется равенство $d(v_1, V_2) = d(v_2, V_2)$, где $d(v, V)$ – количество вершин в V, смежных с v. Очевидно, что все дискретные разбиения являются годными.

В работе МакКея доказано, что для каждого разбиения π существует единственное его наикрупнейшее подразбиение, являющееся годным. Процедура уточнения (refine) разбиения находит это годное подразбиение. В упрощённом виде она выглядит так (см. листинг 2).

Следовательно, чтобы перебрать все годные подразбиения и все происходящие от них перестановки, нам нужно всего лишь вставить вызов процедуры **Refine** в перебор (см. листинг 3).

Процесс перебора перестановок можно изобразить как дерево разбиений, которое обходится слева направо. Среди листьев первым при обходе встречается самый левый. Обозначим его ξ (рис. 1). (Правая половина дерева показана не целиком. Вызовы **Refine**, которые не уточняют подразбиение, также опущены).

Листинг 2

```
function Refine(π = [V0|...|Vk])
   for Vi ∈ π:
      for Vj ∈ π:
         подразбить Vj, сгруппировав элементы v ∈ Vj по d(v, Vi) и
отсортировав по возрастанию d(v, Vi)
         заменить Vj в π на результат этого подразбиения, удлиняя цикл
по Vi, но не удлиняя цикл по Vj
   return π
```

Листинг 3

```
function AllEquitablePermutations(π = [V0|...|Vk])
   Refine(π)
   if π — дискретное разбиение:
      yield π
   else:
      выбрать Vi ∈ π : |Vi| ≠ 1
      for v ∈ Vi:
         AllPermutations([V0|...|{v}|Vi\v|...|Vk])
```

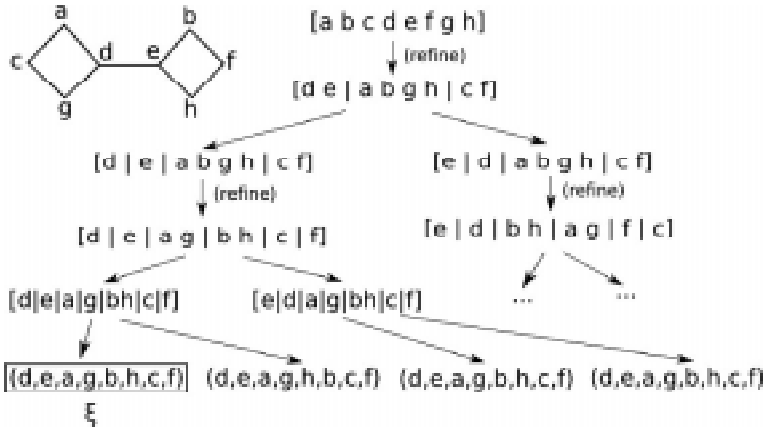

Рис. 1

СРАВНЕНИЯ ПЕРЕСТАНОВОК И АВТОМОРФИЗМЫ

Наша задача – найти каноническую перестановку множества вершин исходного графа. Для этого нужно обзавестить функцией сравнения перестановок cmp, которая бы определяла, какая из двух «лучше» (или они одинаковые). Функция должна задавать транзитивное отношение, а её результат не должен зависеть от первоначальной нумерации вершин графа. Пример такой функции: по исходному графу G и перестановкам π_1 и π_2 построить графы G_1 и G_2, вычислить их матрицы смежности и в качестве $\mathrm{cmp}(\pi_1, \pi_2)$ вернуть результат лексикографического сравнения матриц, развёрнутых в строки. Таким образом, канонической перестановкой будет считаться та, для которой матрица смежности наибольшая; если таких перестановок две и более, то они считаются неразличимыми, и в качестве канонической можно взять любую из них.

Какой смысл для нас имеют «неразличимые» перестановки? Пусть $\mathrm{cmp}(\pi_1, \pi_2)=0$. Несложно доказать, что в этом случае перестановка $\gamma = \pi_2^{-1} \circ \pi_1$ (композиция π_1 и перестановки, обратной π_2) является *автоморфизмом* вершин исходного графа.

Использование автоморфизмов позволяет существенно (иногда – на порядки) сократить перебор при нахождении канонической перестановки. Поэтому в nauty и производных алгоритмах обязательно осуществляется нахождение всей группы автоморфизмов графа, тогда как нахождение собственно канонической нумерации сделано опцией.

Есть следующий способ найти *генераторы* группы автоморфизмов графа: при обработке первого встретившегося дискретного разбиения ξ оно сохраняется. Каждое последующее дискретное разбиение π сравнивается с ξ: если перестановка $\xi^{-1} \circ \pi$ является автоморфизмом графа, то эта перестановка записывается в группу генераторов.

СОКРАЩЕНИЕ ПЕРЕБОРА 1: АВТОМОРФИЗМ $\xi^{-1} \circ \pi$

После обнаружения автоморфизма $\xi^{-1} \circ \pi$ обход можно «поднять» по дереву до общего предка π и ξ, поскольку поддерево этого предка, в котором находится $\xi^{-1} \circ \pi$, «изоморфно» поддереву, в котором находится ξ, и не содержит новой информации. Строгое доказательство приведено в работе МакКея.

СОКРАЩЕНИЕ ПЕРЕБОРА 2: АВТОМОРФИЗМ $\rho^{-1} \circ \pi$

Как было сказано, поиск канонической нумерации – только опция в алгоритме нахождения автоморфизмов. Когда эта опция включена, мы храним ещё одну перестановку ρ, которая является «лучшей» на данный момент (best so far, BSF). При обнаружении первой перестановки ξ мы копируем её в ρ. Каждую последующую перестановку π мы сравниваем с ρ.

Если $\mathrm{cmp}(\pi, \rho) > 0$, то мы копируем ξ в ρ. Если $\mathrm{cmp}(\pi, \rho) < 0$, то ничего не делаем. Если же $\mathrm{cmp}(\pi, \rho) = 0$, то $\rho^{-1} \circ \pi$ является новым автоморфизмом, и обход можно «поднять» по дереву до общего предка π и ρ.

СОКРАЩЕНИЕ ПЕРЕБОРА 3: ОРБИТЫ

Орбита вершины v – это подмножество вершин, в которые может перейти v в результате действия автоморфизмов. Например, на рис. 2 показаны орбиты вершин графа.

В начале работы nauty каждая вершина принадлежит своей собственной орбите. По мере открывания новых автоморфизмов орбиты увеличиваются, а их количество – сокращается. Например, если найден автоморфизм $(a, b, c, d, e) \to (b, a, c, e, d)$, то список орбит из $[a][b][c][d][e]$ становится $[a\ b]\ [c]\ [d\ e]$. Если после этого найден автоморфизм $(a, b, c, d, e) \to (a, c, b, d, e)$, то список орбит становится $[a\ b\ c]\ [d\ e]$.

Рассмотрим «линию предков» первого дискретного разбиения ξ. Предположим, мы в процессе обхода дерева вернулись к какому-либо из предков o (назовём его υ), чтобы продолжить обход других потомков υ, перебирая другие элементы ячейки V_i. Если рассматриваемый элемент $v_k \in V_i$ лежит на одной орбите с уже рассмотренным элементом $v_j \in V_i$, то существует автоморфизм, открытый в процессе обхода более ранних узлов, который переводит v_j в v_k. В работе МакКея доказано, что в этом случае нет смысла рассматривать всё поддерево, произведённое отделением v_k: все листья этого поддерева будут изоморфны ранее открытым листьям, и все автоморфизмы, которые будут найдены при обходе этого поддерева, будут повторять предыдущие.

Небольшое замечание: после реализации процедуры **`Refine`** может показаться, что уточнение тривиального разбиения как раз и определяет орбиты вершин. Для многих графов это действительно так, но не для всех. На рис. 3 показан пример графа, на котором это не выполняется.

Для всех графов, однако, верно то, что разбиение по орбитам является подразбиением уточнения тривиального разбиения.

СОКРАЩЕНИЕ ПЕРЕБОРА 4: fix и mcr

С узлами, которые не являются предками ξ, предыдущее сокращение перебора делать нельзя. Для них есть менее сильный, но тоже действенный критерий отсечения потомков. Для применения этого критерия необходимо сохранять информацию о каждом автоморфизме в отдельности (вместо множества орбит, которое накапливает информацию о всех автоморфизмах в совокупности). Для каждого найденного автоморфизма g запоминаются два множества: $\mathrm{fix}(\gamma)$ и $\mathrm{mcr}(\gamma)$. $\mathrm{fix}(\gamma)$ – это множество вершин, которые при действии автоморфизма остаются на своих местах, $\mathrm{mcr}(\gamma)$ – это множество вершин, которые являются минимальными по номеру в своих орбитах. Имеются в виду орбиты, составленные по одному автоморфизму γ, а не те, которые используются в предыдущем способе отсечения.

Рассмотрим множество V_i, элементы которого мы перебираем для продолжения обхода дерева из какого-то узла глубины l. В процессе пути от корня дерева до этого узла l вершин были «зафиксированы», когда мы отделяли их от их ячеек. Назовём множество этих вершин fixed.

В работе МакКея доказано следующее: если $\mathrm{fixed} \subset \mathrm{fix}(\gamma)$, то можно присвоить в V_i

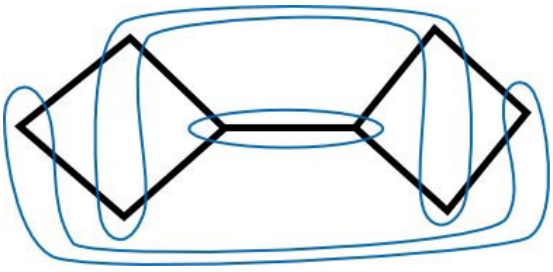

Рис. 2

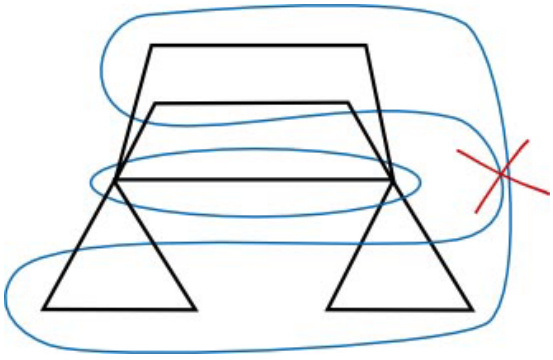

Рис. 3

пересечение $V_i \cap \mathrm{mcr}(\gamma)$ без ущерба для алгоритма. Фактически, речь идёт об автоморфизмах, которые «не затрагивают» зафиксированные вершины. Их орбиты можно эксплуатировать так же, как и в предыдущем разделе, что и делается путём пересечения V_i и mcr(γ). Означенная операция выполняется для всех γ, fix и mcr которых мы успели сохранить на данный момент. Следует отметить, что генераторов группы автоморфизмов графа может быть очень много, и сохранять все fix и mcr не всегда возможно. МакКей рекомендует ограничивать количество сохраняемых fix и mcr числом, пропорциональным количеству вершин в графе с коэффициентом пропорции больше 1 (у него почему-то 50/32).

АЛГОРИТМ ЦЕЛИКОМ

В коде (см листинг 4) имеются глобальные переменные:

`getcanon` – опция получения канонической нумерации в ρ;

`gca_first` – уровень общего предка текущего узла и ξ;

`gca_canon` – уровень общего предка текущего узла и ρ;

`orbits` – массив индексов орбит; содержит для каждой вершины номер минимальной вершины в её орбите;

Ψ – список посчитанных множеств fix и mcr для найденных автоморфизмов.

ДЕТАЛИ РЕАЛИЗАЦИИ

Авторская имплементация nauty содержит трюки для увеличения быстродействия и уменьшения расхода памяти:

– для хранения графа и множеств используются битовые массивы;

– вся цепочка подразбиений от корня дерева к текущему узлу кодируется всего двумя массивами (lab и ptn). При возвращении к родительскому узлу разбиение восстанавливается функцией recover.

Подробности можно узнать в документации по использованию nauty [10] и в исходном коде.

ПРИМЕНЕНИЕ

Прелесть изложенного алгоритма в том, что в нём мало завязок на то, что исходное множество вершин – это вершины простого графа. Фактически можно находить автоморфизмы и каноническую нумерацию для чего угодно, например:

– для ориентированных графов (никаких изменений в алгоритме не требуется; МакКей в статье упоминает ориентированные графы, и в nauty есть соответствующая опция **`digraph`**);

– для графов с раскрашенными вершинами (всё, что требуется, – начинать не с -элементарного разбиения, а сгруппировать вершины по цветам; эта опция также описана у МакКея и имеется в nauty);

– для графов с раскрашенными рёбрами (всё, что требуется, – изменить функцию **`cmp`** так, чтобы она учитывала цвета рёбер; также можно более эффективно написать процедуру **`Refine`** для уточнения разбиения).

МОДИФИКАЦИИ АЛГОРИТМА

Не было упомянуто об огромном количестве дополнительных оптимизаций, имеющихся в коде nauty. МакКей признаёт, что у него самого не все они задокументированы. Вот только некоторые:

- «Даровые» автоморфизмы (cheap automorphisms, implicit automorphisms), для обнаружения которых не требуется доходить до дискретного разбиения. Работают в nauty только для неориентированных графов (цвета вершин допускаются).
- Проверка автоморфизма **`cmp`**(π, ξ) может делаться более эффективно без вызова cmp, с учётом того, что ξ никогда не меняется и нас интересует только, есть ли изоморфизм.
- Псевдо-канонические коды разбиений, благодаря которым можно выявить отсутствие автоморфизма, не проверяя его явно. Они же используются для предварительного определения знака функции cmp.
- При выборе ячейки V_i для подразбиения можно применять различные эвристики с целью сокращения перебора.

Листинг 4

```
function AutomorphismSearch(π)
   FirstNode(1, π)

function FirstNode(level, π = [V0|...|Vk])
   π ← Refine(π)
   if π – дискретное:
      ξ ← π
      gca_first ← level
      if getcanon:
         ρ ← π
         gca_canon ← level
      return level-1
   Vi ← первая ячейка π, в которой больше одной вершины
   for v ∈ Vi
      if orbits[v] ≠ v:
         continue (сокращение 3)
      π'←[V0|...|{v}|Vi\v|...|Vk]
      fixed ← fixed ∪ v
      if v – первая вершина в Vi:
         rtnlevel ← FirstNode(level+1, π')
         gca_first ← level
      else:
         rtnlevel ← OtherNode(level+1, π')
      fixed ← fixed \ v
      if rtnlevel < level:
         return rtnlevel
      if level < gca_canon:
         gca_canon ← level
   return level-1

function OtherNode(level, π = [V0|...|Vk])
   π ← Refine(π)
   if π – дискретное:
      if cmp(π,ξ)=0:
         AddFixMcr(ξ^-1°π)
         JoinOrbits(ξ^-1°π)
         return gca_first (сокращение 1)
      if getcanon:
         comp ← cmp(π, ρ)
         if comp=0:
            AddFixMcr(ρ^-1°π)
            JoinOrbits(ρ^-1°π)
            return gca_canon (сокращение 2)
         if comp>0:
            ρ ← π
      return level-1
   Vi ← первая ячейка π, в которой больше одной вершины
   for (fix, mcr) ∈ Ψ:
      if fixed ⊂ fix:
         Vi ← Vi ∩ mcr (сокращение 4)
   for v ∈ Vi:
      π'←[V0|...|{v}|Vi\v|...|Vk]
      fixed ← fixed ∪ v
      rtnlevel ← OtherNode(level+1, π')
      fixed ← fixed \ v
      if rtnlevel ← level:
         return rtnlevel
      if level < gca_canon:
         gca_canon ← level
   return level-1
```

Литература

1. http://en.wikipedia.org/wiki/Signal_transduction_pathways
2. http://en.wikipedia.org/wiki/Graph_enumeration
3. http://en.wikipedia.org/wiki/Graph_isomorphism#Complexity_class_GI
4. http://en.wikipedia.org/wiki/Graph_canonization
5. *Арлазаров В.Л., Зуев И.М., Усков А.В., Фараджев И. А.* Алгоритм приведения неориентированных графов к каноническому виду. ЖВМ и МФ, 1973.
6. *Brendan D. McKay.* Practical Graph Isomorphism. Congressus Numerantium, Vol 30 (1981), 45–87.
7. http://cs.anu.edu.au/~bdm/nauty/
8. *Tommi Junttila, Petteri Kaski.* Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, SIAM, 2007
9. *Paul T. Darga, Karem A. Sakallah, Igor L. Markov.* Faster Symmetry Discovery using Sparsity of Symmetries. Proceedings of the 45th annual Design Automation Conference, 149–154, 2008.
10. http://cs.anu.edu.au/~bdm/nauty/nug.pdf

Abstract

Canonical code is the notation for the graph structure, which is invariant to isomorphism. The article briefly illustrates the significance of canonical codes in general, and reveals some connection of the canonical code building to other problems common to computer science. A short overview of the currently popular algorithms is presented. The ideas of Brendan McKay's algorithm, implemented in his famous library «nauty», are described. That is, refinement of the vertex partition and use of automorphisms to prune the search tree. A simplified pseudo-code of the entire algorithm is shown.

Павлов Дмитрий Алексеевич,
аспирант кафедры прикладной математики ФМФ СПбГПУ,

dmitry.pavlov@gmail.com