# Algorithm to Generate All Connected Simple Graphs of Given Order

Matsui Tetsushi[1,a)] Uno Takeaki[1,b)]

**Abstract:** We present an efficient algorithm to generate connected simple graphs of given order. It is based on the reverse search technique and a new vertex partitioning strategy. By implementing the algorithm in C, we can generate all connected simple graphs of order 11 in five and a half hours.

**Keywords:** graph, generate, algorithm, reverse search

## 1. Introduction

The efforts to generate all graphs date back to 1960s. The limit of graph generation how big graphs can be cataloged have been gradually raised as available computing resources expanded. Heap is the first person who reported generation of graphs with computer; he completes generating simple graphs with 8 vertices in 1972 [9]. Baker et al. soon obtain all 9 vertex graphs on PDP-10 in 1974 [3]. In 1985, Cameron et al. reach to graphs of order 10 within 15 hours [4].

Checking isomorphisms between graphs is unavoidably necessary not to yield duplicated graphs. On the one hand, Graph isomorphism problem, GI in short, is not an easy problem; it has not been shown to be in polynomial time [15]. On the other hand, GI is not so hard problem either; two random graphs can be distinguished in linear time with very low error probability [2]. There are several algorithms for graph isomorphism problem, including nauty [12], VF and VF2 [5], [7], or gSpan [16], [17]. Despite difficulty of GI, it is known that generating all $n$ vertex graphs is polynomial delay [8], [14].

Motivated to provide a catalogue of graphs and a practical program to make it, this paper presents an efficient algorithm to generate connected simple graphs. It is based on the reverse search technique and a new vertex partitioning strategy. The reverse search is one of the most prominent frameworks for generating combinatorial objects [1], though McKay only reluctantly mentions it in his survey of generating combinatorial objects [13]. Its merit is that the algorithm needs to check graph isomorphisms for only a small number of graphs compared with the previous algorithms.

The structure of the paper is as follows. First, we summarize the basic concepts to be used in our graph generation algorithm. Then, in Section 3, we explain the details of the algorithm. Section 4 describes an implementation and experimental results, followed by discussion.

[1] Principles of Informatics Research Division, National Institute of Informatics
[a)] tetsushi@nii.ac.jp
[b)] uno@nii.ac.jp

## 2. Automorphism and Vertex Partitioning

Automorphism of graph and vertex partitioning play the central rôle in graph generation. By considering automorphisms, we can reduce the number of duplicatedly checked graphs during the graph generation process. Vertex partitioning provides efficient means to handle automorphism, and has been used in several isomorphism tests ([6] and [12], for example).

### 2.1 Basic Terminology

A graph $G$ is a pair $(V, E)$ of a vertex set $V$ and an edge set $E$. An edge of a simple graph connects a pair of distinct vertices, and no pairs of edges connect the same pair of vertices. For a simple graph $G = (V, E)$, if $|V| = n$, we often identify $V$ as $[n] = \{0, 1, \ldots, n-1\}$, and $E$ as a set of two-element subsets of $V$. For two graphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$, a one-to-one onto map of vertex sets from $V_0$ to $V_1$ is called an isomorphism if it is also a one-to-one onto map from $E_0$ to $E_1$ within the identification above. Two graphs are isomorphic if there is an isomorphism between them, denoted as $G_0 \cong G_1$. In particular, if two graphs are the same graph, an isomorphism map is called an automorphism. The set of automorphisms of a graph $G$ forms a group called the automorphism group of $G$, denoted by $\mathrm{Aut}(G)$; it is naturally a subgroup of the symmetric group of vertices $\mathrm{Sym}(V)$, and the identity map is the identity element of the group.

For a set $S$, if there are subsets $S_i$ such that $S = \bigcup S_i$ and $S_i \cap S_j = \emptyset$ for any $i \neq j$, we call $\{S_i\}$ a partition of $S$ and each $S_i$ a cell. In the following, we consider partitions of the vertex set $V$ of $G$. The most important partition is

the automorphism partition: $V = \bigcup V_i$ with each cell $V_i$ is an orbit of $\mathrm{Aut}(G)$. However, since to know the automorphism partition is as hard as GI [11], [15], we do not expect it to be always available. An order of partitions of a set $S$ is given as the following. For partitions $T = \{T_i\}$ and $U = \{U_i\}$ of $S$, if for any $T_i$ there exists a superset $U_j$, then we say $T$ is finer than $U$, or $U$ is coarser than $T$, and $T \leq U$ denotes such a relation. Obviously, the discrete partition $D = \{\{i\} \mid i \in V\}$, the automorphism partition $A = \{\Delta \mid \Delta \text{ is an orbit of } \mathrm{Aut}(G)\}$ and the trivial partition $T = \{V\}$ satisfy $D \leq A \leq T$.

A coloring of $G$, or more precisely a vertex coloring of $G$, is a map from the vertex set $V$ to a set of colors. A coloring map induces a vertex partitioning, or vice versa. Thus, we use the terminologies of partitioning and coloring interchangeably. Moreover, we assume that colors are linearly ordered; without loss of generality, colors are supposed to be integers.

### 2.2 Stabilization

Stabilization is one of the algorithms to obtain one partition to another finer partition, based on the colors of the neighbors [6]. It is sometimes called the 1-dim Weisfeiler-Lehman algorithm.

Let $P$ be a color partition of vertices of $G$, assuming that colors are integers in range $[1, |V|]$. A color partition can be viewed as a function, so that $P(v)$ denotes the color of a vertex $v$ in the color partition $P$. The procedure is described in Algorithm 1.

**Algorithm 1** stabilize($P$, $G$)

```
repeat
    Q = P.
    for all v ∈ V do
        N_v ← array of zeros with length |V| + 1.
    end for
    for all v ∈ V do
        N_v[0] ← Q(v).
        for all u in neighbors of v do
            N_u[Q(v)] ← N_u[Q(v)] + 1.
        end for
    end for
    Sort list M of N_v s with lexicographic order.
    P(v) ← min{i | M[i] = N_v} + 1.
until P = Q.
return P.
```

## 3. The Algorithm

### 3.1 Reverse Search

We describe the algorithm to generate all connected simple graphs of given order in this section. It is based on the reverse search technique and a new vertex partitioning strategy.

Reverse search is one of the most prominent frameworks for generating combinatorial objects. We refer the reader to [1] for general description of the reverse search method.

Any generation algorithms using reverse search need their own "parent-child" relation among generated objects. We define the parent-child relation for graph generation as the following: the parent of a graph $G$ is $G - v$, where $v$ is the "weakest" vertex of $G$. Thus, the order of a graph $G$ is one larger than that of its parent.

Algorithm 2 is the skeleton of the whole generating process. Note that a graph which is put into $P_0$ is not output here. Each of those graphs will be used to call another RevSearch with it after finishing the first invocation of RevSearch. Each commented Step from 1 to 4 is explained later. Before describing Step 1 of Algorithm 2, we introduce a new partitioning algorithm called RUMBA in the next subsection, and Subsection 3.3 describes how to determine the weakest vertex in a graph, which is used on Step 2. Because Step 1 depends on the definition of parent-child relation, its explanation is deferred until Subsection 3.4. Finally, the pools used on Steps 3 and 4 are explained in Subsection 3.5.

**Algorithm 2** RevSearch ($n$, $G$, $P_0$)

```
if the order of G is n then
    Output G, and return.
end if
P ← an empty pool.
C ← a set of next generation graphs constructed by adding a
new vertex v' and incident edges onto G, including all children
of G.                                                  ▷ Step 1
for all G' ∈ C do
    Try to determine the weakest vertex v of G'.       ▷ Step 2
    if v is found then
        if v = v' then
            if any graph H in P with same RUMBA code as G'
are non-isomorphic to G' then
                Put G' into P.
            end if                                     ▷ Step 3
        end if
    else
        Put G' into P_0.                               ▷ Step 4
    end if
end for
for all H ∈ P do
    RevSearch(n, H).
end for
```

### 3.2 RUMBA

To reduce the redundancy in graph generation, it is desirable to classify the vertices into a partition as close to the automorphism partition as possible. We use RUMBA partitioning algorithm; RUMBA stands for RUMBA is Uno-Matsui BFS Algorithm. The algorithm, as the name suggests, uses BFS trees of a graph to distinguish vertices for obtaining a finer partition.

Let $P = \{P_0, \ldots, P_k\}$ be a coloring partition of the vertex set $V$ of a fixed graph $G$. Suppose that a cell $P_i$ includes two or more vertices in it. Then, we proceed as Algorithm 3.

RUMBA tries to update the current partition with one step RUMBA for every cell with two or more vertices, and

**Algorithm 3** one step RUMBA ($P$, $i$, $G$)

Construct BFS trees of $G$ rooted on each $p \in P_i$.
**for all** $v \in V$ **do**
collect the following information:
(1) The current color $c$.
(2) The numbers of shallower neighbors for each color.
(3) The numbers of neighbors on the same level for each color.
(4) The numbers of deeper neighbors for each color.
where shallower means that the root is nearer than $v$, and deeper means farther.
**end for**
Group vertices according to the collected color information.
Sort the groups and recolor the vertices.
Stabilize the new partition.

stops when no such cells lead to update the partition.

The RUMBA partitions often coincide with the automorphism partitions, but there are exceptions. RUMBA is similar to the Corneil-Gotlieb algorithm, and it is easy to show that if Corneil-Gotlieb algorithm gives the automorphism partition for a graph then RUMBA also gives it. Corneil and Gotlieb once conjectured that their algorithm always gives the automorphism partition [6], however Mathon gives a counterexample with 25-vertex graph [10]. Unfortunately, RUMBA also fails to give the automorphism partition for the counterexample graph.

### 3.3 Weakness

Calculation of weakness of vertices is carried out on Step 2 of Algorithm 2. We define the weakness by the lexicographic order of quadruple: connectivity, negated degree, stabilized degree and RUMBA color. The greater the quadruple of a vertex is, the weaker the vertex is ranked.

The first component is the connectivity. Since we are constructing connected graphs, the parent should be connected as well. Thus, to eliminate all cut vertices from the candidate list, we place 0 for cut vertices and 1 for others.

The second to fourth components are colors of labelings. The second component is the negated degree of the vertex, the third component is stabilization of the former, finally the fourth component is RUMBA color. The weakness of a vertex is essentially determined by RUMBA partition, and the second and third components of quadruples are corresponding to shortcuts.

There is a subtle point in this definition; it is automorphism. As explained above, RUMBA sometimes does not give the automorphism partition. In such cases, vertices in the greatest color might have to be separated into two or more cells in the automorphism partition. Thus, we should check whether the greatest colored cell is an orbit of the automorphism group or not. Note, however, that we do not know the automorphism group itself. Therefore, the weakest vertices are the vertices with the greatest color in RUMBA partition which are known to form an orbit of the automorphism group, except cut vertices. If the cell is not an orbit, we leave undetermined which vertex is the weakest.

### 3.4 Constructing the Next Generation

For a given parent graph $G$, we construct a set of next generation graphs including all children of $G$ in Step 1 of Algorithm 2. On this stage, though it is desirable to obtain exactly all children without duplications, we have to tolerate getting children of other parents or identical child graphs; those redundancy will be checked later. By the definition of parent-child relation, constructed graphs have one extra vertex than $G$, whose degree is the lowest among non-cut vertices.

In the following algorithm, $k$ is at most minimum degree of $G$ plus one. Moreover, the partition $P$ satisfies $P \leq A$, where $A$ is the automorphism partition.

The subroutine `subproto` takes three arguments: $G$ the

**Algorithm 4** proto($G$, $k$, $P$)

$\mu \leftarrow$ the minimum non-cut degree of $G$.
**if** $k = \mu + 1$ **then**
  $U \leftarrow$ non-cut vertices with degree $\mu$ in $G$.
  **if** $|U| > k$ **then**
    Error. ▷ There remains vertices with degree $\mu < k$.
  **else if** $|U| = k$ **then**
    Output $U$.
  **else**
    **for all** $W$ in subproto($G$, $k - |U|$, $P$, $P \setminus U$) **do**
      Output $U \cup W$.
    **end for**
  **end if**
**else**
  **for all** $W$ in subproto($G$, $k$, $P$, $P$) **do**
    Output $W$.
  **end for**
**end if**

graph, $k$ the number of edges to be added, $P$ a partition, and $Q$ a partial partition from where neighbor vertices are chosen.

One can obtain a set of next generation graphs, by adding edges between the newly added vertex and every $k$ vertices choice among $n - 1$ vertices of $G$. One possible realization is to use all $k$ element subsets from $Q$, but the procedure can be more efficient by the presence of $P$. If one knows a partition coarser than trivial but finer than the automorphism partition, the number of produced graphs may be decreased, since production of redundant twin graphs may be suppressed.

In the following algorithm, for the (partial) color partition $Q$, $Q_i$ denotes a partition colored as $i$.

**Algorithm 5** subproto($G$, $k$, $P$, $Q$)

$z \leftarrow$ minimum color of $Q$.
**if** $|Q_z| > 1$ **then**
  $m \leftarrow \min(Q_z)$.
  **if** $k = 1$ **then**
    Output $\{m\}$.
  **else**
    Let $P'$ be a copy of $P$ except that $P_z \setminus \{m\}$ get the color $z + 1$.
    $P' \leftarrow$ `stabilize`($P'$, $G$).
    **for all** $W$ in `subproto`($G$, $k - 1$, $P'$, $P' \cap (\bigcup Q \setminus \{m\})$) **do**
      Output $\{m\} \cup W$
    **end for**
  **end if**
**else if** the number of cells of $Q > 1$ **then**
  **if** $k = 1$ **then**
    Output $Q_z$.
  **else**
    **for all** $W$ in `subproto`($G$, $k - 1$, $P$, $Q \setminus Q_z$) **do**
      Output $Q_z \cup W$
    **end for**
  **end if**
**else if** $|Q_z| = k$ **then**
  Output $Q_z$.
**end if**
**if** the number of cells of $Q = 1$ **then** Stop.
**end if**
**for all** $W$ in `subproto`($G$, $k - 1$, $P$, $Q \setminus Q_z$) **do**
  Output $W$.
**end for**

### 3.5 Pool

On Steps 3 and 4 of Algorithm 2, existence of identical graphs has to be checked and all but one representative have to be discarded if found. We use pools of graphs to realize the functionality. On Step 4, the graphs have to be memorized throughout the process, but on Step 3, the memory has to be kept just for one parent graph $G$. Then, we call the pool used on Step 4 the global pool, while ones on Step 3 local pools.

For efficiency of the pools, we need a classification of graphs as fine-grained as possible but it has to be computed

| $\|V\|$ | # of graphs | Time | # of graphs per second |
|---|---|---|---|
| 8 | 11117 | 0.24 s | $4.6 \times 10^4$ |
| 9 | 261080 | 5.6 s | $4.7 \times 10^4$ |
| 10 | 11716571 | 3 m 43 s | $5.3 \times 10^4$ |
| 11 | 1006700565 | 5 h 23 m | $5.2 \times 10^4$ |

**Table 1** Timing Data

easily. Corneil and Gotlieb uses an induced graph from their partitioning [6]. It seems reasonable to use similar induced graph from the RUMBA partitioning.

Since, the Corneil-Gotlieb or the RUMBA partitioning does not provide a full isomorphism test, as noted in Section 3.2, we still need some other isomorphism tests.

# 4. Discussion

## 4.1 Implementation Remarks

There are a few implementation remarks.

Though the definition of the weakest vertex in Section 3.3 depends on RUMBA labeling, There are a few shortcut conditions to determine the weakest vertex: (1) a vertex is the unique non-cut vertex with the smallest degree; (2) a vertex is the unique non-cut vertex with the smallest neighbor degrees among the smallest degree non-cut vertices. These shortcut conditions reduce the need for labeling with RUMBA, which is rather heavy.

As a nature of the reverse search framework, graph generation can be parallelized. In principle, generations from one parent graph and another are independent. The only hindrance is the global pool.

Pools have to check graph isomorphisms, and there are choices of algorithms. On the one hand, in case that there are only a few graphs classified in the same invariant class, a matching type isomorphism is preferable. On the other hand, in case that there are several graphs, a canonical code type isomorphism is preferable.

## 4.2 Implementation and Experiments

We have written a program implementing the algorithm above. The program is written in C99. The timing data presented in Table 1 are the results of execution on Apple MacBook Pro with Intel Core i5 2.53 GHz CPU and 8 GiB memory. The program was compiled with GCC 4.2.1, with the optimization flag `-O3 -fomit-frame-pointer`. Memory consumption during each execution was only a few MiB. The number of graphs per second, the fourth column of the table, is nearly constant in this range of order. Then, if we extrapolate the total time for order 12, it takes about a month to output all 164059830476 graphs.

The nauty package includes a utility called `geng` to generate graphs. Currently, `geng` is faster than our implementation of the algorithm, but we hope our implementation will be improved to beat `geng`. The most significant difference is the use of the group theory. The present algorithm uses mainly the RUMBA labeling, but explicitly considering automorphism groups can reduce operations, such as those needed to obtain lower labelings for Algorithms 4 and 5.

## 4.3 Future Works

Possible future extensions of the algorithm are (1) to generate all disconnected simple graphs of given order, (2) to generate all connected simple graphs of given order and maximum degree, or (3) to generate all connected simple graphs of given order and minor graphs.

## References

[1] Avis, D. and Fukuda, K.: Reverse Search for Enumeration, *Discrete Applied Mathematics*, Vol. 65, No. 1–3, pp. 21–46 (1996).
[2] Babai, L. and Kučera, L.: Canonical Labeling of Graphs in Linear Average Time, *20th Annual Symposium on Foundations of Computer Science*, IEEE, pp. 39–46 (1979).
[3] Baker, H. H., Dewdney, A. K. and Szilard, A. L.: Generating the Nine-Point Graphs, *Mathematics of Computation*, Vol. 28, No. 127, pp. 833–838 (1974).
[4] Cameron, R. D., Colbourn, C. J., Read, R. C. and Wormald, N. C.: Cataloguing the Graphs on 10 Vertices, *Journal of Graph Theory*, Vol. 9, No. 4, pp. 551–562 (1985).
[5] Cordella, L. P., Foggia, P., Sansone, C. and Vento, M.: Performance Evaluation of the VF Graph Matching Algorithm, *International Conference on Image Analysis and Processing*, IEEE, pp. 1172–1177 (1999).
[6] Corneil, D. G. and Gotlieb, C. C.: An Efficient Algorithm for Graph Isomorphism, *Journal of the Association for Computing Machinery*, Vol. 17, No. 1, pp. 51–64 (1970).
[7] Foggia, P., Sansone, C. and Vento, M.: A performance comparison of five algorithms for graph isomorphism, *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pp. 188–199 (2001).
[8] Goldberg, L. A.: Efficient Algorithms for Listing Unlabeled Graphs, *Journal of Algorithms*, Vol. 13, pp. 128–143 (1992).
[9] Heap, B. R.: The production of graphs by computer, *Graph theory and computing*, Academic Press, pp. 47–62 (1972).
[10] Mathon, R.: Sample Graphs For Isomorphism Testing, *Proceedings of ninth Southeastern Conference of Combinatorics, Graph Theory, and Computing*, Congressus numerantium, Utilitas Mathematica Pub., pp. 499–517 (1978).
[11] Mathon, R.: A Note of the Graph Isomorphism Counting Problem, *Information Processing Letters*, Vol. 8, No. 3, pp. 131–132 (1979).
[12] McKay, B. D.: Practical Graph Isomorphism, *Proceedings of the Tenth Manitoba Conference on Numerical Mathematics and Computing, Vol. I*, Congressus numerantium, Utilitas Mathematica Pub., pp. 45–87 (1981).
[13] McKay, B. D.: Isomorph-free Exhaustive Generation, *Journal of Algorithms*, Vol. 26, pp. 306–324 (1998).
[14] Ramon, J. and Nijssen, S.: Polynomial-Delay Enumeration of Monotonic Graph Classes, *Journal of Machine Learning Research*, Vol. 10, pp. 907–929 (2009).
[15] Read, R. C. and Corneil, D. G.: The Graph Isomorphism Disease, *Journal of Graph Theory*, Vol. 1, No. 4, pp. 339–363 (1977).
[16] Yan, X. and Han, J.: gSpan: Graph-Based Substructure Pattern Mining, *Proceedings of 2002 International Conference on Data Mining* (2002).
[17] Yan, X. and Han, J.: gSpan: Graph-Based Substructure Pattern Mining, Technical report, University of Illinois at Urbana-Champaign (2002).