Generating Small Graphs up to Isomorphism

Nagy Szabolcs

2023/24 2nd semester, Math Project II.

Summary of the project The goal of the project is to create a user-friendly tool for generating every small graph of a certain type up to isomorphism.

The realization of this involves the creation of a graph-generating algorithm, a canonization algorithm to be used by the generator, and a customizable graph-filter class that we can use to tell the generator which specific graphs we want to make.

Our approach takes great inspiration from Brendan McKay's and Adolfo Piperno's nauty project, available at their website [3].

Generating labeled graphs Generating every possible labeled graph of size n is simple enough, we can start off with the empty graph $G = (\emptyset, \emptyset)$, then recursively extend G by adding an extra vertex, and connecting it to the already existing vertices in all possible ways, yielding 2^k new labeled graphs from every graph of order k on level k. It is easy to see that such a construction would lead to acquiring all labeled graphs of order n.

In pseudocode:

Algorithm 1 Labeled graph generator

```
1: procedure Gen($n$)                                    ▷ Main call
2:     $G \leftarrow (\emptyset, \emptyset)$
3:     Gen_Rec($G, n$)
4: end procedure
5: procedure Gen_Rec($G, n$)                             ▷ Recursive generator
6:     $k \leftarrow |V(G)|$
7:     for all $P \in 2^{\{0,\dots,k-1\}}$ do
8:         $G' \leftarrow (V(G) + v_k, E(G) + \{v_i v_k : i \in P\})$
9:         if $k + 1 = n$ then
10:            Output $G'$
11:        else
12:            Gen_Rec($G', n$)
13:        end if
14:    end for
15: end procedure
```

This is not usually practical, simply because of the sheer amount of possible labeled graphs. There are $2^{\binom{n}{2}}$ unique labeled graphs of order n, which is a really fast-growing amount. Even with additional graph-property constraints, this type of generation gets out of control really quickly.

Generating unlabeled graphs One way to cut down the number of graphs to compute is to only generate one instance from every isomorphism class of graphs of order n. In most relevant situations, such as when testing some graph-property hypothesis, we really do not need or want to examine more than one graph from any isomorphism class, so if there was an efficient way to filter out duplicates, it could save us a lot of trouble.

Note that the number of unique unlabeled graphs of order n also grows exponentially: given k unique unlabeled graphs of order $n \geq 1$, we can also get $2k$ unique ones of order $n + 1$ by simply adding an extra vertex with degree either 0 or n to each of them. Thus, 2^{n-1} is a trivial lower-bound for the number of unlabeled graphs. Still, they multiply at a somewhat more modest rate than labeled graphs, meaning that computations can be made for higher values of n.

n	1	2	3	4	5	6	7	8
#labeled	1	2	8	64	1,024	32,768	2,097,152	268,435,456
#unlabeled	1	2	4	11	34	156	1,044	12,346

Table 1: Number of labeled/unlabeled graphs of order n.

Brendan McKay [2] came up with a neat and "simple" way to modify the above algorithm such that the yielded graphs are all non-isomorphic: after G' is created, we check to see whether the new graph meets two particular conditions, and only keep it if both are satisfied. Before stating these conditions, we will establish some necessary definitions.

Set-orbits To understand what these conditions even are, we need to establish what *vertex set-orbits* are.

An *automorphism*, as usual, is a permutation μ of the vertices of G such that the graph G^μ obtained by applying the permutation to G is once again the original graph: $G^\mu = G$.

We say that two vertex subsets $X, Y \subseteq V(G)$ are in one *set-orbit* if there exists some automorphism μ on G such that $\mu(X) = Y$.

Note that when $|X| = |Y| = 1$ this just gives us the usual definition of vertex orbits.

Canonization Another definition that needs to be established before the conditions can be is graph canonization.

Let us say we have some graph-labeling process that, given a graph G, labels the nodes from 0 to $|V(G)| - 1$. We call such a process a *canonization* if, for any two isomorphic graphs, the newly labeled graphs are identical. We will refer to the labeled graph obtained from the canonization of G as the *canonical labeling* of G (according to our canonization).

The unlabeled generator's conditions We can finally define the special conditions that make McKay's unlabeled generator function properly. Let G' be the new graph, v its latest-added vertex, and let us also fix some strict ordering on the subsets of $V(G')$.

(1) The neighborhood of v is the minimal representative of its set-orbit in $G' - v$ according to the fixed ordering,

(2) v and w are in the same orbit in G', where w is the vertex that gets the largest label in the canonical labeling of G'.

We can show that with these conditions required, no two isomorphic graphs are ever generated.

Let us assume that G_1 and G_2 are isomorphic but are both kept (with latest vertices v_1 and v_2), and are the smallest such graphs. Then the parents of these must also be isomorphic because of condition (2), since there is an isomorphism μ between G_1 and G_2 such that $\mu(v_1) = v_2$.

Because of our assumption, these parent graphs are not only isomorphic, but exactly the same: $G_1 - v_1 = G_2 - v_2$, but then the neighborhoods of v_1 and v_2 must be in one set-orbit, so due to condition (1), only at most one of them could have been kept, leading to a contradiction.

We can also see that we still generate each unlabeled graph at least once: given an unlabeled graph G of order n, label it canonically, then remove the vertex with label $(n - 1)$ to get F. It is clear that if F is examined during generation, G will be kept as its child when abiding by the conditions.

Thus, inductively, each unlabeled graph of order at most n will pass conditions (1) and (2) exactly once during generation.

Things we need to compute Clearly, there are a few non-trivial things we need to compute for this to work:

- A canonical labeling for a given G,
- Decide whether $u, v \in V(G)$ are in one orbit in G,
- Decide whether $X \subseteq V(G)$ is minimal in its set-orbit.

A brute-force approach to achieve these may be to permute the vertices of G in all $n!$ possible ways, keeping the lexicographically minimal form of the graph according to some fixed graph-ordering, while making note of any automorphisms found (we need these to compute set-orbits).

This approach would clearly take way too long.

McKay developed a canonization algorithm [1] that achieves these computations rather efficiently.

The idea of McKay's algorithm The basic idea is to use graph degree properties that are entirely independent of the actual labeling to reduce the scope of examined labeled forms, so that isomorphic graphs still choose from the same pool of labeled graphs, and the lexicographically minimal one will still be the same between them.

This is achieved via cell refinement on ordered vertex-partitions.

An *ordered partition* is exactly like a regular partition, except that the partition cells have a set order. We say an ordered partition of the vertices of G is *equitable* if, for any partition cells V_1 and V_2 (these can be the same), $d(v, V_2)$ is the same for all $v \in V_1$.

Note that this property does not depend on the labeling of G, nor the order of vertices within cells. In [1], McKay shows a method to *refine*, that is, systematically turn a non-equitable ordered partition into a coarsest finer equitable one in a way that also only depends on the "shape" and order of the current partition cells.

More precisely, given graphs G_1 and G_2 with a μ isomorphism between them, and their ordered partitions $V_1^1, \dots V_k^1$, $V_1^2, \dots V_k^2$ such that $\mu(V_i^1) = V_i^2$, this cell-matching property is kept even after the refining process, although not necessarily with the same μ.

A summary of the algorithm We make a search-tree of ordered partitions. In the root, we have the unit partition, which has all vertices of G in one cell. Note that this satisfies the earlier cell-matching property for any two isomorphic graphs.

In the tree, an ordered partition's children will always be finer versions of its parent (that is, some cells are replaced by ordered partitions of themselves). The leaves of the tree are ordered partitions with all singleton cells, which define permutations on the vertices of G.

In the brute-force approach, we would simply take the partition we are at now, pick the earliest non-singleton cell V, and create a child-partition for each $e \in V$ by replacing V with $\{e\}, V \setminus \{e\}$. This would lead to every possible permutation of vertices being reached as a leaf.

In McKay's algorithm, we do the same thing, except we always refine the partitions before generating its children.

Following the earlier line of thought, reducing the search tree in this way will not cause problems with canonical labeling, since the leaves given by two trees of isomorphic graphs will yield permutations resulting in identical graphs, although these may be reached in different orders.

Though not enough to make the canonization algorithm polynomial (consider K_n), this refining step drastically reduces the number of leaves that need to be examined in the vast majority of cases. There are of course other optimization tricks that can be employed here and there, such as *pruning* the tree based on already found automorphisms, but refinement is the key step that elevates this algorithm above the brute-force approach.

Computing set-orbits We still need to address how we are going to check the set-orbit conditions, but luckily, McKay's algorithm [1] also gives a handy way to compute those whilst the canonization algorithm runs.

We are going to compute the minimal representatives of the set-orbits of G as we canonize it before generating its children, and when we do, they will all refer to the set-orbit values of their parent.

The key is to find enough automorphisms to form a generator of G's automorphism group ($\mathrm{Aut}(G)$). To see why this is helpful, imagine a graph H of order 2^n, with its vertices being every possible subset of $V(G)$.

Let $\{a_i\}_{i \in I}$ be a generator of $\mathrm{Aut}(G)$. Now for the edges of H, we say that for any $X, Y \subseteq V(G)$, $XY \in E(H)$ exactly if $a_i(x) = y$ for some $i \in I$. With this construction, it follows by definition that the subsets X, Y are in one component of H exactly when they are in one set-orbit.

From here, deciding whether two vertices are in one orbit, or picking a lexicographically minimal representative in a set-orbit is simple.

Of course, no such graph is ever created, only implicitly, as for every a_i, we check $a_i(x)$ for every $x \subseteq V(G)$, and update "component" information from there.

Clearly, this results in $|I|2^{|V(G)|}$ "component information" update steps, as well as $\mathcal{O}(2^{|V(G)|})$ bits of memory to store for the generator, which is an exponential amount of required space. Thankfully, the vast amount of generated graphs becomes a computing problem way before this can, although it is something to consider in the long-term.

Finding a generator for Aut(G) When going through the ordered partition search tree, we find our canonical labeling by checking whether the currently examined leaf yields a better graph than the permutation we have stored as the "best one", initialized by the first leaf we reach.

Similarly, in the case of automorphisms, we check for permutations that yield equivalent graphs.

We fix a particular permutation p_0 of the vertices in the beginning, specifically the first one we acquire during the run of the algorithm. Then, for every other q permutation we find as we find more leaves, we check whether G^{p_0} and G^q are identical. If so, we have found an automorphism of G, namely $p_0 \circ q^{-1}$.

Given Corollary 2 of [1], which says that if a permutation p_0 is acquired as a leaf of the search-tree, and a_0 is an automorphism of G, then $p_0^{a_0}$ is also a leaf, (trivial for the brute-force tree, not difficult with refinement), this will indeed get us enough automorphisms to form a generator.

In the version of the algorithm described above, we will add as many automorphisms to our "generator" as many times we find a permutation that leads to the first labeled graph found. This might be a lot more than necessary, but an efficient way of deciding whether a new automorphism is independent of the ones we already have is not readily available.

When pruning the search-tree further, as is done in McKay's optimized version of the algorithm [1], redundant automorphism finds may be avoided, although one always needs to be careful to accommodate for automorphisms potentially lost when abandoning given branches of the search-tree.

Vertex-hereditary properties Aside from offering an opportunity to filter out isomorphisms, this type of DFS-style graph generation also gives us a convenient way to make our graphs adhere to *vertex-hereditary properties.*

We say that a graph-property $P : \{G : G \text{ is a graph}\} \to (0, 1)$ is vertex hereditary if, for any graph G for which $P(G) = 1$, it follows that for any subgraph $F \subset G$ we have $P(F) = 1$.

Some classic vertex hereditary properties include among others:

- degree-boundedness,
- K_3-freeness,
- C_4-freeness,
- edge number boundedness,
- bipartiteness.

It follows trivially that if $P(G) = 0$, then for any supergraph $H \supset G$, we have $P(H) = 0$. Using this observation, we can modify our generator once again to only generate graphs that satisfy P.

Aside from the special conditions established earlier, we also require that a graph G satisfy P, that is, $P(G) = 1$, before we let it generate children. Given the definition of vertex-hereditariness, it is clear that this will remove exactly those graphs from our output that violate the property P.

Other properties There are other properties we may often want to require from our generated graphs which are not vertex-hereditary. Properties such as connectedness and degree lower-boundedness may come to mind.

These can of course be filtered out by simply checking for them before the graphs are output at the last level, but we can sometimes do better.

A non-vertex-hereditary property P can be made into such by defining $P'(G)$ as "either $P(G) = 1$, or there is a supergraph $H \supset G$ of order at most n for which $P(H) = 1$". Sometimes, especially when additional constraints are known for the upcoming generated vertices, we can notice when this P' becomes impossible to fulfill for any H supergraph of order at most n. This will be done with the help of *graph filters.*

Graph filters Filters run alongside the generator with their own subroutines and memorized data. Using filters lets us keep track of potential properties more efficiently than just by checking for them every time.

For example, if we want to keep graphs with degrees at most k, we do not want to calculate every individual vertex's degree every time we decide whether to keep a graph or not.

A degree-at-most filter can keep track of all degrees by simply incrementing $d(v)$ saved values whenever the vertex v is added, and decrementing them when we step back in the recursion.

Doing this, the filter can immediately notify the generator if a vertex exceeds the degree bound, and we can move on to the next potential graph.

Advantages of filters First of all, creating a filter for a new graph-property only involves writing graph-theory related code, so users can create their own specific graph filters without having to understand all the ins and outs of how the generator operates.

Secondly, when we have multiple filters running in the generator, they can help each other out by giving each other information about the limitations in place for the rest of the vertices to be generated.

For example, if we have a degree-at-most-3 filter in place, as well as a connectedness filter keeping track of the number of components, we may at some point notice that there are still more than $1 + 2k$ components left with only k vertices left to go, so clearly, no supergraphs of order at most n will satisfy both conditions.

Shared data among filters can generally decrease the number of visited graphs, and it can even let us detect violation of properties that are not purely vertex-hereditary multiple levels before the final level, saving computing time.

In general, the more graph-properties filtered out, the less graphs need to be generated, which means that relevant graphs of higher order can be generated within a reasonable amount of time.

Our work in this semester My main contribution to the project this semester was fully implementing the mentioned canonization algorithm along with its set-orbit computing functionality, helping in getting the unlabeled generator to work by utilizing it, as well as helping my team understand the mathematics of how and why the algorithms work (or do not work when something goes wrong).

While the pseudocode and description of the algorithms are provided in McKay's papers, the actual implementation is a fairly old-fashioned code written in C that is not really in support of user customization, so we implemented it in a more approachable, object-oriented C++ program.

With my help, we made a version of the graph-generator that:

1. Accomplishes what is described above with the help of the implemented canonizer,
2. provides support for custom canonizers,
3. provides support for filters.

The base for user-customizable filters is finished and working, and a handful of concrete filters have already been made and tested in the actual generator.

Future plans Now that we have a base program that appears to be functional, the main focus is on improvement and optimization, both in runtime and memory, as even though our base generator works as it should, it is not nearly as efficient at doing so as McKay's Nauty-based generator program [3].

Aside from better technical solutions, we are also planning to work out a more general way for filters to efficiently share useful information with each other, helping decrease the amount of examined graphs during generation.

References

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

[2] Brendan D. McKay. Isomorphism-free exhaustive generation. *Journal of Algorithms 26*, pages 306–324, 1998.

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