Generating Small Graphs up to Isomorphism

Nagy Szabolcs

2023 1st semester, Math Project I.

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, found at [3].

Generating the graphs For now, let us just say we want to generate all graphs with $n \in \mathbb{N}$ nodes up to isomorphism. The following algorithm, loosely described in [2], accomplishes this efficiently:

```
gen($n$):
  $G \leftarrow (\{v_1\}, \emptyset)$
  gen_req($G, n$)
gen_req($G, n$) :
  $k \leftarrow |V(G)|$
  for all $P \in 2^{\{1,\ldots,k\}}$ do
    $G' \leftarrow (V(G) + v_{k+1}, E(G) + \{v_i v_{k+1} : i \in P\})$
    if LexIso($G', k+1$) and GreatOrb($G', k+1$) then
      if $k + 1 = n$ then
        Output $G'$
      else
        gen_req($G', n$)
      end if
    end if
  end for
```

Basically, we are exploring a search tree of possible graphs, where each level connects the next node with some collection of nodes already processed.

Here, LexIso(G, k) returns true if and only if G is the "lexicographically smallest" representative of its isomorphism class in the $2^{\{1,\ldots,k\}}$ cycle, and GreatOrb(G, k) returns true if and only if v_i and v_k are in the same orbit of G, where v_i is the node that gets the greatest index when the nodes of G are canonically labeled.

Computing these two values is no simple task, but possible, and is massively aided by a canonization algorithm in the implementation.

Simple induction can be used to show that the algorithm above is going to output each graph exactly once, with respect to isomorphism.

If we not only want to generate all graphs, but specifically want a certain family of graphs to be generated, such as connected, triangle-free or degree-bounded graphs, this algorithm can be modified to filter out unwanted graphs early on, meaning that the desired graphs can be generated faster.

Canonization Let us say we have some graph-labeling process that, given a graph G, labels the nodes from 1 to $|V(G)|$. We call such a process a *canonization* if, for any two isomorphic graphs, the newly labeled graphs are identical. Such a labeling is called the canonical labeling.

Canonization is a key tool used by the graph-generating algorithm given above, so we want to implement it effectively.

The description of a canonization can be found in [1], and a working implementation can be found at [3], as well as the graph-generating algorithm built on top of it.

The canonization algorithm of [1] in broad terms We aim to choose the "lexicographically smallest" labeling according to some ordering of labels, such as taking the adjacency matrix of the labeled graphs and treating them as base-2 numbers, but considering every possible permutation of nodes would take way too long, so we try to cut down on the number of checked labelings by creating a search tree of ordered partitions.

At the root of the tree is the unit partition, where every node is in one cell, and the children of a node have finer versions of their parents' partitions. The leaves of the tree have ordered partitions with only singleton cells, giving a labeling.

We do this in a manner such that we can notice that a branch is going to yield no better labeling than what we found so far, cutting out a lot of unnecessary labelings.

Our work in this semester We have spent a lot of time thoroughly studying the complex canonization algorithm described in [1], the graph-generating algorithm described in [2], as well as the C code of [3], so we now understand what the program needs in order to work efficiently. We have already begun re-implementing the program in C++. My main task in this semester was comprehending the mathematics behind the algorithms, and implementing the canonization.

Future goals We are going to create the base for the customizable filters that can be given to the generator, and implement the graph-generator that can properly use all of the other tools we have made.

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/`.