An heuristic for graph isomorphisms

Nguyễn Lê Thành Dũng Blanchard Nicolas Koliaza

December 12, 2012

Foreword

The problem considered is the graph isomorphism (GI) problem : given two graphs G and G', can one compute an isomorphism between them. The original goal was to implement the Weisfeiler-Lehman heuristic in the C programming language, but we chose to implement our own heuristic (named PN for Path-Neighbour) to test its performance. The main goal was to find an heuristic which would behave polynomially on the a bigger subset of problems, even if the polynomial degree increases.

1 Algorithmical Considerations

We adopt the notations of graphs with n vertices and m edges. $G(n, p)$ denotes the graph generated through the Erdös-Rényi model with n vertices and probability p for each edge.

There is a strong link between GI and algebra : finding an isomorphism is the same as finding a basis in which the adjacency matrix of G becomes that of G'. Testing all solutions is equivalent to testing all permutations and a naïve algorithm would take $n!$ operations. The GI problem lies between the complexity classes P and NP and Schöning [4] proved that it is not NP-complete unless $P = NP$. However, no polynomial solution has been found and the best deterministic algorithm so far is due to Eugene Luks and runs in $2^{O(\sqrt{n\,log n})}$[1]. However some efficient heuristics are available, including *nauty* and *conauto*, which runs in $O(n^5)$ with very high probability for any graph in $G(n, p)$[2]. In our algorithm and test we often consider multi-graphs, as the hard cases are much easier to generate for multi-graphs and the difference has no impact on our heuristic. Finally, we tried to make an heuristic which would behave well when launched on p different graphs to check if any two are isomorphic.

2 The Heuristics

2.1 The Weisfeiler-Lehman heuristic

The original Weisfeiler-Lehman (WL) heuristic works by coloring the edges of a graph according to the following rules :

- We begin with a coloring that assigns to every vertex the same color (this is the 1-dimensional version).
- At each pass, the color of each vertex is determined by the number of neighbours of color c for each c.
- After at most n passes, the colors don't change anymore.

These rules actually only produce a certificate of non-isomorphism. To construct an isomorphism using WL one uses backtracking coupled with the fact that the image of a vertex has to be of the same color. It is easy to see that two isomorphic graphs behave in the same way when subject to WL coloring, but the converse does not generally hold. Notably, some special classes of graphs make backtracking omnipresent, e.g. in k-regular graphs, thus leading to time complexity up to $O(n^n)$. The heuristic can be improved by changing the initial coloring of the vertices, but we shall only study this case.

2.2 Another heuristic

2.2.1 The idea

PN is based on the following property :

Let $N_k(x)$ be the number of neighbours at distance exactly k from x, and $P_k(x)$ the number of paths of length k starting from x, then if f is an isomorphism between G and G', $N_k(x) = N_k(f(x))$ and $P_k(x) = P_k(f(x))$. Thus, by computing the different N_x and P_x we can prune the search tree and limit the possibilities. We name the array of couples $P_k(x), N_k(x)$ for k between 1 and n PN(x), and compute an array containing PN(x) for each x, obtaining the PN arrays.

2.2.2 Structure of the algorithm

The algorithm we use actually incorporates multiple testing phases to quickly eliminate easy cases. It can be decomposed in the following steps :

1. Lecture and choice of data structure
2. Primary test phase
3. Construction of each PN-array
4. Sorting of each PN-array
5. Comparison of the PN-arrays
6. Comparison of the neighbours of compatible vertices
7. If possible construction of an isomorphism by refined bruteforce

2.2.3 Details

We keep the list of image-possibilities for all vertices in a matrix form. When one line or column becomes full of zeroes, we have a negative certificate. The matrix actually corresponds to classes of an equivalence relation, and every class of cardinal c has at most $c!$ possibilities of isomorphism. When the product of the $c!$ becomes small, bruteforce becomes possible. The goal of the PN matrix is to separate the vertices in as many classes as possible, but bruteforce can sometimes be used earlier, as is the case in the primary test phase. However, when one computes the PN arrays, it is not always enough, and so we use a technique from WL and compare the immediate neighbourhood of each vertex to see if it is compatible with all of its images, to prune some more.

2.2.4 Primary test phase

The problem with the naïve algorithm is that the number of possibilities groes exponentially in n, however, when one considerons permutations which are composed of one transposition, there are at most n^2possibilities and each take at most m operations. The first test phase uses different simple techniques to quickly find a certificate of non-isomorphism or an easy isomorphism when doing so is possible. The tests are run in this order :

1. Check for equality between the matrices
2. Check size and number of connected components
3. Compare the list of degrees and record the number of possibilities
4. When the number of possibilities is small ($O(n^2)$) use bruteforce
5. Still check for simple transpositions when it is not the case

2.2.5 Details and optimization

The choice of data structures varies between adjacency matrix and list, and sometimes both, and allows us to take advantage of faster algorithms on sparse graphs using lists, while allowing us to know in O(1) if two vertices are linked.

The construction of the N-array is generally the most time-expensive task, it is mostly done by multiplication of the adjacency matrix (or by equivalent operations on lists). Sorting the N-array is easily done with a heapsort. The construction of the isomorphism is done by trying to assign to an x an y such that$N(x) = N(y)$ and backtracking when unsuccessfull.

Some code optimizationsmight be added : the construction of N-arrays by matrix multiplication can be parallelized easily, and multi-threading could be used to do so. It can also be used it for sorting the arrays. However we didn't use Strassen's algorithm because for the matrix size considered ($n \leq 500$) the gain is very small and mostly compensated by the overhead.

3 Tests and Benchmark

We have included different basic test generation programs, depending on the model :

3.1 Random generation

- The Erdös-Rényi $G(n, p)$ model with probability p for each edge works in general quite fast because the probability that two graphs are not isomorphic (and trigger a certificate) is very high.
- The similar model with $G(n, M)$ behaves the same way.
- A model to generate random k-regular multigraphs that generates k random permutations of n vertices and links each vertice of the initial array to each of its images in the resulting permutations.

All other methods so far for generating k-regular graphs (and not multigraphs) are probabilistic except for very small k, and when $k \geq log(n)$ have exponential expected time [3], so we restricted ourselves to those three models, as our algorithm should work as well on multigraphs as on normal graphs.

3.2 Isomorphic graph generation

To generate two isomorphic graphs is actually really easy : as two graphs are isomorphic if and only if their adjacency matrices are similar, one has to compute a random permutation (which corresponds to a change of basis), and to apply it to any graph to obtain an isomorphic copy. We generate those from all three different models of random graphs already implemented.

3.3 Benchmark

4 Comparison with Weisfeiler-Lehman

4.1 Proof

Our goal here is to show that the subset of problems solved in polynomial time by WL is strictly included in the subset solved by our heuristic. We shall consider an extended version of WL (EWL) to facilitate the proof, where colors aren't redistributed among $[1; n]$ but are instead an injective function of the vertex's previous color and the multiset of its neighbour's colors into the set of colors (which is not anymore included in $[1; n]$). Two cases allow WL to reduce the number of possibilities and hence have a polynomial runtime : the first is when the colorings of G and G' are incompatible, and the second when there is a color shared by an unique vertex in each graph.

We must show three properties :

- When the graphs are isomorphic, WL, EWL and PN compute the isomorphism, and produce a negative certificate in the other case (correctness)
- When WL finds two incompatible coloring, then so does EWL, and PN also gives a negative certificate (polynomial negative certificate)
- When WL colors a vertex in an unique color in both graphs, then EWL does the same, and our algorithm also recognizes that those two vertices must be linked if the graphs are isomorphic (polynomial isomorphism)

The first is immediate, because the operations we use are invariant through permutations of the vertices, hence if the two graphs are isomorphic it might take exponential time but will surely end. The two other properties trivially hold for extended WL, and we must show that it implies that they hold for PN. However, there is an advantage of EWL over WL, because the coloring is unique for the two graphs (it being injective), so if there is exactly one vertex in each graph colored by C, then the isomorphism must assign one to the other (problems may arise in WL because the coloring is not necessarily injective). This gives an easy proof of the third depending on the second, because the fact that a vertex is colored uniquely in G and G' is implied by the fact that it is colored differently from every single other vertex. Hence we must only prove the second property.

We shall proceed by induction to prove that if EWL colors a vertice in G and another in G' in different colorsat run k, then PN also allows us to differentiate those two vertices.

At the first run of the coloring, every vertex is colored by its degree so it is trivially true.

Now let us suppose that the property holds at run k, and show that it holds at run k+1 :

Consider two vertices V and W (one per graph) that had the same color for all previous runs, but which are colored differently at run k+1. Then their neighbourhoods are incompatible, which means that the multiset of colors is not the same, so there is a color c such that there is one more vertex of color c in N(V) than in N(W). However, those colors were attributed at run k, so it means that when comparing the lists of neighbours of V and W in PN, those lists will be incompatible by hypothesis. Hence V and W can't be linked in PN, which ends the induction.

4.2 Complexity analysis

As the GI problem isn't known to be in P, it is not absurd to have a worst running time of $O(n!)$. However, we can go into details and see that in most cases the real running time is generally much lower.

The reading phase takes at least $O(m)$, but is implemented in $O(n^2)$ as we use matrices.

The first test phase consists of a list of at most five tests of increasing time complexity, to quickly solve cases of increasing difficulty.

The first three tests run in $O(n^2)$ although they could be implemented in $O(m + n * log(n))$.

The two other tests take at most $O(n^4)$ but only $O(n^2 m)$ when run on lists. However, even though this polynomial is of high degree, it does not effectively take as much time because most of the tests fail immediately, hence its presence before PN has an utility.

The most time-consuming phase is the array generation. It does n multiplications of $n * n$ matrices for each graph, which takes $O(n^4)$ and a sort after that (which takes at most $O(n^2 log(n))$). We could have used Strassen to decrease that cost but the overhead on small matrices reduces its interest. However, using lists in case of sparse graphs would not really decrease complexity (except in graphs with very bad expansion properties).

The next phase is the comparison of the arrays. It takes at most $O(n^2)$, and when we also check compatibility with the neighbours we obtain $O(n^3)$.

If we haven't found an isomorphism or a negative certificate by then we launch a bruteforce on the remaining possibilities, which at worst runs in $O(n!)$.

A quick analysis of WL gives a higher bound (in polynomial cases) of $O(mn^2)$ when implemented with ajdacency lists although the expected running time is lower. We can then separate the problems in five categories :

1. The very simple ones that are solved much faster by PN thanks to the primary test phase
2. The medium-easy cases that are solved by PN but not the primary test phase, and where WL is asymptotically better
3. The medium cases that are solved by both PN and WL in polynomial time but in $O(n^4)$ and where they differ only by a constant
4. The medium-hard cases that are solved polynomially by PN but exponentially by WL
5. The hard cases where both algorithms behave exponentially.

We see that we could improve PN by adding WL between the first test phase and the array generation, but the overhead would once more increase.

References

[1] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 171–183, New York, NY, USA, 1983. ACM. 1

[2] José Luis López-Presa and Antonio Fernández Anta. Fast algorithm for graph isomorphism testing. In *SEA*, pages 221–232, 2009. 1

[3] Brendan D. McKay and Nicholas C. Wormald. Uniform generation of random regular graphs of moderate degree. *Journal of Algorithms*, 11(1):52–67, March 1990. 3.1

[4] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1988. 1