An heuristic for graph isomorphisms

Nguyễn Lê Thành Dũng   Blanchard Nicolas Koliaza

December 15, 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 a bigger subset of problems, even if the polynomial degree increases. 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.

Contents

1 First Considerations

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 naive 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 $2^{O(\sqrt{n\,logn})}$[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. We then make one random choice before coloring again, using backtracking.

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 The PN 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 $PN(x)$ the array of couples $P_k(x), N_k(x)$ for k between 1 and n, 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. Input parsing and choice of data structure
2. Primary test phase
3. Construction of each PN-array
4. Sorting and comparison of the PN-arrays
5. Comparison of the neighbourhoods of compatible vertices
6. If possible construction of an isomorphism by refined bruteforce

2.2.3 Details

We keep a list of equivalence classes for the vertices of both graphs. Each information we gather with the different tests allows us to refine the separation into classes, and when they are not compatible to produce a certificate of non-isomorphism. 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, or during the generation of the PN matrices sometimes. However, when one computes the PN arrays, it is not always enough, and so we use a technique from WL and compare the neighbourhoods of k-order 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 naive algorithm is that the number of possibilities grows exponentially in n, however, when one considers permutations which are composed of one transposition, there are at most n^2 possibilities 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

3 Tests and performance

3.1 Random generation

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

- 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. There is a deterministic multigraph version and a probabilistic version in the other case.
- 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 mostly 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 Performance

We checked the performance of PN on the three types of non-isomorphic graphs and here are the results (time in seconds), including parsing time (On an intel core i7 2.3Ghz with ubuntu 12.10)

n in G(n,p)	p	time
100	0.05	0.002
1000	0.05	0.039
10000	0.05	13.9
100	0.5	0.008
1000	0.5	1.4
2000	0.5	10.5

n in G(n,M)	M	time
100	2000	0.007
1000	50000	0.087
10000	200000	0.22
10000	500000	0.84
10000	2000000	9.1
20000	2000000	5.0

n	k-regular	time
50	4	0.016
200	4	0.48
500	4	8.2
50	12	0.012
200	12	0.49
500	12	8.2

We saw that in the G(n,p) and G(n,m) the work is most often done by the primary test phase, and in k-regular by the first two multiplications of the matrix. The same tests run on multigraphs show very similar results:

n in G(n,M)	M	time
100	2000	0.006
1000	50000	0.090
10000	200000	0.24
10000	500000	0.85
10000	2000000	9.3
20000	2000000	5.4

n	k-regular	time
50	4	0.018
200	4	0.48
500	4	8.0
50	12	0.016
200	12	0.48
500	12	8.0

Finally, run on isomorphic graphs :

n in G(n,p)	p	time
100	0.05	0.092
300	0.05	1.8
500	0.05	8.5
100	0.2	0.065
300	0.2	1.8
500	0.2	8.5

n	k-regular	time
50	4	0.029
100	4	0.026*
500	4	43
50	12	0.39
100	12	5.9
500	12	3930

The running time in the k-regular case varies a lot (the * case once took about 6s). PN proved indeed useful as it reduces drastically the number of possibilities around the fifth multiplication. The use of neighbourhood simplification at the end also reduces the number of possibilities quite fast. However, when launched on bigger graphs, it is still slow.

4 Comparison with Weisfeiler-Lehman

4.1 Proof

Our current algorithm incorporates elements from WL at the end, to facilitate this proof. We had an initial proof that did not rely on that part but we realized it was flawed one day before the deadline for this report, so even though that bit of code might never be used in practice it allows us to show that the subset of problems solved polynomially is strictly greater than the subset solved by WL. It might be possible to show that with minimal modifications our algorithm doesn't need that part, but we had no time to try to do so.

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 neighbours' 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 colors at run k, then PN also allows us to differentiate those two vertices before examining neighbourhoods for the k-th time.

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 at run k, 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 last takes 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.

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 $O(n^3 log(n))$.

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 adjacency lists although the expected running time is lower. We can then separate the problems in four 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-hard cases that are solved polynomially by PN but exponentially by WL
4. The hard cases where both algorithms behave exponentially.

Also, every information we gather doing initial tests can be used by the following ones, which wouldn't be the case with WL (it would use the previous ones but would have no impact on the following ones).

4.3 Differences when run on k graphs

One advantage of PN is when it is run on k different graphs, as the PN matrices can then be sorted in $O(k * log(k))$ time, instead of launching a comparison algorithm k^2 times. Also the generation of PN matrice can easily be parallelized. It is an advantage over WL as the latter is very hard to parallelize in such fashion. When run on only two graphs however, we can interlace the sorting-and-comparing phase with the generation of the PN matrix, thus greatly reducing the running time (as most cases can be solved by only computing the first few multiplications of PN).

5 Implementation and Optimization

5.1 Implementation problems

The biggest problem was that the numbers in the PN matrix quickly grow out of proportions (one can bound them by n^n but this bound might be reached), so one has to use multiplication modulo p, and we chose a prime number p close to 2^{61} to be able to do bit-shift easily while having low probability of collisions. We also added another version where the modulo is done with $p < 2^{32}$, that is much faster although the probability of collision is also higher. This means that our implementation is only probabilistic, and there is a small chance that it behaves exponentially where another implementation would not. Then again, it might never happen, depending on how collisions happen, but we have not looked into this possibility.

The sorting was also source of problems because manual implementations behave quite slower than the ones included in the libraries so we used qsort_r to increase performance, which is not a standard function.

Finally, we had to use some goto in the code, to be sure to free the memory. One could implement PN without them, but it would necessitate changing the whole control flow.

5.2 Function Optimization

As with many graph algorithms, PN can be improved by using adjacency lists in the case of sparse graphs. However, that seldom changes the asymptotic complexity, as the PN-array generation can only benefit from it in case of graphs with very poor expansion properties. The list representation is mostly used in the primary test phase, where it allows us to have tests in $O(m)$ instead of $O(n^2)$. Optimization on the matrix operations cannot really be done in practice as Strassen's algorithm has no interest on the matrix size we generally consider. However we took care of memory locality in their representation, which might improve the performance in practice.

5.3 Parallelism

The advantage of the PN-arrays is that they can be generated in a completely independent fashion, and so are easy to parallelize. This is an advantage of PN over WL as the threads do not have to communicate, while the latter has to do some comparisons between each coloring at each run, requiring constant interaction between threads. In practice the version that runs on two graphs is monothread and interlaces the equivalence class creation with PN to run faster, but this improvement couldn't be used in a parallelized version.

5.4 Potential improvements

Due to time constraints we couldn't implement as many functions as we hoped to, but here's a list of potential improvements :

- Optimize the mulp function, which is the bottleneck of the algorithm. This has been done partially with mulp_small at the price of a small probability of going exponential. A detailed analysis of the cases when this could arise might even allow us to use naive multiplication modulo 2^{64} but it isn't sure.
- Parallelization inside the matrix multiplication could yield good results. We ran some tests on how well parallelization works on our machine (with the sorting algorithms for PN), with and without multithreading, and we obtain increase in speed up to a factor 4 plus a 30% increase when using multithreading.
- Interlacing the neighbourhood function with the PN generation would greatly improve the speed, but would complexify the proof.
- We hesitated whether to add a fifth primary test that would try for all simple transpositions, which runs quite fast. That kind of method might come in handy when the two graphs are very "close" (i.e. we could imagine that there is a perturbation on the order the vertices are received).
- A very efficient Strassen multiplication might also be useful, mostly if we want to run PN on bigger graphs.

5.5 Files and usage

All the files can be found at http://github.com/koliaza/Heuristitique. The files can be separated into three categories (plus the pdf and tex files) :

- Basic functions : io.c, util.c, matrix_mod.c, partitions.c
- Graph functions : graph.c, random_graph.c, pn_heuristic.c
- Global functions : main.c, test_gen.c

Commands : one uses main, with the option –verbose, on either two filenames or a directory (for k graphs). test_gen can be called with no arguments to see the exact syntax.

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