PJ, an algorithm for Contest BioGraph'14

Jean-Christophe Janodet[1] and Frédéric Papadopoulos[1]

University of Evry, IBISC, F-91027, France
{janodet,fpapadopoulos}@ibisc.univ-evry.fr

Abstract. Graphs are usual data structures in Computational Biology and Chemistry. Indeed, molecules and many biological structures can easily be represented as graphs and stored in large databases. For this reason and other applications in Image Analysis for instance, a lot of exact and approximate matching algorithms have been proposed to address the problem of searching for patterns in large graphs. In this paper, we describe an exact algorithm, called PJ, tailored to participate to Contest BioGraph'14. We develop both theoretical and empirical arguments that explain the features of this algorithm.

1 Introduction

Graph matching is an old problem, that has been addressed with many techniques and heuristics (see [CFSV04a,FPV14] for surveys). Several algorithms are now well-established, in particular VF2 [CFSV04b] and Lad [Sol10]. See [CFV13] for a comparison.

In this paper we to introduce a new algorithm, called PJ, that tackles the problem of finding all the occurrences of a query (or pattern) graph in a target graph. PJ stands in the category of exact pattern matching algorithms. Nevertheless, PJ was developed in the framework of Contest BioGraph'14, and is doubtless optimized with respect to the databases provided by the organizers.

More precisely, there were 3 databases with similarities and proper characteritics. Concerning the similarities, all the graphs are sparse, labeled, simple, undirected (but for rare cases) and connected. Concerning the dissimilarities:

Molecules: This base is made of 50 query (or pattern) graphs, whose size varies from 4 to 64 vertices with average degree 2, and 10 000 target graphs with 8 to 100 vertices with average degree 3. The labels are atomic symbols such as C for Carbon, H for Hydrogen, ... Rare symbols such as Ag (Silver) or P (Phosphorus) appear in many graphs. Notice that the arity of each symbol (*e.g.*, a Carbon should have 4 neighbours) is not preserved in the graphs, perharps to keep them simple (that is, with no multiple edges).

Proteins: 60 queries with 8 to 256 vertices, and 300 targets with 535 to 10000 vertices. All the vertices have average degree 2, and the labels are the same as those of base *Molecules*. Nevertheless, as the size of the graphs is quite large, they are often close to chain graphs: the average treewidth is < 2.

Contact Maps: 60 queries with 8 to 256 vertices and average degree 19, and 300 targets with 100 to 733 vertices and average degree 24. The labels are the 22 amino acids, and within each graph, using the labels to distinguish the vertices is very discriminating.

Due to the important disparities between the bases, one possibility was to create several algorithms, each of them tailored to solve specific problems separately. However, we have decided to investigate a general procedure, unique for all the graphs, which may adapt itself to the characteristics of each graphs dynamically.

The theoretical background of our algorithm has its roots in some papers dealing with the isomorphism problem [BES80,Cod13] where the authors study the possibility to distinguish the vertices of a graph by using criteria such as the degree sequences and the distance sequences. They show that the labeling problem can be efficiently solved with high probability for random graphs with respect to several types of distributions over the graphs. This implies that the isomorphism problem itself can be solved efficiently on random graphs. PJ also uses the degree sequences as a criterion to compute the possible assignments of any query vertex.

In Section 2, we recall main definitions and properties about subgraph matching. Section 3 aims at describing the principle of the PJ algorithm; we develop the procedure and prove the correctness of the algorithm. Finally, Section 4 focuses on the heuristics used by PJ to prune the search tree and improve the efficiency. A comparison of PJ with other standard algorithms is still in progress and not presented here.

2 Preliminaries

Let $G = \langle X, A \rangle$ be a *directed graph*, where X denotes the set of *vertices* and A the set of *arcs* (that is, directed edges). We assume that the vertices are labeled, that is, there exist a finite set L, fixed and unique for all the graphs, and a mapping $lab : X \to L$ that assigns label $lab(u)$ to every vertex of u of G.

Given an arc $(u, u') \in A$, we say that vertex u' is a *successor* of vertex u, and that u is a *predecessor* of u'. As usual, we denote by $deg^+(u)$ the *outdegree* of vertex u, *i.e.*, the number of successors of u, and $deg^-(u)$ the *indegree* of vertex u, that is, the number of predecessors.

More generally, we say that vertex u' is a *k-successor* of vertex u if u' is a successor of a $(k-1)$-successor of u. We recursively define

$$\begin{cases} succ_0(u) = \{u\}, \\ succ_1(u) = \{u' : (u, u') \in A\}, \\ succ_k(u) = \{u'' : \exists u' \in succ_{k-1}(u) \wedge (u', u'') \in A\} \text{ for all } k > 1. \end{cases}$$

We similarly define k-predecessors and sets $pred_k(u)$.

Note that the k-successors of a vertex are different from its *k-neighbours*: The former are the vertices which are linked to vertex u with a sequence of arcs of

length k, whereas the latter are the vertices whose distance to u is k. *E.g.*, in the case of an undirected graph, if $(u, u') \in A$, then $(u', u) \in A$, thus $u \in succ_2(u)$, but u is at distance 0 from itself.

Let $Q = \langle X_Q, A_Q \rangle$ and $T = \langle X_T, A_T \rangle$ be two labeled digraph, respectively called the *query* graph and the *target* graph. We usually use u or u' for the query vertices (of Q), and v or v' for the target vertices (of T).

Definition 1. *We say that Q is a* subgraph *of T if there exists a mapping $h : X_Q \to X_T$, called a* matching function*, such that*

1. *h is an injective function: for all $u, u' \in X_Q$, $u \neq u' \Longrightarrow h(u) \neq h(u')$;*
2. *h preserves the labels: for all $u \in X_Q$, $lab(u) = lab(h(u))$;*
3. *h preserves the arcs of Q: for all $u, u' \in X_Q$,*
$$(u, u') \in A_Q \Longrightarrow (h(u), h(u')) \in A_T;$$
4. *h preserves the absence of arcs in Q: for all $u, u' \in X_Q$,*
$$(u, u') \notin A_Q \Longrightarrow (h(u), h(u')) \notin A_T.$$

This definition corresponds to that of *induced* subgraphs. If Condition 4 is skipped, we get the notion of *partial* subgraphs, which is out of the scope of Contest BioGraph'14.

Determining whether a query graph is an (induced or partial) subgraph of a target graph is an $\mathcal{NP}$-complete problem [GJ79], thus enumerating all matching functions is intractable. In order to get round of this problem, standard techniques consist in developing a search tree that explores all possible query vertex assignments until finding a solution.

Nevertheless, for efficiency reasons, the search tree is generally pruned by using one or several consequences of Def. 1. In our contribution to Contest BioGraph'14, we have been using the following ones:

Proposition 1. *Let $Q = \langle X_Q, A_Q \rangle$ and $T = \langle X_T, A_T \rangle$ be two labeled digraph, $h : X_Q \to X_T$ a matching function, and u, v two vertices such that $h(u) = v$. Then:*

1. *$lab(u) = lab(v)$, and $deg^+(u) \leq deg^+(v)$, and $deg^-(u) \leq deg^-(v)$;*
2. *$h(\underline{succ_1(u)}) \subseteq \underline{succ_1(v)}$ and $h(\underline{pred_1(u)}) \subseteq \underline{pred_1(v)}$;*
3. *$h(succ_1(u)) \subseteq succ_1(v)$ and $h(pred_1(u)) \subseteq pred_1(v)$;*
4. *for all $k \geq 0$, $h(succ_k(u)) \subseteq succ_k(v)$ and $h(pred_k(u)) \subseteq pred_k(v)$.*

Proof. The degree relations of Claim 1 is a direct consequence of Claim 2, which is itself an instance of Claim 4 for $k = 1$. Concerning Claim 4, Case $k = 0$ is straightforward. Suppose that $h(succ_k(u)) \subseteq succ_k(v)$ for some $k \geq 0$ and let $u' \in succ_{k+1}(u)$. There exists $u'' \in succ_k(u)$ such that $u' \in succ_1(u'')$. By induction hypothesis, we have $h(u'') \in succ_k(v)$. Moreover, as $u' \in succ_1(u'')$, we deduce that $(u'', u') \in A_Q$, so $(h(u''), h(u')) \in A_T$, that is, $h(u') \in succ_1(h(u''))$. Therefore, $h(u') \in succ_1(succ_k(v)) = succ_{k+1}(v)$. Finally, concerning Claim 3, let $u' \notin succ_1(u)$, then $(u, u') \notin A_Q$, so $(h(u), h(u')) \notin A_T$, that is, $h(u') \notin succ_1(v)$, thus $h(\overline{succ_1(u)}) \subseteq \overline{succ_1(v)}$. Notice that Claim 3 does not hold with k-successors if $k \geq 2$. □

3 The PJ Algorithm

3.1 Principle

Algorithm PJ aims at finding all the matching functions between a query graph Q and a target graph T. As the problem is $\mathcal{NP}$-hard, our procedure consists in constructing and exploring a search tree where we progressively build the solutions.

A node N of the search tree contains:

- a function $dom : X_Q \to 2^{X_T}$ that gives, for each query vertex, the set of target vertices to which it is or it may be assigned;
- a set $U \in X_Q$ of query vertices that have not been assigned yet;
- a partial solution, that is, a set S of couples (u, v) where u is a query vertex that has been assigned to target vertex v;
- the set of all complete solutions that have already been found at this point of the exploration of the search tree.

When exploring the search tree rooted by node N, PJ may face two situations. Either there is no more unassigned query vertex (that is, $U = \emptyset$); in this case, we shall prove that (complete) solution S is necessarily a matching function from Q to T. Thus this solution is kept in the set of all complete solutions, and PJ backtracks.

Or, PJ selects an unassigned query vertex $u \in U$, and creates, for each possible target vertex $v_i \in dom(u)$, a new node N_i where u is assigned to v_i in new partial solution. Then, in node N_i, PJ reduces the domains of the query vertices by taking into account new assignment (u, v_i) and its consequences. Finally, PJ recursively explores the subtree rooted by node N_i.

PJ backtracks if $dom(u)$ is empty when query vertex u is selected. Also, before recursion, PJ regularly checks if any domain is empty and in this case, stops the exploration and backtracks.

3.2 Main procedures

See Algorithms 1, 2 and 3.

Algorithm 1: PJ(Q,T)

Input: Two labeled digraphs Q and T
Output: The list of all the matching functions from Q to T

```
1 if ImmediateFailCriteria(Q,T) then /* see Section 4.5 */
2     return (∅);
3 else
4     N ← CreateInitialNode(Q,T);
5     ExploreTree(N,Q,T);
6     return (N.all_solutions);
```

Algorithm 2: CreateInitialNode(Q,T)

Input: Two labeled digraphs Q and T
Output: A node N, that is, a record whose fields are:
- unassigned: the list of query vertices that have not been assigned yet;
- domains: an array that gives, for each query vertex, the list of target vertices to which it may be assigned;
- current_solution: a list of couple (u,v) where u is a query vertex u, and v is its assignment in graph T;
- all_solutions: the list of matching functions that have already been found.

```
1 N ← new node();
2 N.unassigned ← X_Q;
3 N.current_solution ← ∅;
4 N.all_solutions ← ∅;
5 for u ∈ X_Q and v ∈ X_T do
6     if lab(u) = lab(v) and deg⁺(u) ≤ deg⁺(v) and deg⁻(u) ≤ deg⁻(v) then
7         N.domains.(u) ← N.domains.(u) ∪ { v };
8 N.domains ← RefineInitialDomains(N,Q,T) /* see Section 4.1 */;
9 return N;
```

3.3 Correctness of PJ

Before describing the heuristics that are used to prune the search tree, we show that PJ is correct, that is, all the solutions returned by PJ are effective matching functions.

This question is unavoidable since in lines 1-2 of Procedure ExploreTree, as soon as all the query vertices have been assigned to target vertices, we do not check whether the current solution satisfies the Conditions of Definition 1 before we add it to the set of complete solutions.

Theorem 1. *When all the query vertices have been assigned to target vertices, any (complete) solution S is the graph of a matching function from graph Q to graph T.*

In order to prove this result, we first show that in each node of the search tree, partial solution S satifies following invariants:

1. if $(u, v) \in S$, then $lab(u) = lab(v)$, and
2. if $(u, v_1) \in S$ and $(u, v_2) \in S$ then $v_1 = v_2$, and
3. if $(u_1, v) \in S$ and $(u_2, v) \in S$ then $u_1 = u_2$.

Concerning Claim 1, the initial domain of query vertex u is made of target vertices with the same label as u (line 6 of Function CreateInitialNode). Since vertex v comes from this domain when (u, v) is added to S (line 7 of Proc. ExploreTree), we deduce that $lab(u) = lab(v)$.

Claim 2 comes from the fact that when u is selected from the list of unassigned query vertices (line 4 of Proc. ExploreTree), we immediately eliminate u from

Algorithm 3: ExploreTree(N,Q,T)

```
   Input: A node N of the search tree, the digraphs Q and T
   Output: Void, but N.all_solution is the set of all matching functions
 1 if N.unassigned = ∅ then
 2     N.all_solutions ← { N.current_solution } ∪ N.all_solutions
 3 else
 4     u ← SelectUnassignedVertex(N,Q) /* see Section 4.2 */;
 5     for v ∈ N.domains.(u) do
 6         P ← new node();
 7         P.current_solution ← N.current_solution ∪ { (u,v) } ;
 8         P.unassigned ← N.unassigned \ { u };
 9         P.domains ← copy N.domains;
10         P.domains.(u) ← { v };
11         for u' ∈ P.unassigned do
12             P.domains.(u') ← P.domains.(u') \ { v };
13         for u' ∈ X_Q do
14             if u' ∈ succ_1(u) then
15                 P.domains.(u') ← P.domains.(u') ∩ succ_1(v);
16             else
17                 P.domains.(u') ← P.domains.(u') \ succ_1(v);
18         P.domains ← RefineDomainsFurthermore(P,Q,T,u,v) /* see
           Section 4.3 */;
19         if NoEmptyDomains(P.domains) = true then /* see Section 4.4 */
20             P.all_solutions ← N.all_solutions;
21             ExploreTree(P,Q,T);
22             N.all_solutions ← P.all_solutions;
```

the list of unassigned vertices (line 8 of Proc. ExploreTree). So if (u, v_1) is added to S, there is no way to further add (u, v_2) to S.

As for Claim 3, it comes from the fact that when (u_1, v) is added to S (line 7 of Proc. ExploreTree), we eliminate v from the domain of unassigned query vertices (lines 11-12 of Proc. ExploreTree). So there is no way to further assign v to any other unassigned query vertex u_2.

Hence, by Claims 1, 2 and 3, we deduce that when all the query vertices have been assigned, complete solution S is the graph of a injective function $h : X_Q \to X_T$ that preserves the labels of Q, thus h fulfills Conditions 1 and 2 of Def. 1.

In order to establish Conditions 3 and 4, we use both following invariants:

4. if $(u, v) \in S$ and $u' \in succ_1(u)$ and $(u', v') \in S$, then $v' \in succ_1(v)$, and
5. if $(u, v) \in S$ and $u' \notin succ_1(u)$ and $(u', v') \in S$, then $v' \notin succ_1(v)$.

Both these Claims come from lines 13-17 of Procedure ExploreTree. Indeed, when couple (u, v) is added to S (line 7 of Proc. ExploreTree), we filter

the domain of every successor u' of u with successors of v (line 15 of Proc. ExploreTree). This is sound with respect to Claim 2 of Prop. 1. The consequence is that when any successor u' of u is further selected and (u', v') added to S, then v', which is necessarily selected from $dom(u')$, is a successor of v.

As for Claim 5, when couple (u, v) is added to S, we eliminate all the successors of v from the domain of non successors of u (line 17 of Proc. ExploreTree). This is sound with respect to Claim 3 of Prop. 1. The consequence is that when any non successor u' of u is further selected and (u', v') added to S, then v', which is necessarily selected from $dom(u')$, is a non-successor of v.

Now by Claims 4 and 5, we get that when all the query vertices have been assigned, complete solution S is the graph of a function $h : X_Q \to X_T$ that both preserves the presence and the absence of arcs in Q. In other word, h fulfills Conditions 3 and 4 of Def. 1, what completes the proof.

4 The heuristics of PJ

4.1 Initial domains of the variables

We here develop the alternatives that we have considered for function RefineInitialDomains(Q,T) of function CreateInitialNode.

Firstly, a query vertex u can only be matched with a target vertex v for which Condition 1 of Prop. 1 is satisfied. So we set:

$$N.domains.(u) \leftarrow \{v : lab(u) = lab(v) \wedge deg^+(u) \leq deg^+(v) \wedge deg^-(u) \leq deg^-(v)\}.$$

It could be possible to be less restrictive, *e.g.*, by setting $N.domains.(u)$ to X_T. This choice is probably the best for very small query graphs of base *Molecules 4*, up to that fact that one must now check whether a complete solution is a matching function or not (since Condition 2 of Def. 1 is no more guaranteed). Nevertheless, for most graphs of Contest BioGraph'14, some labels rarely appear, and in this case, the domain of corresponding query vertices is very small. Moreover, some labels are very common, thus considering the degree of corresponding vertices helps to reduce their domain.

Secondly, we have exploited Condition 2 of Prop. 1 as follows: suppose that $N.domains.(u) = \{v_1, \ldots v_n\}$ and $u' \in succ_1(u)$; in this case, u' can only be assigned to a target vertex which is a successor of $v_1, \ldots v_n$. In other words, we can reduce the domain of u' with:

$$N.domains.(u') \leftarrow N.domains.(u') \cap \left(\bigcup_{i=1}^{n} succ_1(v_i) \right).$$

We can use this reduction rule until convergence, that is, as long as we do not reach a fixpoint where all the domains are irreducible. In practice, this strategy was interesting for most query graphs of bases *Molecules* and *Contact Maps*, as the convergence occurred after 2 or 3 iterations. However, in the case of *Proteins*, convergence could occur after hundreds of iterations, and in this

case, the computation cost of the reduction was much larger than its benefits. Therefore, we have decided to stop the reduction after $D = 5$ iterations, even if at this stage the convergence had not occurred. This threshold was empirically fixed.

At this point, let *min_init_dom* be the smallest size over all the domains of query vertices. We have empirically observed 3 typical situations. Either the reduction process has converged and *min_init_dom* is very small, that is, there exists at least one query vertex that cannot be assigned to any target vertex, or necessarily to 1 vertex, sometimes 2 at most. In this case, procedure ExploreTree starts the exploration of the tree with such a vertex (see Section 4.2), that allows matching functions to be discovered rapidly, and limits the depth where fail node may appear in the search tree.

Or the convergence of the reduction process has occurred but *min_init_dom* is still quite large, *e.g.*, 60 target vertices. We particularly saw this phenomenon on small graphs of bases *Molecules 8* and *Proteins 8*, where the labels of the vertices are common, the degrees are low, thus our criteria to define the initial domains are not discriminant. In this case, the best choice is to pass the deal to procedure ExploreTree: assigning any query vertex u to some target vertex v dramatically reduces the domains of the successors of u (lines 13-15 of Proc. ExploreTree), and a large value *min_init_dom* is not a problem, *a posteriori*.

Or the reduction process has not converged after 5 iterations and the value of *min_init_dom* is very large, that is, the smallest domain may contain more than 200 target vertices, and up to 800 target vertices. This is often the case with large graphs of bases *Proteins 128* and *Proteins 256*. The point here is that even though we pass the deal to procedure ExploreTree, successive assignments of query vertices often have little impact on the reduction of other domains for such graphs. So we have decided to postpone further domain reductions to function RefineDomainsFurthermore (line 18 of Proc. ExploreTree), which uses the value of *min_init_dom* as a parameter (see Section 4.3).

Lastly, we could have used Condition 3 of Prop. 1 as follows: suppose that $N.domains.(u) = \{v_1, \dots v_n\}$ and $u' \notin succ_1(u)$; in this case, u' cannot be assigned to any target vertex which would be a common successor to $v_1, \dots v_n$. In other words, we could reduce the domain of u' with:

$$N.domains.(u') \leftarrow N.domains.(u') \setminus \left(\bigcap_{i=1}^{n} succ_1(v_i) \right).$$

We did not develop this idea in the framework of Contest BioGraph'14 because all the graphs are sparse, so the degree of the vertices is very low. Therefore, in the target graphs, it is almost always the case that the intersection of the sets of successors of $v_1, \dots v_n$ is empty, whereas the computation cost of such a rule is not null. This rule might be interesting if we had to deal with dense graphs.

4.2 Choice of the next vextex to assign

This choice is made by function SelectUnassignedVertex(N,Q) at line 4 of procedure ExploreTree: we consider all unassigned query vertices, and compare them with respect to the size of their current domain; then we select the vertex whose domain is the smallest.

In the case where several vertices have the same domain size, we select the vertex which has the maximum number of successors. Indeed, once a query vertex is matched, PJ reduces the domains of its successors (see Section 4.3), so this heuristics favours a large number of domain reductions afterwards.

And in case of equality, we also consider the number of 2-successors of the vertices, and also try to maximise it for the same reason. Note that it is sometimes interesting to consider k-successors for $k \geq 3$, in particular when the algorithm is used on large graphs of base *Proteins*. But the computation cost of this operation is prohibitive for other bases and we did not keep this idea.

4.3 Domain reduction during the exploration of the search tree

We have already seen that whenever a query vertex u is assigned to any target vertex v, procedure ExploreTree modifies and adapts the domain of other query vertices. Indeed, lines 8-17 are necessary to get a matching function from graph Q to graph T when the set of unassigned vertex is empty, as shown in Section 3.3.

Further refinements are performed by function RefineDomainsFurthermore, which makes use of Condition 4 of Prop. 1, that is, if vertex u is assigned to vertex v, then every k-successor of v, for $k \geq 2$, must be assigned to some k-successor of v. We use this rule for 2-successors, 3-successors, ... up to a maximum threshold max_dist that we discuss below. Notice that the distinction we made about successors with respect to neighbours is important here, since several reductions of the same domain can yield an empty domain, which allows the algorithm to backtrack precociously.

Concerning parameter max_dist, there is about $\mathcal{O}\left(av\_deg^{max\_dist}\right)$ domains to reduce each time the algorithm assigns a query vertex to a target vertex, where av_deg denotes the average degree of the query graph. So a large value of max_dist can penalize the execution time of the algorithm, even though this rule dramatically reduces many domains.

After several experiments, we have finally decided to relate this parameter with the minimal size min_init_dom of initial domains (defined in Section 4.1) as follows:

$$max\_dist = 2 + \left\lfloor \frac{min\_init\_dom}{100} \right\rfloor .$$

Indeed, once initial domains reduction is performed, most graphs have domains of small size. In this case, it is useless to refine the domains of successors at a distance greater than 2 in procedure ExploreTree. On the other hand, when the refinement process of initial domains evolves so slowly that we have to stop

it before convergence, the query graphs can have very large initial domains, and for these graphs, it is often interesting to refine the domains of far k-successors.

For instance, with $min\_init\_dom = 364$, which is quite common with base *Proteins 256*, the algorithm will refine all the domains of 1-, 2-, 3-, 4- and 5-successors of every query vertex u that is assigned to some target vertex.

4.4 Testing the emptyness of the domains

Each time a domain is reduced, it is interesting to check if it became empty, what function NoEmptyDomains performs (line 19 of Proc. ExploreTree). In this case, the corresponding query vertex cannot be matched with any target vertex, and the algorithm has to backtrack. Nevertheless, this verification is not costless, because many vertices can have their domain reduced after each assignment.

More precisely, our experiments have showed that for small query graphs, with less than 8 vertices, procedure ExploreTree was more efficient if it never checked the emptyness of any reduced domain. As for large graphs, the price of a systematic testing was also challenging.

So we have decided to test the emptyness of the domains every 9 performed assignments only. This empirical value appeared to be a good tradeoff between the cost of the verification, and the capacity to early detect dead branches of the search tree. An improved mechanism would aim at finding the best parent node of the search tree where to backtrack in this case.

4.5 Immediate fail criteria

We here explain function ImmediateFailCriteria(Q,T). Motivated by the enumeration of matching functions in the framework of induced subgraphs (as opposed to partial graphs) over connected graphs, we have considered following criteria which allows PJ to declare no solution immediately:

- if Q has strictly more vertices than T;
- if Q has strictly more arcs than T;
- if Q and T have the same number of vertices, but not the same number of arcs: indeed, Q and T must be isomorphic in this case;
- if Q and T have the same number of arcs, but not the same number of vertices: indeed, Q and T are connected, so this situation also requires Q and T to be isomorphic;
- if Q is a directed graph and T a non-directed graph: Q could be a partial subgraph of T, but not an induced subgraph;
- if $Lab_Q \not\subseteq Lab_T$, where Lab_Q and Lab_T denote the multisets of labels in Q and T respectively. *E.g.*, if Q contains three Carbon (among others), and T only two Carbon, then the matching is impossible.

References

[BES80] L. Babai, P. Erdös, and S. M. Selkow. Random graph isomorphism. *SIAM Journal of Computing*, 9(3):628–635, 1980.

[CFSV04a] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

[CFSV04b] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[CFV13] V. Carletti, P. Foggia, and M. Vento. Performance comparison of five exact graph matching algorithms on biological databases. In *Proc. International Conference on Image Analysis and Processing (ICIAP'13)*, pages 409–417. LNCS 8158, 2013.

[Cod13] P. Codenotti. Distinguishing vertices of inhomogeneous random graphs. Technical Report IMA Preprint Series 2419, Institute for Mathematics and its Applications, University of Minnesota, Minneapolis, Minnesota, July 2013.

[FPV14] P. Foggia, G. Percannella, and M. Vento. Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(1), 2014.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Sol10] C. Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13):850–864, 2010.