A Fast Discovery Algorithm for Large Common Connected Induced Subgraphs

Alessio Conte, Roberto Grossi, Andrea Marino, Lorenzo Tattini, Luca Versari

A Fast Discovery Algorithm for Large Common Connected Induced Subgraphs

Alessio Conte[a], Roberto Grossi[b], Andrea Marino[b], Lorenzo Tattini[c], Luca Versari[b]

[a] *National Institute of Informatics, Tokyo, Japan,* conte@nii.ac.jp
[b] *Dipartimento di Informatica, Università di Pisa, Pisa, Italy,* {grossi,marino,luca.versari}@di.unipi.it
[c] *IRCAN, CNRS UMR 7284, Nice, France,* lorenzo.tattini@unice.fr

Abstract

We present a fast algorithm for the classical problem of finding common subgraphs, which are useful for detecting structural relationships between biological macromolecules. Although the cost is potentially high for this hard problem, we improve performance by several orders of magnitude compared to known algorithms. We validate our findings with experiments on proteins with thousands of atoms.

Keywords: Enumeration, Maximal Common Subgraphs, Isomorphisms, Output Sensitive

1. Introduction

For any two given input graphs G and H, a subgraph S of G is *in common* with H if S is isomorphic to a subgraph of H: it is maximal if there is no other common subgraph that strictly contains it, and maximum if it is the largest. The *maximum* common subgraph problem asks for the maximum ones, or simply for their size: this problem is classical and useful for modeling structural similarity. The *maximal* common subgraph (MCS) problem further requires discovering all the MCS's of G and H.

The MCS problem can be constrained to *connected* and *induced* subgraphs (MCCIS) [1, 2, 3], where the latter means that all the edges of G between nodes in the MCS are mapped to edges of H, and vice versa: considering induced subgraphs reduces the search space [1], and requiring connected subgraphs has been employed to further alleviate the explosion of the number of solutions [2, 3].

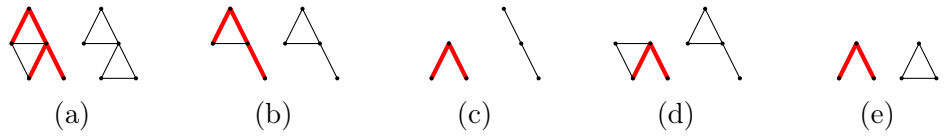

Figure 1: (a) Two graphs G and H, where T is shown in red; (b) a T-MCCIS that is also MCCIS; (c) a T-MCCIS; (d) not a T-MCCIS since it is a MCCIS but not spanned by T; (e) not a T-MCCIS since it is spanned by T but is not a common induced subgraph.

Problem of interest. In this paper, we consider the following modified version of the MCCIS problem, which we call T-MCCIS problem.

Problem 1 (T-MCCIS Problem)**.** *Given two graphs G and H and a spanning tree T of G, list all the maximal common connected induced subgraphs between G and H for which the subgraph in G is connected using edges of T.*

We call these subgraphs T-MCCIS's, and some examples are shown in Fig. 1. As we will see, their interest arises for their biological application. While a T-MCCIS is not necessarily a MCCIS, it is still a common connected subgraph. The adoption of the T-MCCIS model allows us to find in practice common structures which are of larger size and more numerous with respect to the ones found by the state-of-the-art methods on for MCCIS (see Section 3.3).

We remark that spanning trees have been previously employed to prune the search for frequent subgraphs [4], although such techniques do not extend to this problem.

Furthermore, it should be remarked that all state of the art approaches, as well as ours, by MCCIS (or T-MCCIS) actually mean *isomorphisms* corresponding to a MCCIS (or T-MCCIS), i.e. the subgraphs G and H *and* a compatible mapping of the nodes of G into those of H. There are currently no known techniques for efficiently finding actual common subgraphs without considering isomorphisms. Listing isomorphisms may result in finding the same pair G and H more than once with different mappings; however, we will also address the issue of removing this redundancy a posteriori (see Section 3).

When nodes of G and H are labeled, the notion of isomorphism naturally extends by requiring, for instance, nodes of the common subgraph to have matching labels. We can also consider a generalized compatibility function between the nodes of the two graphs that determines whether a given node i of G can be mapped in a node j of H. An analogous definition can be given for edges.

Contributions. In this paper we solve the T-MCCIS problem by providing an output sensitive algorithm for (isomorphisms corresponding to) T-

MCCIS's, i.e. an algorithm which takes polynomial time (and space) per solution found. In particular, given the two graphs G and H, and a spanning tree T of G, let Δ_G and Δ_H be their maximum degree, respectively. For each listed T-MCCIS, we pay a parametrized cost of $O(q^3\Delta_{black}^2(\Delta_G + \Delta_H))$ time, where q is the number of nodes of the listed T-MCCIS, and Δ_{black} is a parameter bounded by $O(\Delta_G\Delta_H)$. Note that a strength of this bound is that it is *independent* of the sizes of G and H. This is the first output sensitive algorithm for T-MCCIS.

To achieve our goal, we explore a variation of the product graph P [5] obtained from G and H, so that T-MCCIS's are found as special cliques in P. Recent advances on clique enumeration [6] might help to find these special cliques but, due to the new constraints, we have to solve several non-trivial issues to avoid the explosion of combinations of partial solution to get the new ones. Moreover, our approach does *not* materialize P, but navigates it implicitly to improve memory usage and running time. We refer the reader to Section 2.

Another contribution is that we use our solutions to design a new method to find LACCIS's, which are *large* common connected induced subgraphs (not necessarily maximal nor maximum). They are relevant when comparing macromolecules in computational biology. For a set of spanning trees $T_1, \ldots, T_k$, we consider the set of LACCIS's such that each LACCIS L contains a T-MCCIS S for some $T \in \{T_1, \ldots, T_k\}$. In general, L satisfies $S \subseteq L \subseteq M$ for a MCCIS M, where $\subseteq$ denotes the containment relation among induced subgraphs. Hence the larger L, the closer it is to a MCCIS.

Our algorithm, called FLASH (Fast LAccis Searching Heuristic), takes two connected labeled graphs G and H as input, along with some random spanning trees $T_1, \ldots, T_k$ of G. It applies the above approach, and returns a set of LACCIS's, where each LACCIS is represented as a pair of subsets of nodes, one from G and the other from H.

FLASH uses our solution for the T-MCCIS problem for each tree, accumulating the found T-MCCIS's for $T = T_1, \ldots, T_k$. Then, it greatly reduces their number by a filtering criterion to make sense of the massive output: for a user-defined percentage σ (e.g. 70%), it selects a "covering" set of small size, such that each of the discarded T-MCCIS's has more than σ overlap with a retained one (priority is given to large ones). Interestingly this filter shows that FLASH quickly finds solutions spanning different parts of G and H, whereas other approaches such as [3] tend to spend lot of time on the same nodes: small local additions and deletions of nodes produce a plethora of different subgraphs that significantly overlap.

We show that using T-MCCIS's instead of MCCIS's is more efficient for

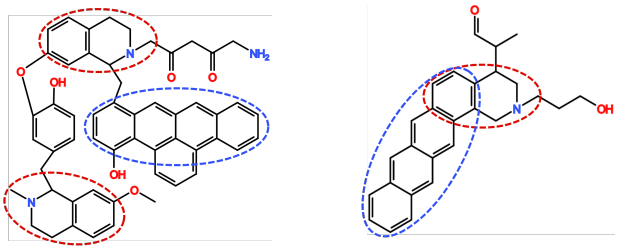

Figure 2: Common structures in Liensinine derivatives

finding LACCIS's since the running time of FLASH for a given spanning tree T is provably proportional to the number of reported T-MCCIS's. This is in contrast with the known algorithms that use maximal common subgraphs, which have the drawback of running into a computational blackhole, going through an exponential number of substructures even if there are few T-MCCIS's: for a T-MCCIS of k nodes, these algorithms have to potentially find and discard 2^k included subgraphs of it. When dealing with graphs of non-trivial size (e.g., thousands of nodes) we argue that the state-of-the-art approaches that use maximal common subgraphs do not terminate within a conceivable time, thus making a practical comparison hard to perform. In our experiments, the size k of common subgraphs can easily be in the order of the hundreds and FLASH performs well in practice even though its theoretical worst-case complexity is exponential. We refer the reader to Section 3.

The experimental results in this paper have been presented in [7, 8].

Maximum vs maximal common subgraphs. Graph-based methods provide a natural complement to sequence-based methods in bioinformatics and protein modeling. Graph algorithms can identify compound similarity between small molecules, and structural relationships between biological macromolecules that are not spotted by sequence analysis [9]. These algorithms find motivation in the increasing amount of structured data arising from X-ray crystallography and nuclear magnetic resonance. Many examples of graphs fall under this scenario, such as chemical structure diagrams [10, 11], 3D patterns for proteins [11, 3], amino acid side-chains [12], and compound similarity for the prediction of gene transcript levels [13], to name a few. It is believed that finding similar structures leads to highlighting similar biochemical properties and functionalities [14, 15]. For these reasons, the bioinformatics community has repeatedly expressed its interest in common subgraphs detection from a computational point of view [16, 17, 18, 19, 3, 13, 20, 9, 21, 22, 23, 24, 25].

Some people might confuse the problem of finding the *maximum* common subgraph with that of finding all the *maximal* common subgraphs: they are indeed different problems, solved with different techniques and results. It is in practice much faster to find the maximum common subgraph than all MCS's (e.g. see [18]). However, a maximum common subgraph is not always meaningful as a structural motif, as it does not necessarily contain all the relevant or large common structures. Consider the two molecules represented by the graphs in Figure 2 as an example. If we examine these molecules, a natural question is determining which parts of the two are common. As it can be seen, the two molecules share a tetracenic moiety (blue) and an *N*-methylbenzopiperidinic moiety (red). It is worth observing that the maximum common structure in terms of size is represented by the blue group, which is however a fairly common structure in organic molecules; the red one which is maximal (but not maximum), however, is more likely to be interesting as it is more peculiar. In general, there may be arbitrarily large common substructures that give little information because of their frequent appearance in special type of macromolecules or polymers. In this context, modeling the problem from a mathematical perspective can be useful to efficiently retrieve these common structures.

Furthermore, when spotting structural motifs, it is not always possible to fix a priori the scoring system, and the maximum common subgraphs are not necessarily the ones getting the best score: a postprocessing can apply several scoring systems with a fast filtering and ranking of the MCCIS. This is more efficient than repeating a branch-and-bound search for each score.

Related works. Both maximum and maximal common subgraphs problems have been studied for decades [22, 26, 11]. In the case of the maximum common subgraph, the corresponding decision version is NP-complete as it solves the subgraph isomorphism problem. The problem remains NP-hard, even on restricted graph classes such as outerplanar graphs, and becomes polynomial only if the degree is bounded or for trees [27]. The problem is difficult to approximate (MAX-SNP hard) even within a polynomial factor [19], and is $W[1]$-hard when parameterized by the treewidth of the input graphs [16]. Due to these difficulties, enumerating all the maximum common subgraphs cannot be done in an output sensitive way, e.g. with polynomial delay, unless P=NP. Listing all maximal common connected induced subgraphs is a different problem, since it can produce exponentially more solutions than finding maximum common subgraphs. Moreover, finding just the maximum common subgraphs, as opposed to listing all maximal ones, allows for very effective cuts to the search space (e.g. branch-and-bound) which makes the

computation much faster in practice, allowing researchers to process larger graphs with the available resources.

Due to the strong connection between graph isomorphism and common substructures, the bioinformatics community has repeatedly expressed its interest in both these problems from a computational point of view, looking at them as two possible ways of finding LACCIS's. However, we observe that the great majority of the works deal with maximum common subgraphs, rather than maximal ones.

For both the problems, the previous work can be roughly classified into three categories: clique-based methods [11, 3, 28], non-clique-based backtracking methods [29, 30, 31], and other techniques based on special classes of graphs [27, 32].

Clique-based methods are widely employed and rely on the product graph P, transforming the common subgraphs of G and H into maximal cliques in P. This reduction dates back to the 70s [5] and has been shown to be effective on biological networks [18, 11, 3, 28]. For finding the maximal cliques, the algorithms by Bron and Kerbosch [33] or Carraghan and Pardalos [34] have been employed. Cao et al. [26] observe that materializing P can be memory-wise expensive, and we show how to avoid this in our approach.

As previously observed, finding MCCIS's of G and H corresponds to finding "special" cliques [2, 3]. To this aim, previous works on explicit product graphs for listing all the MCCIS's, such as the ones by Koch [2, 3], employ a modified version of the Bron-Kerbosch algorithm which does not perform *pivoting*, a pruning technique. The resulting algorithm is not output sensitive, since it iterates on every possible subset of each common subgraph, and its complexity and cost per solution are not clearly bounded. Many works focused on using and improving Koch's algorithm to get only the *maximum* common connected induced subgraphs: [35] compares favorably Koch algorithms with other existing algorithms to compute this maximum. It is often applied with this purpose in the case of protein structure comparison [36]. [37] lists all the common connected induced not necessarily maximal subgraphs to extract the maximum without using a product graph. [38] relaxed the constraint of induced and compared to the maximum found by using Koch. When instead considering all the maximal common connected induced subgraphs, Koch's algorithm has been the state of the art until recently and it is still greatly used in practice even though it is not output sensitive (see `www-ikn.ist.hokudai.ac.jp/~wasa/enumeration_complexity.html` or [39, 40]). Indeed, the present paper introduces the first output sensitive algorithm for T-MCCIS's, and a subsequent paper [41] describes the first output sensitive algorithm for MCCIS's.

Backtracking algorithms mostly build up on Ullman's strategy [42] for subgraph isomorphism (e.g. [30]). They often use branch-and-bound heuristics based on the specific requirements of the application at hand. The comparison in [18] shows how direct implicit methods for the maximum common subgraph, such as the one in [30], can outperform methods that exploit the product graph if the input graphs are small or contain many different labels. However, they do not apply efficiently to listing all the MCCIS's. Other techniques, such as dynamic programming can be employed for special classes of graphs [27].

Other variations of the MCCIS problem have been considered, when G, H, and their isomorphisms are restricted to trees [43].

2. Output Sensitive T-MCCIS's Enumeration

This section is devoted to solve the T-MCCIS problem. In the first part, namely Section 2.1, we show how to turn the problem of finding T-MCCIS's into the problem of finding suitable cliques in a new graph. In Section 2.2, we perform an output sensitive listing of these cliques.

2.1. Problem transformation: Implicit product graph

We employ a variant of the transformation adopted by Koch [2] and borrowed from Levi [5], where we modify the color rule to take into account the edges of the spanning tree T. Define a *colored product graph* $P = GH$. Letting $P = (V_P, E_P)$ be the resulting undirected graph, the nodes in V_P correspond to ordered pairs of compatible nodes from G and H, the first from G and the second from H, and the edges in E_P are as follows. Given two nodes in V_P corresponding to (x, i) and (y, j), where $x, y \in G$, $i, j \in H$ and $x \neq y, i \neq j$, there is an edge in E_P between (x, i) and (y, j) iff: (i) $\{x, y\} \in T$ (tree edge) and $\{i, j\} \in H$ (in this case, the edge is *black*), (ii) $\{x, y\} \in G \setminus T$ (non-tree edge) and $\{i, j\} \in H$ (the edge is *white*), (iii) both $\{x, y\} \notin G$ and $\{i, j\} \notin H$ (the edge is *white*). There are no edges between nodes (x, i) and (y, j) in V_P, where $x, y \in G$, $i, j \in H$, if $x = y$ or $i = j$.

The difference with the transformation by Koch [2] is in condition (ii): we obtain exactly Koch's transformation if those edges are marked as black instead of white. Nonetheless, we will show that this difference is quite crucial to obtain our output sensitive algorithm. An example of the above transformation is shown in Fig. 3. As in [2], there is a one-to-one correspondence between maximal cliques in P and maximal isomorphisms between

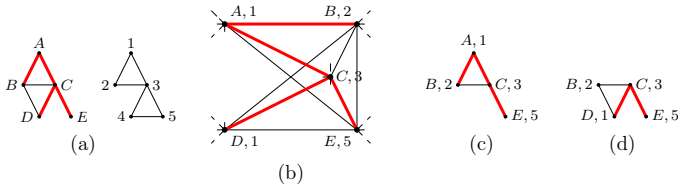

Figure 3: (a) Two graphs G and H, where T is shown in red; (b) a portion of the induced subgraph of P involving $(A, 1)$,$(B, 2)$,$(C, 3)$,$(D, 1)$, and $(E, 5)$ (note that the edge $(A, 1), (D, 1)$ is not present); (c) a T-MCCIS (d) not a T-MCCIS since it is common but not spanned by T.

subgraphs of G and H. The main difference is that in our case a maximal clique connected by black edges corresponds to a maximal subgraph connected by edges of T, instead of generic edges of G.

We call this kind of black-connected maximal clique a BC-*clique*, and reduce the problem of finding the T-MCCIS's to that of finding the isomorphisms/BC-cliques in the implicit P. We observe that the same T-MCCIS can give raise to several maximal isomorphisms/BC-cliques (matching the two sets of nodes in that T-MCCIS) that should be successively distilled to list it exactly once.

Building and navigating P is costly: P is a dense, massive graph with large maximum degree even when G and H are relatively small, sparse and with bounded degree. We avoid storing P explicitly, and only store G and H: we check compatibility between assignments in constant time and iterate on neighbors in constant time per element by applying the rules used for the generating P "on the fly". Even thought the time complexity bound is the same, navigating P implicitly saves both memory and time, since G and H are much smaller and faster to access than P. A practical study on the difference between computing P and using it implicitly when computing MCCIS's has been done in [44].

In Section 2.2 we will show how to find the BC-cliques in P.

2.2. Finding BC-*cliques*

We describe in this section our algorithm for finding BC-cliques in the implicit product graph P. The algorithm works incrementally by finding all BC-cliques of increasingly larger subgraphs of P, to finally find all BC-cliques of P. We first give a simplified version of the algorithm in Section 2.2.1, then the final one in Section 2.2.2. This incremental approach resembles the one in [45] for generating maximal independent sets.

2.2.1. First algorithm for BC*-cliques*

Our algorithm works by *extending* cliques: given a BC-clique K, a node v *fully extends* K if it is connected to all the nodes in K and has at least one black edge to a node in K; v *partially extends* K if it has at least one black edge to a node in K, but is not connected to all the nodes in K; v *extends* K if it fully or partially extends K. A crucial ingredient is finding a *good ordering* of the nodes in V_P, which is defined as follows.

Definition 1. *Given a graph $P = (V_P, E_P)$ whose edges are either black or white, we say that an ordering of the nodes in V_P is a good ordering if: for any triple of nodes $x, y, v \in V_P$, with $x \prec y \prec v$ and $\{x, v\}, \{y, v\} \in E_P$, we have that if both $\{x, v\}$ and $\{y, v\}$ are black, then $\{x, y\} \notin E_P$.*

In other words, if v has two black edges towards neighbors that come earlier in the good ordering of P, those neighbors are not connected to each other. We show that it is always possible to find a good order for the product graph P defined in Section 2.1.

Lemma 1. *Given two graphs G and H, and a spanning tree T of G, it is always possible to find a good order for the corresponding product graph P.*

Proof. We can number the nodes of G using a preorder traversal of T, so that given any node u of G, there is at most *one* edge of T between u and a neighbor u' with $u' < u$. Considering H, we just number its nodes in an arbitrary way. As for P, we consider the lexicographical order $\prec$ on the pairs (x, i), that corresponds to the nodes in V_P, where x follows the order for G mentioned above and i the order for H. The numbering of the nodes in V_P obtained by numbering them consecutively in increasing lexicographical order $\prec$ of the corresponding pairs is a good ordering: for each (x, i), the nodes of P smaller than (x, i) connected to it by black edges are of the form $(x', \cdot)$, where x' is the unique smaller neighbor of x in T; such nodes are clearly not connected to each other by definition of P. □

In the remainder of the paper, let $P_{\prec v}$ denote the subgraph of P induced by the nodes $v' \prec v$, and $P_{\prec v} \cup \{v\}$ the one induced by $v' \preceq v$. The following lemma holds.

Lemma 2. *If C is a* BC*-clique in $P_{\prec v} \cup \{v\}$, and P is ordered in a good ordering, then $C \setminus \{v\}$ is connected with black edges in $P_{\prec v}$.*

Proof. If v has only one black edge towards the rest of C, then $C \setminus \{v\}$ would still be connected with black edges as no path made of black edges between

nodes of $C \setminus \{v\}$ could involve v. Otherwise, assume v has at least two black edges towards nodes of $C \setminus \{v\}$: by the good ordering, these nodes cannot be connected to each other in P as they are smaller black neighbors of v, meaning that C would not be a clique, a contradiction. □

Observe that a BC-clique in $P_{\prec v} \cup \{v\}$ is either a BC-clique of $P_{\prec v}$ or contains v. Moreover, a BC-clique C containing v is such that $C \setminus \{v\}$ is connected by black edges, because of Lemma 2, and so it is contained in a BC-clique of $P_{\prec v}$. This property ensures that every BC-clique can be found incrementally: when adding a new node v to the set of BC-cliques found up to that point for the nodes of $P_{\prec v}$, two or more of the latter BC-cliques cannot be united because of the black edges incident to v, since the removal of v cannot disconnect the BC-clique. Hence we can consider just *one* of them to be extended by v, rather than any combination of them.

We are now ready to describe our algorithm which assumes that P is explicitly stored. We call this algorithm INC-GENERATOR (incremental-generator).

Let R_v be the set of BC-cliques in $P_{\prec v} \cup \{v\}$. Considering the nodes $v_1, \ldots, v_p \in P$ in the good order, for each v_i we build the set R_{v_i} from $R_{v_{i-1}}$ (R_{v_1} contains only $\{v_1\}$). Clearly, R_{v_p} is the set of all BC-cliques of P. In order to obtain R_{v_i},

1. we iterate over the cliques K in $R_{v_{i-1}}$ and we do the following.
   (a) If v_i fully extends K, add $K \cup \{v_i\}$ to R_{v_i}, otherwise add K to R_{v_i}.
   (b) If v_i partially extends K, define C as the component connected by black edges and containing v_i in the graph induced by $K \cap N(v_i) \cup \{v_i\}$[1]. If C cannot be fully extended with a node smaller than v_i in $P_{\prec v_i} \cup \{v_i\}$, add the clique C to R_{v_i}.
2. If no clique in $R_{v_{i-1}}$ is extended by v_i, add $\{v_i\}$ to R_{v_i}.

Note that the same clique might be generated more than once, thus it is important to keep R_{v_i} as a set, not retaining duplicates. INC-GENERATOR tries to extend each K in $R_{v_{i-1}}$ with v_i. If v_i does not fully extend K, K is maximal also in R_{v_i}. Moreover, each time an extension (full or partial) of K is possible, this is done, except in the step 1.b when C may be extended with a node smaller than v_i: indeed C is not maximal, and it is not added to R_{v_i}. Note that the completeness of the approach is not affected by discarding C:

[1]Note that C is a BC-clique and contains some, but *not* all, nodes of K.

by Lemma 3, if there exists a node v_j smaller than v_i extending C, there exists a clique C' in $R_{v_{i-1}}$ such that $(C \setminus \{v_i\}) \cup \{v_j\} \subseteq C'$.

Lemma 3. *Any* BC*-clique in R_{v_i} is either in $R_{v_{i-1}}$ or can be obtained by extending a clique in $R_{v_{i-1}}$ with v_i.*

Proof. Assume there exist a BC-clique $C \in R_{v_i}$ that is not obtained by the algorithm when processing v_i. All cliques in R_{v_i} which do not contain v_i are in $R_{v_{i-1}}$ by definition, thus C contains v_i. By Lemma 2, $C \setminus \{v_i\}$ is a clique connected with black edges. If $C \setminus \{v_i\}$ is maximal in $P_{\prec v}$, then it is fully extended by v_i and C is found in step 1.a. Otherwise, $C \setminus \{v_i\}$ is contained in a clique C' maximal in $P_{\prec v}$. We thus have $C \subseteq (C' \cup \{v_i\})$. If $C \neq C' \cup \{v_i\}$ then C is not maximal in R_{v_i}, a contradiction. □

Lemma 4. INC-GENERATOR *outputs just* BC*-cliques.*

Proof. We prove that at any step, R_{v_i} contains only BC-cliques, which are maximal in $P_{\prec v_i} \cup \{v_i\}$. By induction, all the cliques that contain v_i are discarded if not maximal (step 1.b) and the other ones, since they cannot be fully extended by v_i and were maximal in $R_{v_{i-1}}$, are still maximal. Thus in the final step R_{v_p} will contain only BC-cliques maximal in $P_{\prec v_p} \cup \{v_p\} = P$. □

By Lemma 3 and since INC-GENERATOR outputs just BC-cliques by Lemma 4, we obtain the following.

Corollary 1. *Algorithm* INC-GENERATOR *outputs all and only* BC*-cliques.*

For the sake of completeness we remark that Algorithm INC-GENERATOR has average cost per solution which is polynomial in the input size, i.e. in the size of G and H, as shown by the following lemma.

Lemma 5. *Algorithm* INC-GENERATOR *has polynomial cost per solution.*

Proof. Note that, since $|R_{v_{i-1}}| \leq |R_{v_i}|$, we have that $\sum |R_{v_i}| \leq p|R_{v_p}|$; in the execution of our algorithm, the number of times in which any clique is processed is bounded by p times the number of BC-cliques of P. Since processing each clique takes polynomial time, our total running time is given the number of BC-cliques in the graph multiplied by a polynomial. □

2.2.2. Refined version

In this section we present a refined version of the algorithm shown in Section 2.2.1, showing that our new version is equivalent to INC-GENERATOR. Our new algorithm is called GENERATOR and is a variant of the reverse search [46] that takes into account the distinction between black and white edges and recursively examines all the BC-cliques as explained before. In this version, we avoid using the sets R_{v_i} as they can potentially include an exponential number of BC-cliques. Moreover, we deal with the fact that the product graph is not explicitly materialized, meaning that when retrieving desired nodes of P, which are pairs of nodes in G and H, this task should be addressed just by looking at G and H. Note that deciding whether a node in P is less than another, can be done just by looking at the order induced on G and H as shown in Lemma 1.

The pseudocode is shown in Algorithm 1. The roots of the recursion trees are given by all the nodes that have no black backwards edges, i.e. nodes having no black edges going to previous nodes in the good order in Definition 1. Recall that, in algorithm INC-GENERATOR, each BC-clique $K' \in R_{v_i}$, for any i, is either a single node with no backward black edges or is generated from another BC-clique K in R_{v_j}, with $j < i$. Given a BC-clique K in R_{v_j}, we analyze the properties of all K' generated from K in algorithm INC-GENERATOR. In particular, we identify for which $i > j$, K can generate a clique in R_{v_i}. The Lemma below trivially follows from the definition of BC-clique.

Lemma 6. *In algorithm* INC-GENERATOR*, if a* BC*-clique K generates a new* BC*-clique when processing v_i, then there is a black edge from v_i to a node in K.*

By Lemma 6, it is easy to find all nodes v_i that can be used to extend K using the following.

Property 1. *A node v_i can extend K if it is connected to K with black edges and it is greater than the largest node in K in the good ordering.*

The set of these nodes v_i corresponds to the set X in Algorithm 1. Property 1 helps to find the K' which are generated from K in algorithm INC-GENERATOR: they are found among the ones obtained recursively extending $C \cup \{v_i\}$, where C is the set of nodes in $K \cap N(v_i)$ that v_i can reach using black edges. This allows us to generate the BC-cliques without using the sets R_{v_i} explicitly.

Given a BC-clique C, and letting v be the largest node of C in the good ordering, we define the *parent* of C as the BC-clique in $P_{\prec v}$ obtained by

Algorithm 1: GENERATOR: Finding the BC-cliques in the product graph P

Input: Two graphs G and H and a spanning tree T of G
Output: The BC-cliques in the implict product graph $P = GH$ for T

Let $v_1, \ldots, v_p$ be the nodes of V_P in lexicographic order (good ordering);
foreach *v_i having no black edges going to v_j with $j < i$* **do**
    `Generate`$(\{v_i\})$;

Procedure `Generate`(K)
    if *K is maximal* **then** output K
    Let X be the nodes larger than $\max(K)$ connected to K by a black edge
    for $v_i \in X$ **do**
        Let C be the nodes in $(K \cap N(v_i)) \cup \{v_i\}$ that v_i can reach using black edges in $(K \cap N(v_i)) \cup \{v_i\}$
        if *C is a child of K* **then** `Generate`(C)

maximalizing $C \setminus \{v\}$ in $P_{\prec v}$, that is, recursively adding the smallest node in $P_{\prec v}$ that can fully extend the current (non-maximal) BC-clique until it is maximal in $P_{\prec v}$ (no more nodes can be added).

Lemma 7. *The parent K of a* BC*-clique $C \in R_{v_i}$ with $v_i = max(C)$, is unique and is a* BC*-clique in $R_{v_{i-1}}$.*

Proof. By Lemma 2, $C \setminus \{v_i\}$ is a clique connected by black edges. By recursively adding nodes in $P_{\prec v_i}$ that fully extend it, we obtain a clique K that is connected by black edges and maximal in $P_{\prec v_i}$, that is, a BC-clique in $P_{\prec v_i}$. As $P_{\prec v_i} = P_{\prec v_{i-1}} \cup v_{i-1}$, K is a BC-clique in $R_{v_{i-1}}$. Furthermore, recall that $R_{v_{i-1}}$ is a set; as there are no duplicates the parent K of C is unique. □

Considering that the nodes added to $C \setminus \{v_i\}$ to compute K are not neighbors of v_i reachable with black edges (otherwise C would have not been maximal), we have that $(K \cap N(v_i)) \cup \{v_i\} \supseteq C$, thus by Lemma 7 it follows that:

Corollary 2. *A clique C, with $v = max(C)$, can be obtained by extending its parent with v.*

Definition 2. *Given two cliques K and C, and $v = max(C)$, we say that C is a* child *of K iff: C is maximal in $P_{\prec v}$, and the parent of C is K.*

Note that in Algorithm 1, when trying to generate C from K, we accept C only if its parent is K; otherwise, C is discarded. This avoids generating

duplicates: as every BC-clique has exactly one parent, and can only be generated in one way from any BC-clique (when adding v_i), clearly it is impossible to generate any clique more than once, obtaining:

Lemma 8. *Any* BC*-clique produced by* GENERATOR *is generated exactly once.*

Finally, we characterize for which v_i the BC-clique $\{v_i\}$ in the algorithm in Section 2.2.1 is added to R_{v_i}. This corresponds to the v_i having no black edges going to v_j, with $j < i$. For each of these sets, in Algorithm 1 we start our recursive procedure. From this observation, applying Property 1, and Lemma 7, we can conclude the following.

Lemma 9. *Algorithm* GENERATOR *is equivalent to algorithm* INC-GENERATOR.

By Lemma 1 and Lemma 9, we can conclude that Algorithm 1 is correct. Let us now discuss its time complexity, explaining also how not to store explicitly P.

In the following, we refer to q as the maximum number of nodes in a BC-clique, Δ_{black} as the maximum number of black neighbors of a node in P, and Δ_G and Δ_H as the maximum degrees respectively in G and H. Note that checking the existence of an edge in P takes constant time, assuming that it takes constant time in G, H and T, e.g. using Cuckoo Hashing. Indeed, given two nodes (x, i) and (y, j) it is sufficient to apply the rules described in Section 2.1, i.e., checking the existence of an edge in respectively G, H and T. The following two results hold.

Lemma 10. *Maximalizing a clique K in $P_{\prec v}$ can be done in $O(q\Delta_{black}(\Delta_G + \Delta_H))$ time.*

Proof. Maximalization of K in $P_{\prec v}$ can be done by computing the set B of all the black neighbors of nodes in $K \setminus \{v_i\}$; then consider each b in B in increasing lexicographical order, and add b to the clique if it is adjacent to all nodes of K. Each time a node is added, its black neighbors are added to B. The total cost for building B is $O(q\Delta_{black})$ and its size $|B| \leq q\Delta_{black}$. Note that, if we assume that our graph is given by the product of two graphs G and H, checking if a node $v = (a, b)$ is adjacent to all nodes of K can be done in $O(\Delta_G + \Delta_H)$ time. Indeed, we only need to check if the sets of neighbours of a in the G-side of C is mapped exactly to the sets of neighbours of b in the H-side of C. Since these two sets can be computed in $O(\Delta_G)$ and $O(\Delta_H)$ respectively, the total time take is $O(\Delta_G + \Delta_H)$. This gives us a total cost of $O(|B|(\Delta_G + \Delta_H)) = O(q\Delta_{black}(\Delta_G + \Delta_H))$ □

Lemma 11. *In Algorithm 1, for each $v_i \in X$, C can be computed in $O(q\Delta_{black})$ time.*

Proof. Let $C' = (K \cap N(v_i)) \cup \{v_i\}$; computing C' is done in $O(|K|)$ by iterating over K and checking the neighborhood with v_i in P (which takes constant time). To compute C, it is sufficient to retain in C' only nodes reachable from v_i with black edges in C'. This can be done with a traversal of C' from v_i that only uses black edges in C'. As the number of black neighbors of a node is bounded by Δ_{black} this takes $O(|C'|\Delta_{black})$. As $|C'| \leq |K| + 1$ the cost $O(q\Delta_{black})$ follows. □

Lemma 12. *A recursive call of Algorithm 1 takes $O(q^2\Delta^2_{black}(\Delta_G + \Delta_H))$ time.*

Proof. Checking wether K is maximal takes $O(q\Delta_{black}(\Delta_G+\Delta_H))$ by Lemma 10. Computing X (line 3) can be done in $O(q\Delta_{black})$ by iterating over the black neighbors of all nodes in K, and it gives us $|X| \leq q\Delta_{black}$. The loop is executed $|X|$ times, and each iteration costs the sum of lines 5 and 6. Line 5 takes $O(q\Delta_{black})$ by Lemma 11, and line 6 takes $O(q\Delta_{black}(\Delta_G + \Delta_H))$ by Lemma 10, as it can be done by checking that maximalizing $C \setminus \{max(C)\}$ in $P_{\prec max(C)}$ yields K, and maximalizing C in $P_{\prec max(C)} \cup \{max(C)\}$ yields C. The total cost is thus dominated by the cost of the loop, that is $O(q\Delta_{black}(q\Delta_{black}+ q\Delta_{black}(\Delta_G + \Delta_H))) = O(q^2\Delta^2_{black}(\Delta_G + \Delta_H))$ □

As for each recursive call that outputs a BC-clique there are at most $q-1$ calls which do not, the total cost per solution is given by the following.

Theorem 1. *Algorithm 1 has cost per* BC*-clique equal to $O(q^3\Delta^2_{black}(\Delta_G + \Delta_H))$ time and $O(q|V_G||V_H|)$ space.*

Proof. Either a solution is output in line 2 or K is not maximal, meaning that it can be fully extended by one or more nodes in X. Let v be the smallest of such nodes; $K' = K \cup \{v\}$ is certainly a good child of K, as there are no smaller candidates, nor nodes smaller than $max(K)$, that can fully extend K', and $K' \setminus max(K') = K' \setminus \{v\} = K$ thus K is the parent of K'.

The same is true for K', which will either be maximal or produce a child which includes K'. Finally a descendant recursive call will output a maximal BC-clique $K'' \supseteq K' \supset K$. As in every recursive call a node is added to the temporary result, the number of such calls is bounded by $|K''|$. Thus, for each recursive call that outputs a BC-clique there are at most $q - 1$ calls which do not output a BC-clique.

It follows that the cost per solution of Algorithm 1 is bounded by q times the cost of a recursive call, i.e. $O(q^3\Delta^2_{black}(\Delta_G + \Delta_H))$. As for the space, we note that a single recursive call only needs to store the current solution, of size $O(q)$. Furthermore, the recursion depth is at most the number of nodes in P, i.e., at most $|V_G| \cdot |V_H|$, since the approach always tries to add a larger node to the current solution. Thus, the total space usage is bounded by $O(q|V_G| \cdot |V_H|)$. □

3. Application to Find LACCIS's in Proteins

This section shows how to use our solution for the T-MCCIS problem to obtain a new heuristic, called FLASH, to quickly find LACCIS's for two labeled undirected graphs G and H. In Section 3.1 we show how FLASH works. We then validate FLASH in the subsequent sections. In particular, in Section 3.2 we show how we generated our testbed data while in Section 3.3 we show our experimental results.

3.1. Turning T-MCCIS's into LACCIS's

FLASH aims to find LACCIS's for two labeled undirected graphs G and H. It solves the T-MCCIS problem for a set of given (random)[2] spanning trees of G $\{T_1, \ldots, T_k\}$, obtaining the set of T-MCCIS's for each $T \in \{T_1, \ldots, T_k\}$. It then applies two phases.

Filtering. It is important to distill all the T-MCCIS's found for each T. Those leading to the same LACCIS's are clearly redundant, and those that are small or mostly overlapping prevent us from making sense of a massive output. The FILTER procedure scans the found T-MCCIS's, giving priority to large ones and excluding the ones smaller than a given minimum size τ, and incrementally adds them to a "cover" set if their overlap with every other isomorphism in the set is smaller than σ. Namely we retain T-MCCIS iff, for either its subgraphs of G or H, the number of common nodes with any other T-MCCIS in the cover divided by its size is smaller than σ.

Recombining. As observed in the introduction, the T-MCCIS found may be fragments of larger (maximal) common subgraphs. To enlarge them, FLASH merges and uses FILTER on the output of all trees, then runs a RECOMBINE procedure which combines *compatible* T-MCCIS to generate larger LACCIS's. Two T-MCCIS's are compatible if they can be (partially) merged and their

[2]Other non-random selection strategies for the spanning trees can be considered, but we found no significant difference in the result.

induced subgraphs in G and H are connected by one or more edges: RECOMBINE takes the largest part of the second T-MCCIS that can be added to the first and creates a larger LACCIS by merging them. This process is repeated as long as new LACCIS's are created.

After that, FILTER is applied again to remove redundant and partially overlapping isomorphisms, if any. We refer to the sequence of operations FILTER, RECOMBINE, FILTER as PROCESS. The resulting isomorphisms identify LACCIS's composed of parts of the T-MCCIS's for $T = T_1, \ldots, T_k$. This is the final output of FLASH.

3.2. From Proteins to Graphs: generating the testbed data

FLASH can bring benefits when modeling proteins as graphs since higher resolution can be exploited. Current approaches benefit from a reduced computational load as they use coarse-grained models. For example, the 3D patterns of secondary structure elements in proteins have been modeled as graphs by using secondary structure elements, such as the α-helices and the β-strands, as nodes. They are approximately linear structures and they are represented as vectors in space, sometimes annotated with the length of their residues and hydrophobicity. As for the edges, they represent relationships between nodes expressed in terms of the angles and the distance between midpoints of the corresponding vectors [13]. In another representation nodes are represented similarly, but edges are calculated on the basis of contacts between the atoms belonging to the respective structures/nodes, and indicate the spatial arrangements of the structures. In this way structural patterns can be also found in proteins with weaker similarities [3]. We refer the reader to Table 1 in [13] for a list of applications.

We think that exploring fine-grained models with FLASH, which was precluded with previous algorithms, can give finer details once data noise is filtered. However the design and validation of a finer-grained model is outside the scope of this paper, and deserves further independent study. Consider Fig. 4, which shows an example of the coarse-grained model adopted in [3].[3] Indeed, note how the two proteins, which would have respectively 2763 (9488) and 3841 (12923) nodes (edges) in the all-atom representation, are reduced to graphs which are orders of magnitude smaller. While patterns in such graphs can indeed be valuable, the loss of information may be significant.

[3]The graphs can be found at `http://ptgl.uni-frankfurt.de/`

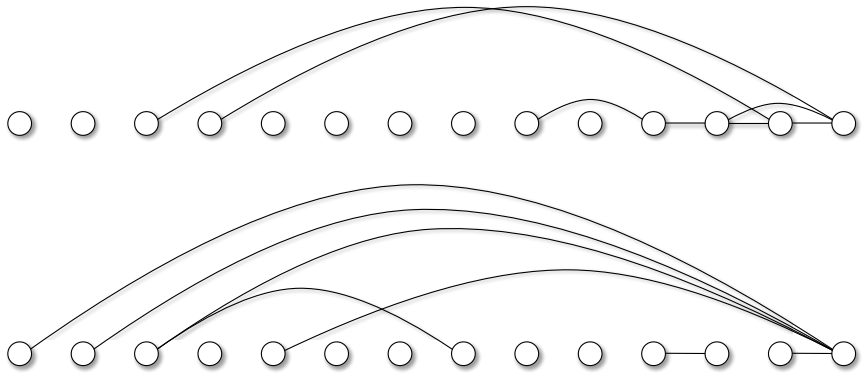

Figure 4: Proteins `1ald` (above) and `1fcb` (below) as modeled in [3]. Nodes correspond to α-helices and edges connect close structures.

We created a stress test with an all-atom fine-grained model for generating graphs from proteins from the PDB (`www.rcsb.org`). We thus exploited PDB data of `1ald`, `1fcb` (chain A), and `1gox` proteins (which belong to TIM barrel families) to generate graphs where labeled nodes represent atoms within known secondary structures (as reported in PDB) while edges represent covalent bonds (both backbone and non-backbone) as well as non-covalent interactions.

We generated input graphs by means of `pdb2graph` [47]. First, PDB data is processed to generate edges from covalent bonds. Non-covalent interactions are estimated by extending the interaction distance up to 3.2 Å. Nodes are labeled with the element symbol and a secondary structure identifier according the PDB data. We thus generated 3 graphs with 2763 (9488), 3841 (12923), and 2696 (9059) nodes (edges) for `1ald`, `1fcb`, and `1gox` respectively. Furthermore, we also considered two variants of a structure extracted from `1ald` to test consistency and robustness of FLASH, discussed later.

3.3. Experimental results

We describe our experimental results for FLASH. In order to better understand its performance, we analyzed FLASH by considering the aggregated *raw* result after its output sensitive search, and the *post*-PROCESS form after filtering and recombining the latter results by the method PROCESS. Specifically, we fixed $\tau = 10$ for the threshold on minimal size and $\sigma = 70\%$ for the overlapping threshold of FILTER (recall that PROCESS indicates the sequence FILTER, RECOMBINE, FILTER).

We chose to run FLASH with k random spanning trees for several values of k and with a set of spanning trees which forms a cover of the graph G (in

our test case the number of spanning tree covering G was 5). We refer to the former variant as k-FLASH, with $k = 1, 3, 6, 12$, and to the latter as c-FLASH. As for the raw result, FLASH runs k threads, one for each spanning tree, which (are forced to) terminate within a fixed time t, and then aggregates their results. We let each thread run for at most t/k hours, so that the bound for the overall CPU time is the same for all runs, with $t = 12$ hours.

Following the discussion in the introduction, the baseline for the comparison of FLASH is Koch's algorithm [2], due to its popularity, so that LACCIS's are obtained using MCCIS's [39].[4] For a fair comparison, we optimized its implementation, denoted KOCH, so that it can use the implicit product graph P as well, noting that this optimization greatly improves its performance [44]; moreover, the computation is terminated after fixed time $t = 12$ hours. We remark that the RECOMBINE step has no effect on KOCH, whose output is made of MCCIS's that cannot be enlarged, and thus its additional time is negligible.

The above framework has been implemented in `C++`. Our computing platform is a 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, with 128GB of shared memory. The operating system is Ubuntu 14.04.2 LTS, with Linux kernel version 3.16.0-30.

For each pair of graphs in Table 1, we report the real execution time, that is the time (bounded by t/k hours for RAW) of the threaded execution PAR, and the total CPU time WORK (bounded by $t = 12$ hours for RAW). Note that WORK of FLASH for RAW is less than t in all the cases as almost all the threads terminate earlier than the time limit t/k. We also report some analysis of the results, before and after applying PROCESS, in columns RAW and post PROCESS respectively. For each result set, we show the size of the greatest LACCIS in this set (i.e., MAX), the maximum h such that there are at least h LACCIS's of size h, (i.e., H-IND), and the number of LACCIS's found (i.e., COUNT).

On the choice of the spanning trees. Referring to the upper part of Table 1, given the pair of graphs `1ald` and `1fcb`, we compare the results of our k-FLASH for different values of k and of c-FLASH. It is worth observing (RAW column) that the number of T-MCCIS's found increases with the number of spanning trees used, recalling that c-FLASH uses 5 spanning trees, 6-FLASH and 12-FLASH produce a higher number of T-MCCIS's. After the

[4] As already discussed, the backtracking algorithms for the maximum common subgraph, which are faster than MCS and can operate on large graphs [18], employ a cutting rule unsuitable for LACCIS's.

	RAW					post PROCESS				
	TIME(h)					TIME(h)				
METHOD	PAR	WORK	MAX	H-IND	COUNT	PAR	WORK	MAX	H-IND	COUNT
`1ald` vs `1fcb`										
1-FLASH	0:12	0:13	63	59	9 215 182	0:01	0:06	70	39	17 468
3-FLASH	0:12	0:25	62	62	22 165 459	0:31	0:45	179	43	29 082
6-FLASH	0:29	1:01	59	58	47 329 927	0:24	1:05	155	48	41 080
12-FLASH	0:30	2:28	70	69	107 383 973	15:41	17:27	229	53	55 231
c-FLASH	0:13	0:40	55	54	38 315 376	0:09	0:43	118	42	41 410
KOCH	12	12	63	63	1 297 231	<0:01	<0:01	63	24	170
`1ald` vs `1gox`										
6-FLASH	0:32	1:53	68	68	60 120 366	4:30	5:10	68	49	26 954
KOCH	12:00	12:00	64	64	4 775 963	<0:01	<0:01	64	6	6
`1fcb` vs `1gox`										
6-FLASH	2:08	8:18	153	151	144 658 776	0:13	1:08	153	47	26657
KOCH	12:00	12:00	82	82	4 412 419	<0:01	<0:01	82	25	158
`HelixD-1ald` vs `1ald`										
6-FLASH	2:01	9:00	171	170	58 925 057	0:07	0:25	171	38	6 561
KOCH	12:00	12:00	60	60	197 236	<0:01	<0:01	60	2	2
`mod-HelixD-1ald` vs `1ald`										
6-FLASH	0:10	0:18	162	160	6 876 538	0:02	0:03	162	35	7 592
KOCH	12:00	12:00	60	60	81 884	<0:01	<0:01	65	2	2

Table 1: Experimental results

post-processing, c-FLASH and 6-FLASH produce a similar number of LACCIS's, while 12-FLASH produces a larger number of LACCIS's but at the price of a higher post-processing time. For these reasons, we decided to focus on 6-FLASH in the remaining experiments in this section.

Running the Experiments. For the following pairs of graphs, `1ald` vs `1fcb`, `1ald` vs `1gox`, and `1fcb` vs `1gox`, we report the results for both KOCH and 6-FLASH. We remark how our algorithm, though heuristic, finds in this given time slot more LACCIS's than KOCH, whose result set includes in theory all the BC-cliques. Moreover, it seems that FLASH is able to find larger LACCIS's than KOCH, as shown in the post PROCESS columns, where LACCIS's found by FLASH are greatly enlarged. Furthermore, it is clear that KOCH focuses the search on a limited portion of the graph, while FLASH is able to produce many more LACCIS's that do not overlap with each other (see COUNT in post PROCESS). For instance, consider the `1ald` vs `1gox` comparison. Even though KOCH finds 4 775 963 LACCIS's, after PROCESS we are left with just 6, of size at most 64: this means that all the remaining LACCIS's found by KOCH overlap with these 6 by at least σ =70%. This is not the case with FLASH, which obtains 26 954 LACCIS's after PROCESS. Even though our post PROCESS time is greater, this is compensated by the quantity and quality of our results, as well as the smaller running time of

the algorithm.

Consistency and robustness. To test the consistency and quality of the results, we extracted a portion of the protein `1ald` (from `Pro158` to `Asn180`), including an α-*helix*, and searched for it in the original protein. The corresponding subgraph has 171 nodes and 584 edges. Clearly, an effective algorithm must find at least a LACCIS which involves a large portion of the helix. Table 1 (`HelixD-1ald` vs `1ald`) shows that our algorithm finds a LACCIS involving the whole helix (171 nodes), while this is not the case for KOCH in the time slot.

For the robustness, we introduced errors in the helix: we changed the labels of the alpha carbon atoms of `Arg172` and `Asn166` to a dummy label 'X'. We refer to this modified graph as `mod-HelixD-1ald`. A robust algorithm should not be significantly influenced by the introduced noise, and should find results similar to the ones obtained with `HelixD-1ald`. The lower part of Table 1, i.e. `mod-HelixD-1ald` vs `1ald`, shows that both the algorithms are robust in this sense, as both are consistent with the previous results. As a consequence, even in the presence of slight changes, our algorithm favorably compares with KOCH, as it finds almost the whole helix, while KOCH finds just a small portion of the helix.

4. Conclusion

A great body of research is dedicated to the problem of finding the maximum common subgraph, which is fast in practice but provides a limited result. On the other hand, finding *all* maximal common connected induced subgraphs in large molecules is known to be more challenging. We proposed the first approach that improves significantly upon simple adaptations of the Bron-Kerbosch algorithm. Our approach aims at making the search for common subgraphs feasible for larger graphs. Even though some subgraphs can be missed, we find in practice much more and much larger results than existing approaches within a reasonable time limit. We believe that our contribution can support higher quality analysis of macromolecules by enabling researchers to find a larger number of meaningful common structures between two molecules.

Acknowledgments

This work has been partially supported by MIUR under PRIN 2012C4E3KT national research project AMANDA — Algorithmics for MAssive and Networked DAta and by U. Pisa under PRA 2015 project Computational Meth-

ods for Personalized Medicine. Alessio Conte is supported by JST CREST, Grant Number JPMJCR1401, Japan.

References

[1] Y. Cao, A. Charisi, L. Cheng, T. Jiang, T. Girke, Chemminer: a compound mining framework for r, Bioinformatics 24 (15) (2008) 1733–1734.

[2] I. Koch, Enumerating all connected maximal common subgraphs in two graphs, Theor Comput Sci 250 (1) (2001) 1–30.

[3] I. Koch, T. Lengauer, E. Wanke, An algorithm for finding maximal common subtopologies in a set of protein structures, J Comput Biol 3 (2) (1996) 289–306.

[4] J. Huan, W. Wang, J. Prins, J. Yang, Spin: mining maximal frequent subgraphs from graph databases, in: Proc. of the tenth ACM SIGKDD, ACM, 2004, pp. 581–586.

[5] G. Levi, A note on the derivation of maximal common subgraphs of two directed or undirected graphs, CALCOLO 9 (4) (1973) 341–352.

[6] A. Conte, R. Grossi, A. Marino, L. Versari, Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques, in: 43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy, 2016, pp. 148:1–148:15.

[7] A. Conte, R. Grossi, A. Marino, L. Tattini, L. Versari, A fast algorithm for large common connected induced subgraphs, in: Algorithms for Computational Biology - 4th International Conference, AlCoB 2017, Aveiro, Portugal, June 5-6, 2017, Proceedings, 2017, pp. 62–74.

[8] A. Conte, R. Grossi, A. Marino, L. Tattini, L. Versari, A fast algorithm for large common connected induced subgraphs, WEPA 2016 : First Workshop on Enumeration Problems and Applications 21-22 Nov 2016 Aubière (France) (2016).

[9] P. Artymiuk, R. Spriggs, P. Willett, Graph theoretic methods for the analysis of structural relationships in biological macromolecules, J AM Soc Inf Sci Tec 56 (5) (2005) 518–528.

[10] D. Bonchev, Chemical graph theory: introduction and fundamentals, CRC, 1991.

[11] E. Gardiner, P. Artymiuk, P. Willett, Clique-detection algorithms for matching 3-dimensional molecular structures, J Mol Graph Model 15 (1997) 245 – 253.

[12] P. Artymiuk, A. Poirrette, H. Grindley, D. Rice, P. Willett, A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures, J Molecular Biology 243 (2) (1994) 327–344.

[13] R. Van Berlo, W. Winterbach, M. De Groot, A. Bender, P. Verheijen, M. Reinders, D. de Ridder, Efficient calculation of compound similarity based on maximum common subgraphs and its application to prediction of gene transcript levels, Int J Bioinformatics Res Appl 9 (4) (2013) 407–432.

[14] M. Oh, T. Yamada, M. Hattori, S. Goto, M. Kanehisa, Systematic analysis of enzyme-catalyzed reaction patterns and prediction of microbial biodegradation pathways, J Chem Inf Model 47 (4) (2007) 1702–1712.

[15] R. Sheridan, S. Kearsley, Why do we need so many chemical similarity search methods?, Drug Discov Today 7 (17) (2002) 903 – 911.

[16] F. Abu-Khzam, Maximum common induced subgraph parameterized by vertex cover, Inf Process Lett 114 (3) (2014) 99–103.

[17] L. Brun, B. Gaüzère, S. Fourey, Relationships between Graph Edit Distance and Maximal Common Unlabeled Subgraph, Tech. rep. (Jul. 2012).

[18] D. Conte, P. Foggia, M. Vento, Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs., J Graph Algorithms Appl 11 (1) (2007) 99–143.

[19] V. Kann, On the approximability of the maximum common subgraph problem, in: Proc. of STACS '92, Lect Notes Comput Sci, 1992, pp. 377–388.

[20] J. Raymond, E. Gardiner, P. Willett, Rascal: Calculation of graph similarity using maximum common edge subgraphs, Comput J 45 (2002) 2002.

[21] H. Barrow, R. Burstall, Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques, Inf Process Lett 4 (1976) 83–84.

[22] A. Brint, P. Willett, Algorithms for the identification of three-dimensional maximal common substructures, J Chem Inf Comput Sci 27 (4) (1987) 152–158.

[23] X. Huang, J. Lai, S. Jennings, Maximum common subgraph: some upper bound and lower bound results, BMC Bioinformatics 7 (Suppl 4) (2006) S6.

[24] H. Ehrlich, M. Rarey, Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review, Wiley Interdisciplinary Reviews: Computational Molecular Science 1 (1) (2011) 68–79.

[25] M. Cone, R. Venkataraghavan, F. McLafferty, Molecular structure comparison program for the identification of maximal common substructures, J Am Chem Soc 99 (23) (1977) 7668–7671.

[26] Y. Cao, T. Jiang, T. Girke, A maximum common substructure-based algorithm for searching and predicting drug-like compounds, Bioinformatics 24 (13) (2008) i366–i374.

[27] T. Akutsu, T. Tamura, A polynomial-time algorithm for computing the maximum common connected edge subgraph of outerplanar graphs of bounded degree, Algorithms 6 (1) (2013) 119.

[28] W. Suters, F. Abu-Khzam, Y. Zhang, C. Symons, N. Samatova, M. Langston, A new approach and faster exact methods for the maximum common subgraph problem, in: Computing and combinatorics, 2005, pp. 717–727.

[29] E. Krissinel, K. Henrick, Common subgraph isomorphism detection by backtracking search, Software: Practice and Experience 34 (6) (2004) 591–607.

[30] J. Mcgregor, Backtrack search algorithm and the maximal common subgraph problem, in: Software | Practice and Experience 12, 1982, pp. 23–34.

[31] T. Wang, J. Zhou, Emcss: A new method for maximal common substructure search, J Chem Inf Comput Sci 37 (5) (1997) 828–834.

[32] A. Gupta, N. Nishimura, Finding largest subtrees and smallest supertrees, Algorithmica 21 (2) (1998) 183–210.

[33] C. Bron, J. Kerbosch, Finding all cliques of an undirected graph (algorithm 457), Commun ACM 16 (9) (1973) 575–576.

[34] R. Carraghan, P. Pardalos, An exact algorithm for the maximum clique problem, Oper Res Lett 9 (6) (1990) 375 – 382.

[35] D. Breuker, P. Delfmann, H.-A. Dietrich, M. Steinhorst, Graph theory and model collection management: conceptual framework and runtime analysis of selected graph algorithms, Information Systems and e-Business Management 13 (1) (2015) 69–106. doi:10.1007/s10257-014-0243-6.
URL http://dx.doi.org/10.1007/s10257-014-0243-6

[36] T. Fober, M. Mernberger, G. Klebe, E. Hüllermeier, Graph-based methods for protein structure comparison, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 3 (5) (2013) 307–320. doi:10.1002/widm.1099.
URL http://dx.doi.org/10.1002/widm.1099

[37] B. Cuissart, J.-J. Hébrard, A direct algorithm to find a largest common connected induced subgraph of two graphs, in: International Workshop on Graph-Based Representations in Pattern Recognition, Springer, 2005, pp. 162–171.

[38] P. Vismara, B. Valery, Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms, in: Modelling, Computation and Optimization in Information Sys. and Management Sciences, 2008, pp. 358–368.

[39] R. Welling, A performance analysis on maximal common subgraph algorithms, in: 15th Twente Student Conference on IT, The Netherlands, Un. of Twente, 2011.

[40] Y. Yuan, G. Wang, L. Chen, H. Wang, Graph similarity search on large uncertain graph databases, The VLDB Journal 24 (2) (2015) 271–296.

[41] A. Conte, R. Grossi, A. Marino, L. Versari, Finding maximal common subgraphs via time-space efficient reverse search, in: 24th International Computing and Combinatorics Conference, COCOON 2018, July 2-4, 2018, Qingdao, China. To appear, 2018.

[42] J. Ullmann, An algorithm for subgraph isomorphism, J ACM 23 (1) (1976) 31–42.

[43] A. Droschinsky, B. Heinemann, N. Kriege, P. Mutzel, Enumeration of maximum common subtree isomorphisms with polynomial-delay, in: International Symposium on Algorithms and Computation, Springer, 2014, pp. 81–93.

[44] L. Versari, Ricerca veloce di pattern comuni a due grafi, bachelor Thesis (in Italian), University of Pisa (2015).

[45] E. L. Lawler, J. K. Lenstra, A. Rinnooy Kan, Generating all maximal independent sets: Np-hardness and polynomial-time algorithms, SIAM Journal on Computing 9 (3) (1980) 558–565.

[46] D. Avis, K. Fukuda, Reverse search for enumeration, Discrete Appl Math 65 (1) (1996) 21–46.

[47] L. Holder, PDB-to-graph program, `https://github.com/mikeizbicki/datasets/tree/master/graph/pdb2graph`, accessed: 2016-05-04 (2015).