A Fast Algorithm for Large Common Connected Induced Subgraphs

Alessio Conte[1], Roberto Grossi[1], Andrea Marino[1](✉), Lorenzo Tattini[2], and Luca Versari[3]

[1] Inria, Università di Pisa and Erable, Pisa, Italy
{conte,grossi,marino}@di.unipi.it
[2] IRCAN, CNRS UMR, 7284 Nice, France
lorenzo.tattini@unice.fr
[3] Scuola Normale Superiore, Pisa, Italy
luca.versari@sns.it

Abstract. We present a fast algorithm for finding large common subgraphs, which can be exploited for detecting structural and functional relationships between biological macromolecules. Many fast algorithms exist for finding a single maximum common subgraph. We show with an example that this gives limited information, motivating the less studied problem of finding many large common subgraphs covering different areas. As the latter is also hard, we give heuristics that improve performance by several orders of magnitude. As a case study, we validate our findings experimentally on protein graphs with thousands of atoms.

Keywords: Proteins · Structure similarity · Isomorphisms · Graphs · Listing

1 Introduction

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 [2]. 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 [4], 3D patterns for proteins [17], amino acid side-chains [1], and compound similarity for the prediction of gene transcript levels [25], to name a few.

Context for the Study. We are interested in common subgraphs between two given input graphs G and H. Recalling that a subgraph S of G is a subset of its nodes and connecting edges, S is said to be common with H if S is isomorphic to a subgraph of H: S is maximal if no other common subgraph 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 classically

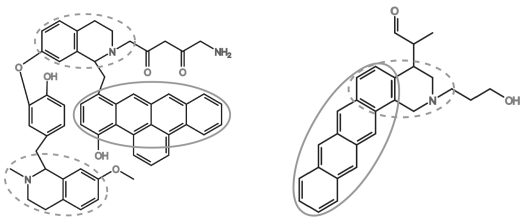

Fig. 1. Common structures in Liensinine derivatives. The two molecules share a tetracenic moiety (large circles) and an N-methylbenzopiperidinic moiety (smaller dotted circles). The maximum common structure is however a fairly common structure in organic molecules; the structure in the dotted circle which is maximal (but not maximum) is more likely to be interesting as it is more peculiar.

related to structural similarity. The *maximal* common subgraph (MCS) problem requires finding all the MCS's of G and H. The MCS problem can be constrained to *connected* and *induced* subgraphs (MCCIS) [8,16,17]: the latter means that all edges of G between nodes in the MCS are mapped to edges of H, and vice versa.

Maximum vs Maximal. Maximum common subgraphs are very often confused with MCS's, but they are *different* concepts: it is much faster in practice to find the maximum common subgraph size than all MCS's (e.g. [12]). As discussed later, while there are many results for the former, not much algorithmic research has been done in the past 20 years for MCS's, and the seminal results in [16,17] are still the state of the art. Note that 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, as shown for the two molecules represented by the graphs in Fig. 1. In general, there may be arbitrarily large common substructures that give few information because of their frequent appearance in special type of macromolecules or polymers. 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.

Problem of Interest. It is well known that finding similar structures leads to highlighting similar biochemical properties and functions [22]. To this aim, we focus on *large* common connected induced subgraphs (LACCIS's). Indeed, considering *induced* and *connected* subgraphs reduces the search space and the number of solutions, while preserving the most significant ones [8,16,17]. To quickly find LACCIS's, we consider a modified version of the MCCIS problem: given a spanning tree T of G, we are interested in the common subgraphs between G and H for which their subgraph in G is connected using edges of T. We call these subgraphs T-MCCIS's (see Fig. 2 for an example).

For a set of spanning trees $T_1, \ldots, T_k$, we consider a 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

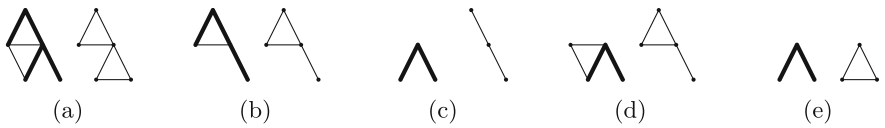

Fig. 2. (a) Two graphs G and H, where edges of T are shown as thicker; (b) a T-MCCIS that is also MCCIS; (c) a T-MCCIS; (d) not a T-MCCIS since it is a MCCIS but not spanned by T; (e) not a T-MCCIS since it is spanned by T but is not a common induced subgraph.

among induced subgraphs. Hence the larger L is, the closer is to a MCCIS. Note that L implicitly establishes an *isomorphism* between sets of matching nodes of G and H. However, this isomorphism is not unique: for example, if L is a clique of q nodes (i.e. all pairwise connected), it can be mapped through $q!$ isomorphisms.

When the nodes of G and H are labeled, the notion of LACCIS naturally extends by requiring, for instance, the nodes of the LACCIS to have matching labels, or considering a generalized compatibility function between nodes or edges.

Contributions. In this paper we provide an algorithm, called FLASH (Fast LAccis Searching Heuristic), that takes two connected labeled graphs G and H as input, along with some (random) spanning trees $T_1, \ldots, T_k$ of G, 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. For a spanning tree $T \in T_1, \ldots, T_k$, FLASH explores a variation of the product graph P [19] obtained from G and H, so that LACCIS are found as special cliques in P. FLASH does *not* materialize P, but navigates it implicitly to improve memory usage and running time, and produces T-MCCIS's at a fixed rate by employing a refined variation of an output-sensitive algorithm to find maximal cliques [11]. We remark that spanning trees have been previously employed to prune the search for frequent subgraphs [14], although such techniques do not extend to this problem.

FLASH applies the above approach to 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 selecting larger sized ones). This filter shows that FLASH quickly finds solutions spanning different parts of G and H, whereas other approaches such as [17] 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.

Since FLASH could miss some maximal subgraphs (i.e. the ones not spanned by T_i for any i, as in Fig. 2d), it exploits the fact that the number of T-MCCIS's after filtering is relatively small, and performs a postprocessing to combine them with the purpose of enlarging the common subgraphs thus found (e.g. the subgraph in Fig. 2d is discovered if a non-spanned edge is spanned by another choice of T). The novelty of our approach is that the running time for a given spanning tree T is provably proportional to the number of reported T-MCCIS's, as confirmed by our experiments in Sect. 3. In other words, the more we pay in running time, the more T-MCCIS's we get. This is in contrast with the known algorithms for maximal common subgraphs that could be adapted to discover T-MCCIS's. They have the drawback of running into a computational black-hole, going through an explosive number of substructures even if there are few T-MCCIS's: for a T-MCCIS of k nodes, these algorithms have to potentially discard 2^k included subgraphs, and branch and bound does not help much in this case.

As a result, FLASH finds more solutions than other approaches and improves their performance by several orders of magnitude. When dealing with graphs of non-trivial size (e.g., thousands of nodes) we argue that the state-of-the-art approaches for 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 (see Sect. 3) and FLASH performs well in practice even though its theoretical worst-case complexity is exponential.

Case Study with Proteins. The computational power of FLASH can bring benefits when modeling macromolecules such as proteins as graphs. To create a stress test for FLASH we adopted a fine-grained model, called all-atom, for representing the proteins `1ald`, `1fcb`, and `1gox` from the Protein Data Bank (PDB), where the labeled nodes represent atoms within known secondary structures while the labeled edges represent covalent bonds (both backbone and non-backbone) as well as non-covalent interactions.

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 α-helices and β-strands, as nodes. These elements 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 [25]. In another representation, edges are calculated on the basis of contacts between the atoms in the respective structures/nodes, and indicate the spatial arrangements of the structures. In this way patterns can be also found in proteins with weaker similarities [17]. We refer the reader to Table 1 in [25] 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 fine-grained model is outside the scope of this paper, and deserves further independent study. Future investigations will be devoted also to the definition of a scoring function to rank the solutions produced by FLASH, and the application of our approach to problems where knowledge can be extracted from structural similarity, such as protein annotation.

Related Work. Both the problems of finding maximal or maximum common subgraphs has been studied for decades [5,9,25]. The corresponding decision version is NP-complete as it solves subgraph isomorphism problem, even on some restricted graph classes such as outerplanar graphs. The problem is difficult to approximate (MAX-SNP hard) even within a polynomial factor [15]. This motivates the search for effective heuristics.

Due to the strong connection between graph isomorphism and common substructures, the bioinformatics community has repeatedly expressed its interest in this problem from a computational point of view [7,12,15,17,21,25]. Finding the maximum common subgraph, as opposed to finding all maximal ones, allows for very effective cuts to the search space (e.g. branch-and-bound). This makes the computation much faster in practice, allowing researchers to process larger graphs with the available resources. However, as previously noted, their cut rules cannot be applied efficiently to LACCIS's.

Looking at previous work for graphs, it can be roughly classified into two categories: clique-based methods [17,23], non-clique-based backtracking methods [18,20,27].

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 [19] and has been shown to be effective on biological networks [12,17,23]. Algorithms such as the ones by Koch [16,17] are the state of the art [28], having laid down the basic principles for clique-based approaches; however, a tool able to efficiently enumerate LACCIS has not yet emerged. We will compare experimentally FLASH to the algorithms in [16,17]. For finding the maximal cliques, the algorithms by Bron and Kerbosch [6] or Carraghan and Pardalos [10] have been employed. We will use a refined variant of [11]. Cao et al. [9] observe that materializing P can be memory-wise expensive, and FLASH is able to avoid this issue.

Backtracking algorithms mostly build up on Ullman's strategy [24] for subgraph isomorphism (e.g. [20]). They often use branch-and-bound heuristics based on the specific requirements of the application at hand. The comparison in [12] shows how direct implicit methods for the maximum common subgraph, such as the one in [20], 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 LACCIS's.

2 Methods

We give the main ideas behind FLASH for two labeled undirected graphs G and H, and a set $\{T_1, \ldots, T_k\}$ of (random) spanning trees of G. Let $T \in \{T_1, \ldots, T_k\}$.

Implicit Product Graph. We employ a variant of the transformation adopted by Koch [17] and borrowed from Levi [19], where we modify the color rule to take into account the edges of the spanning tree T. Define a *colored product graph* $P = G \cdot H$, with $P = (V_P, E_P)$. The nodes in V_P corresponds to ordered pairs of compatible nodes from G and H (e.g., with the same label), the first from G and the second from H, and the edges in E_P are as follows (where x and y denotes any two nodes in G and i and j any two nodes in H). There is a *black* edge in E_P between the nodes in V_P corresponding to (x, i) and (y, j) if

$\{x, y\} \in T$ (tree edge) and $\{i, j\} \in H$. There is a *white* edge in E_P between the nodes in V_P corresponding to (x, i) and (y, j) if either $\{x, y\} \in G \backslash T$ (non-tree edge) and $\{i, j\} \in E$, or both $x \neq y$ and $i \neq j$ are not connected by an edge in their graphs (resp. G and H). As in [17], there is a one-to-one correspondence between maximal cliques in P and maximal isomorphisms between 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. FLASH avoids storing P explicitly, and only stores G and H: it checks compatibility between assignments in constant time and iterates on neighbors in constant time per element by applying "on the fly" the rules used for generating P. This saves both memory and time, as G and H are much smaller and faster to access than P.

We now have to find the BC-cliques in $P = (V_P, E_P)$. Previous work on explicit product graphs employs a modified version of the Bron-Kerbosch algorithm [16] which does not perform *pivoting*, a pruning technique. The resulting algorithm thus iterates on every possible subset of each common subgraph, and its complexity and cost per solution are not clearly bounded. We do not reuse the above algorithm and use a different approach for FLASH as shown next.

Good Ordering. We use the spanning tree T to provide a *good ordering* of the nodes in $V(P)$. Let the nodes of G be numbered in such a way that, given a node u of G, there is only *one* edge of T between u and its neighbors u' with $u' < u$. This numbering can be computed by a pre-order visit of T, and induces the lexicographical order $\prec$ on the pairs (x, i) that corresponds to the nodes in $V(P)$. The *good ordering* of the nodes in V_P is obtained by numbering them consecutively in increasing lexicographical order $\prec$ of the corresponding pairs.

The T-based numbering of G's nodes induces an ordering of P's nodes that has the following *property*, whose proof is not given here for the sake of space. Let $P_{<v}$ denote the subgraph of P induced by the nodes $v' < v$, and $P_{<v} \cup \{v\}$ the one induced by $v' \leq v$, if C is a BC-clique in $P_{<v} \cup \{v\}$, then $C \backslash \{v\}$ is connected with black edges in $P_{<v}$.

This property ensures that every BC-clique can be found incrementally by FLASH: when adding a new node v to the set of BC-cliques found up to that point for the nodes of $P_{<v}$, 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 combinations of them. This speeds up significantly the computation.

Output-Sensitive Search. We perform an output-sensitive search of the BC-cliques by adapting to this problem a recent maximal clique enumeration algorithm [11]. The latter runs as fast as Bron-Kerbosch *with* pivoting, but it has the additional feature of guaranteeing that the total running time only depends on the number of returned BC-cliques's times a polynomial, meaning that a long execution will always yield many results. Furthermore, it allows us to define a parent-child relationship between partial solutions: this produces a stateless, memory-efficient search which avoids generating duplicate solutions. We adapted it to work on the implicit product graph P using the good ordering as it grows partial results by incrementally adding new nodes to them (see Sect. 4).

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 add them to a "cover" set if their overlap with every other isomorphism in the set is smaller than σ. Namely we retain T-MCCIS if, 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 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 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 final output of FLASH identifies LACCIS's composed of parts of the T-MCCIS's for $T = T_1, \ldots, T_k$.

3 Results

In this section, we describe our experimental results for FLASH. We considered 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 (see Sect. 2). 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

Table 1. Experimental results

METHOD	RAW					post PROCESS				
	TIME (h)		MAX	H-IND	COUNT	TIME (h)		MAX	H-IND	COUNT
	PAR	WORK				PAR	WORK			
`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

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 $t = 12$ h.

Following the discussion in the introduction, the baseline for the comparison of FLASH is Koch's algorithm [16], which produces LACCIS's using MCCIS and is the best algorithm known so far for MCCIS [28]. 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 [26]; the computation is terminated after fixed time $t = 12$ h. Note 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++` and is available at `github.com/veluca93/laccis`. Our platform is a 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz, with 128 GB of shared memory. The system is Ubuntu 14.04.2 LTS, with Linux kernel 3.16.0-30.

We give a quick tour on our experimental study. First, we explain how we generated the data which has been used for the testbed of our experiments. Next, we show the experimental measures we have considered and we discuss how the choice of the spanning trees affects our analysis. We then analyze the experimental outcome for pairs of proteins, running FLASH and KOCH. Finally, we analyze the quality and consistency of the results returned with a protein and one of its sub-parts as input; also, we argue that FLASH is robust when perturbations of the input are introduced, e.g. node labels change.

Generating the Testbed Data. As mentioned in the introduction, 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` [13]. 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. 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.

Experimental Measures. 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 $t = 12$ h 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 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 an 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 is in theory complete. 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 $\sigma = 70\%$. 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. 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`. Table 1 (`mod-HelixD-1ald` vs `1ald`) shows that our algorithm still finds almost the whole helix, while KOCH, although consistent with the previous result, finds just a small portion of the helix.

4 Implementation

In this section, we give more details about our algorithm to find BC-cliques in the implicit product graph P. First of all, we observe that a BC-clique in $P_{<v} \cup \{v\}$ is either a BC-clique of $P_{<v}$ or contains v. Moreover, a BC-clique C containing v is such that $C \backslash \{v\}$ is connected by black edges and so it is contained in a BC-clique of $P_{<v}$. Using the good ordering $v = v_1, \dots v_p$ for the nodes of P, we can incrementally add nodes of P starting from an empty graph, and computing the new solutions using the previous ones. A simple implementation of this approach requires storing and scanning the whole set of BC-cliques found so far, which can be costly in practice. Note that we need the latter ones for two reasons: to avoid duplication of BC-cliques and to retrieve BC-cliques to be extended by the current node $v = v_i$. We give an alternative way as follows.

Let R_v denote the set of BC-cliques in $P_{<v} \cup \{v\}$. Each BC-clique $K' \in R_{v_i}$, for some 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 characterize which K' are generated from K. In particular, we identify for which $i > j$, K will generate a clique in R_{v_j}. We observe that a BC-clique K generates

Algorithm 1. Finding the BC-cliques in the implicit 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 $P = G \cdot H$ for $T$
$\pi \leftarrow$ lexicographic order of the nodes in $V_P$ (good ordering);
for $v_i$ having no black edges going to $v_j$ with $j < i$ do visit($\{v_i\}$) ;

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

a new BC-clique in $R_{v_i} \backslash R_{v_{i-1}}$ if there is a black edge from v_i to a node in K. Thus it is easy to find all nodes v_i that can be used to extend K using the black edges: they are the nodes that are connected to K with black edges and succeed the largest node in K in the good ordering. We call a BC-clique that is generated from K by adding one such node v_i a *potential child* of K: specifically, it is found by 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.

We now focus on how to avoid generating the same BC-clique twice. Given a BC-clique K', and letting v_i be the largest node of K' in the good ordering, we define the *parent* of K' as the BC-clique in $P_{<v_i}$ obtained from $K' \backslash \{v_i\}$ by recursively adding the smallest node that can fully extend the current (non-maximal) BC-clique. Note that the parent of a BC-clique K is unique and is a BC-clique in R_{v_j} with $j < i$. We avoid generating duplicates as follows: when trying to generate K' from K, we accept K' only if K is the parent of K'; otherwise, K' is discarded. 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. Hence we call a potential child K' of K a *real child* of K if it satisfies both conditions: K' is generated by adding v_i and cannot be extended with a node less then v_i, and the parent of K' is K. This technique to remove duplicates is similar to the reverse search approach used in several enumeration algorithms, first introduced in [3].

Our algorithm is a refined variant of [11] that takes into account the distinction between black and white edges and recursively examines all the BC-cliques as explained before. The roots of the recursion trees are given by all the nodes that have no black backwards edges. (The pseudocode in shown in Algorithm 1.) It is clear from what we said before that this algorithm generates exactly once all the BC-cliques of $P_{<v_i}$. When a (partial) BC-clique is generated, we check whether it is maximal in P, and if this is the case we output it; this way our algorithm outputs all BC-cliques of P in an output-sensitive fashion.

Acknowledgments. Work partially supported by projects MIUR PRIN 2012C 4E3KT (all authors except LT, LV) and UNIPI PRA_2015_0058 (authors RG, LT).

References

1. Artymiuk, P., Poirrette, A., Grindley, H., Rice, D., Willett, P.: A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures. J Mol. Biol. **243**(2), 327–344 (1994)
2. Artymiuk, P., Spriggs, R., Willett, P.: Graph theoretic methods for the analysis of structural relationships in biological macromolecules. J. AM. Soc. Inf. Sci. Technol. **56**(5), 518–528 (2005)
3. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Appl. Math. **65**(1), 21–46 (1996)
4. Bonchev, D.: Chemical Graph Theory: Introduction and Fundamentals. CRC Press, Boca Raton (1991)
5. Brint, A., Willett, P.: Algorithms for the identification of three-dimensional maximal common substructures. J. Chem. Inf. Comput. Sci. **27**(4), 152–158 (1987)
6. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
7. Brun, L., Gaüzère, B., Fourey, S.: Relationships between graph edit distance and maximal common unlabeled subgraph. Technical report, HAL Id: hal-00714879, July 2012
8. Cao, Y., Charisi, A., Cheng, L., Jiang, T., Girke, T.: ChemmineR: a compound mining framework for R. Bioinformatics **24**(15), 1733–1734 (2008)
9. Cao, Y., Jiang, T., Girke, T.: A maximum common substructure-based algorithm for searching and predicting drug-like compounds. Bioinformatics **24**(13), i366–i374 (2008)
10. Carraghan, R., Pardalos, P.: An exact algorithm for the maximum clique problem. Oper. Res. Lett. **9**(6), 375–382 (1990)
11. Conte, A., Grossi, R., Marino, A., Versari, L.: Sublinear-space bounded-delay enumeration for massive network analytics: maximal cliques. In: ICALP (2016)
12. Conte, D., Foggia, P., Vento, M.: 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), 99–143 (2007)
13. Holder, L.: PDB-to-graph program (2015). https://github.com/mikeizbicki/datasets/tree/master/graph/pdb2graph. Accessed 04 May 2016
14. Huan, J., Wang, W., Prins, J., Yang, J.: Spin: mining maximal frequent subgraphs from graph databases. In: Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 581–586. ACM (2004)
15. Kann, V.: On the approximability of the maximum common subgraph problem. In: Finkel, A., Jantzen, M. (eds.) STACS 1992. LNCS, vol. 577, pp. 375–388. Springer, Heidelberg (1992). doi:10.1007/3-540-55210-3_198
16. Koch, I.: Enumerating all connected maximal common subgraphs in two graphs. Theor. Comput. Sci. **250**(1), 1–30 (2001)
17. Koch, I., Lengauer, T., Wanke, E.: An algorithm for finding maximal common subtopologies in a set of protein structures. J. Comput. Biol. **3**(2), 289–306 (1996)
18. Krissinel, E., Henrick, K.: Common subgraph isomorphism detection by backtracking search. Softw.: Pract. Experience **34**(6), 591–607 (2004)
19. Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. CALCOLO **9**(4), 341–352 (1973)

20. Mcgregor, J.: Backtrack search algorithm and the maximal common subgraph problem. Softw. Pract. Experience **12**, 23–34 (1982)
21. Raymond, J., Gardiner, E., Willett, P.: Rascal: calculation of graph similarity using maximum common edge subgraphs. Comput. J. **45**, 2002 (2002)
22. Sheridan, R., Kearsley, S.: Why do we need so many chemical similarity search methods? Drug Discov. Today **7**(17), 903–911 (2002)
23. Suters, W.H., Abu-Khzam, F.N., Zhang, Y., Symons, C.T., Samatova, N.F., Langston, M.A.: A new approach and faster exact methods for the maximum common subgraph problem. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 717–727. Springer, Heidelberg (2005). doi:10.1007/11533719_73
24. Ullmann, J.: An algorithm for subgraph isomorphism. J. ACM **23**(1), 31–42 (1976)
25. Van Berlo, R., Winterbach, W., De Groot, M., Bender, A., Verheijen, P., Reinders, M., de Ridder, D.: Efficient calculation of compound similarity based on maximum common subgraphs and its application to prediction of gene transcript levels. Int. J. Bioinform. Res. Appl. **9**(4), 407–432 (2013)
26. Versari, L.: Ricerca veloce di pattern comuni a due grafi. Master's thesis, University of Pisa, Pisa, Bachelor Thesis (in Italian), University of Pisa (2015)
27. Wang, T., Zhou, J.: EMCSS: a new method for maximal common substructure search. J. Chem. Inf. Comput. Sci. **37**(5), 828–834 (1997)
28. Welling, R.: A performance analysis on maximal common subgraph algorithms. In: 15th Twente Student Conference on IT, University of Twente, The Netherlands (2011)