Finding Maximal Common Subgraphs via Time-Space Efficient Reverse Search

Alessio Conte, Roberto P Grossi, Andrea Marino, Luca P Versari

Finding Maximal Common Subgraphs via Time-Space Efficient Reverse Search

Alessio Conte[1], Roberto Grossi[2], Andrea Marino[2], and Luca Versari[2]

[1] National Institute of Informatics, Tokyo, Japan, conte@nii.ac.jp
[2] Università di Pisa, Pisa, Italy, {grossi,marino,luca.versari}@di.unipi.it

Abstract. For any two given graphs, we study the problem of finding isomorphisms that correspond to inclusion-maximal common induced subgraphs that are *connected*. While common (induced or not) subgraphs can be easily listed using some well known reduction and state-of-the-art algorithms, they are not guaranteed to be connected. To meet the connectivity requirement, we propose an algorithm that revisits the paradigm of reverse search and guarantees polynomial time per solution (delay) and linear space, on top of showing good practical performance.

1 Introduction

The problem of finding common subgraphs, as studied in this paper, has been introduced and investigated in the practical setting of proteins [5,13,14], and can be employed to mine significant information in many domains, for example identifying compound similarity and structural relationships between biological molecules [9]. These patterns find motivation in the increasing amount of structured data arising from X-ray crystallography and nuclear magnetic resonance. For these reasons, the bioinformatics community has repeatedly expressed its interest in the detection of common subgraphs (see for instance [9,12,14,21]).

From a computational point of view, the problem has been studied as one of the application examples of an algorithmic framework [7] to efficiently enumerate maximal subgraphs satisfying a given property (e.g. being a clique, a cut, a cycle, a matching, etc.), also known as set systems [16]. In this paper we are interested to design efficient algorithms for the following scenario.

For any two given input graphs H and F, a subgraph S of H is *in common* with F if S is isomorphic to a subgraph of F: 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. The *maximal* common subgraph (MCS) problem further requires discovering all the MCS's of H and F. The MCS problem can be constrained to *connected* and *induced* subgraphs (MCCIS) [3,13,14], where the latter means that all the edges of H between nodes in the MCS are mapped to edges of F, and vice versa: considering induced subgraphs reduces the search space [3], and their connectivity further alleviates the explosion of the number of solutions [13,14], as otherwise each permutation of a maximal independent set corresponds to a different maximal isomorphism.

GRAPH H			GRAPH F			σ	q	Koch [13]		BC-ENUM		par.BC-ENUM	
n	m	Δ_H	n	m	Δ_F			TIME	#sol	TIME	#sol	TIME	#sol
200	235	5	200	234	7	5	12	28s	6691	0.2s	6691	0.04s	6691
100	122	7	100	119	5	4	22	11s	3654	0.6s	3654	0.1s	3654
2763	9488	14	2629	9059	12	12	68	2h	1998	2h	33874	2h	887293

Table 1. Comparison of polynomial space algorithms: running time of Koch's [13] algorithm vs our BC-ENUM and its parallel implementation. The first two rows are two pairs of random Erdos-Renyi graphs, and the last one a pair of graphs representing proteins from the Protein Data Bank (`1ald` and `1gox`) with a time limit of two hours. As for the notation, n, m, and σ are the number of nodes, edges, and node labels of the graphs, Δ_H and Δ_F their maximum degrees, and q the size of the largest found MCCIS.

MCCIS problem. Given any two graphs H and F, list all (isomorphisms corresponding to) maximal common connected induced subgraphs (MCCIS's) between H and F in polynomial time per solution and total polynomial space.

Actually, MCS and MCCIS will refer to isomorphisms corresponding to MCS or MCCIS. Note that solving the above MCCIS problem is computationally more demanding than listing just maximal common induced subgraphs (i.e. relaxing the connectivity constraint) as we will comment later in the state of the art.

Contributions. We present algorithm BC-ENUM, which lists the (isomorphisms corresponding to) MCCIS's with polynomial delay and using linear total space. Given any two graphs H and F, let Δ_H and Δ_F be their maximum degree, respectively. For each reported MCCIS, letting q be its number of nodes, we pay $O(q^4\Delta_H^2\Delta_F^2)$ time using $O(q)$ space: the time complexity gives the *delay*, which is the worst-case time between any two consecutively reported MCCIS's. Note that a strength of these bounds is that they are parameterized by the solution size q and independent of the sizes of H and F (just their maximum degree).

Table 1 reports the running time of a sequential and parallel implementation[3] of BC-ENUM in `C++`, compared to the state-of-the-art algorithm by Koch [13]. Experiments were executed on a 12-core machine with two Intel Xeon E5-2620 CPUs and 128 gigabytes of RAM, with a time limit of two hours, showing that on top of giving theoretical guarantees, BC-ENUM is also fast in practice.

Clarification on maximum vs maximal common subgraphs. As it is clear, *maximal* and *maximum* subgraphs are inherently different problems: listing *all* maximal ones can potentially find an exponential number of solutions, while finding the maximum connected ones corresponds to just the *single* largest one, and is in practice much faster (e.g. [19]). As pointed out in [5,13,14], however, a maximum common subgraph does *not* always contain all the relevant/large common structures, which motivates the MCCIS problem.

[3] Code available at `https://github.com/veluca93/parallel_enum/tree/bccliques` as part of a parallel enumeration framework.

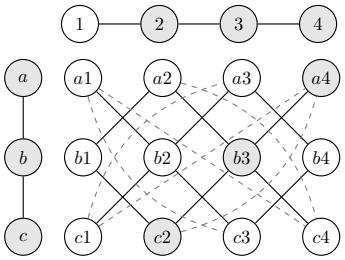

Fig. 1. An example of MCCIS ($\{a, b, c\}$ to $\{4, 3, 2\}$, in this order), with the corresponding BC-clique $\{a4, b3, c2\}$. White edges are represented as dashed lines.

Converting the MCCIS problem to a maximal clique problem. Clique-based methods are widely employed on the product graph G, which transforms common subgraphs of H and F into maximal cliques in G, as proved in [17].

As in [13], we define the *product graph* between H and F as follows. (i) any pair of nodes $(x, i) \in H \times F$ is a node of G iff they have the same label; (ii) there is a *black edge* between (x, i) and (y, j) iff $(x, y) \in E(H)$ and $(i, j) \in E(F)$; (iii) there is a *white edge* between (x, i) and (y, j) iff $x \neq y$, $i \neq j$, $(x, y) \notin E(H)$ and $(i, j) \notin E(F)$, where $E(\cdot)$ denotes the edge set.

The key property is that MCCIS's between H and F correspond to cliques in G spanned by black edges [13], which we will call BC-*cliques*. An example is shown in Fig. 1.

Role of the reverse search. Reverse search is a powerful enumeration technique, introduced by Avis and Fukuda [1], that applies to a wide range of problems (e.g. [5,18]). If we try to apply it to BC-cliques, a number of obstacles appear along the road and thus this paper proposes a novel, restructured, way to use reverse search on BC-cliques: Cao et al. [3] observe that materializing the product graph G can be memory-wise expensive. BC-ENUM does *not* materialize G, but navigates the huge solution space of the BC-cliques by navigating G implicitly using H and F, just requiring $O(q)$ additional space (e.g. for H and F in the last row of Table 1, G would contain millions of nodes whereas $q = 68$). This simultaneously improves memory usage and running time, as detailed in Section 5.

State of the art and related work. Common subgraphs problems have been studied for decades [3,10], with the great majority of the results dealing with *maximum* common subgraphs, rather than MCCIS's as we do. Previous work can be roughly classified into the following categories: backtracking methods [15], techniques based on special classes of graphs [11], clique-based methods [10,14,19], methods which are applications of a generic framework [4], and restricted to trees [8]. Among of them, Koch [13] considers MCCIS's and employs a modified version of the Bron-Kerbosch algorithm [2] to work on explicit product graphs: it is still the state of the art [22], greatly used in practice, even in the very last years (e.g. [23]). Further methods have relaxed the definition of MCCIS to improve the practical performance, at the price of loosing some solutions [5]. Unfortunately, the aforementioned algorithms, when applied to listing all the

MCCIS's, do not give any guaranteed polynomial bound on space or time per solution. Interestingly, a couple of other roads can be pursued successfully.

The framework presented in [4] uses the formulation of the reverse search on restricted problems and introduces new techniques for a class of set systems satisfying the connected hereditary property. The space is proportional to the number of solutions found, so it can be exponential, and thus space efficiency is one of the open problems posed there.

Along these lines, the framework presented in [7] provides new techniques for the class of set systems called commutable, and requires total polynomial space independently of the number of solutions found, thus answering to the question posed in [4].

We observe that BC-cliques can fit both frameworks, with polynomial time per solution. In this paper, we focus on the latter to provide polynomial bounds on the delay and space, while the implementation of the former in practice deserves further investigation in future work to evaluate the impact of the higher space usage. Compared to the bounds polynomial in the graph size from [7], which is a general theoretical framework for which BC-cliques are just an instance, BC-ENUM aims at specializing and parameterizing these bounds for the MCCIS problem, so they are polynomial in the max degree of the graphs (rather than in the size of the product graph), and at providing practical performance.

2 Using Reverse Search for Finding BC-cliques

As in [13], we reduce the problem of finding MCCIS's to finding BC-cliques in the product graph. Hence, in this section, we focus on the problem of listing BC-cliques in a graph G, whose edges are colored black or white, where a BC-clique is a maximal clique whose black edges connect all the nodes. To this aim, we employ reverse search, which can be successfully used when a suitable parent-child relationship between solutions is defined (see for instance [1,6,18]). Here we restructure the technique to deal with the more challenging BC-cliques. We keep the schema very simple for the sake of description, and hide the technical complexity in the definition of the parent-child relationship between solutions, which is the difficult part and it will be described in the following sections.

General Scheme. As is the case of reverse-search algorithms, our algorithm will implicitly define a rooted forest among all solutions, where some solutions are the *roots*, and from each solution we can find all its *children* in the forest. As we can identify all the roots and recursively visit the children of each solution, Algorithm 1 will not miss any solution.

In the following, let P be the *parent* of a solution S, denoted as $\textsc{p}(S)$, if S is a child of P in this rooted forest-like structure. Note that every solution has exactly one parent, except the roots who have none.

Lemma 1. *Algorithm 1 lists all maximal* BC*-cliques when the following conditions are all met.*

Algorithm 1: BC-ENUM: Enumerate all maximal BC-cliques of G

```
Function SPAWN(K)
    foreach S ∈ CHILDREN(K) do
        SPAWN(S);
    Output K;
foreach R ∈ ROOTS(G) do
    SPAWN(R)
```

1. *Each solution is either a* root*, or has exactly one* parent*.*
2. *The edges* $\text{P}(S) \to S$ *induce a forest* $\mathcal{Z}$ *whose sources are the* roots*.*
3. *The generic function* CHILDREN(P) *computes the set* $\{S : \text{P}(S) = P\}$*.*

The proof of Lemma 1 is straightforward as Algorithm 1 corresponds to a recursive traversal of the trees composing the forest $\mathcal{Z}$ induced by the parent-child relationship. We will design the latter so that the properties in Lemma 1 are satisfied. We remark that a tree traversal, rather than a graph traversal, does not require keeping track of visited nodes so far, and can be done without storing any information other than the current node and the previously visited one. We use this property to define an equivalent algorithm, which we call "stateless", that uses just $O(q)$ space and has the same complexity. We give further discussion in Section 5, and refer the reader to [6].

3 Canonical Representation and Operations

We consider G to have an arbitrary ordering of the nodes $\langle v_1 \dots v_n \rangle$, and we consider each node v_i to have label i. A node v_i is *smaller* than v_j if $i < j$. For convenience, we call G_B the subgraph of G induced by the black edges. We introduce the representation for BC-cliques in G at the base of our approach. For a BC-clique K, we call the smallest node in K the *head* of K, and define the notion of *black-edge distance* as follows.

Definition 1 (black-edge distance). *The* black-edge distance $\beta_K(v)$ *of a node* $v \in K$ *is the distance in the induced subgraph* $G_B[K]$ *between* v *and the head of* K*. If* $v \notin K$ *but* $K \cup \{v\}$ *is a* BC*-clique,* $\beta_K(v)$ *is similarly defined on* $G_B[K \cup \{v\}]$*.*

When K is clear from the context, we will omit the subscript and just write $\beta(v)$. (When v is the head, $\beta(v) = 0$.) Moreover, let us define the canonical order of K.

Definition 2 (canonical order). *Given a* BC*-clique* K*, the* canonical order *of* K *is* $\langle k_1, \dots, k_{|K|} \rangle$*, where elements of* K *are ordered in increasing lexicographical order of the pairs* $(\beta(k_i), k_i)$*.*

This order is a specialized version of the layer-based canonical order used in [7], which is the key to bound the running time to parameters of the two original input graphs H and F, rather than that of the larger G.

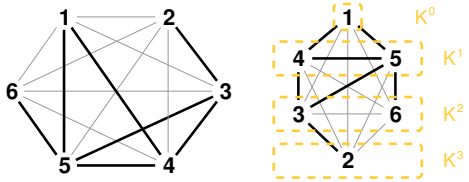

Fig. 2. A BC-clique K (left) and its canonical form $K^0 \dots K^3$ (right)

We will also refer to K^i as the set of nodes v with $\beta(v) = i$. This order essentially corresponds to the visiting order of a *breadth-first* search of $G_B[K]$, starting from the head k_1, where ties in the distance from k_1 are broken by taking the node with smallest label. As prefixes will be used extensively in our approach, we define $K_{<k_i}$ as the prefix $k_1, \dots, k_{i-1}$ of K. It can be easily seen from the above how any prefix of K is a (non maximal) BC-clique.

An example of a BC-clique K in canonical order is shown in Fig. 2 with K^i's ordered by node label, where the head is node 1 and, for instance, $\beta_K(3) = 2$. When levels are relevant in the context, we represent K as a sequence of sets, which corresponds to the K^i's in increasing order: the clique in the example would be represented as $K = \{1\}, \{4, 5\}, \{3, 6\}, \{2\}$. As an example of prefix we have $K_{<3} = \langle 1, 4, 5 \rangle$

We define the lexicographical order on BC-cliques using our canonical form. For any two pairs of integers (a, b) and (c, d), we write $(a, b) < (c, d)$ if the former pair is lexicographically smaller than the latter.

Definition 3 (lexicographical order). *Given two distinct* maximal BC*-cliques K and J, in their canonical orders $\langle k_1, \dots, k_{|K|} \rangle$ and $\langle j_1, \dots, j_{|J|} \rangle$, we say that K is lexicographically smaller than J, denoted as $K < J$, iff $(\beta_K(k_i), k_i) < (\beta_J(j_i), j_i)$, where i is the smallest index for which $(\beta_K(k_i), k_i) \neq (\beta_J(j_i), j_i)$.*

We also define a *forced* order of K with respect to $x \in K$, which is obtained by the same process as the canonical ordering, but computing the black-edge distances β with respect to x rather than the head of K.[4] A prefix of K *with respect to* x corresponds to a prefix of the forced order of K with respect to x. Clearly, this kind of prefix also corresponds to a BC-clique.

We now introduce the function COMPLETE(), which will be a key component as in [6,18]. Given a BC-clique K' which may or may not be maximal, COMPLETE(K') returns a *maximal* BC-clique K such that $K' \subseteq K$. This is achieved by recursively and greedily adding to K' the node $x \notin K'$ that minimizes $(\beta_{K'}(x), x)$, among all x for which $K' \cup \{x\}$ is a BC-clique. It is important to notice that the head of K' changes (as well as the values of $\beta_{K'}()$) whenever an element with label smaller than the current head is added. The COMPLETE operation is detailed in Algorithm 2. Finally, we remark that the properties of the COMPLETE function defined in [6], that make the reverse search algorithm work for cliques, are NP-hard to obtain in the context of BC-cliques [7].

[4] Thus the forced order of K with respect to its head is indeed the canonical order.

Algorithm 2: HEADS, CHILDREN functions that make Algorithm 1 working for BC-cliques

```
Function COMPLETE(K)
    while A ← {v ∈ N(K) : v has a black neighbour in K} ≠ ∅ do
        add argmin_{x∈A}{(β_k(x), x)} to K
    return K
Function PI(K)
    return argmax_{v∈K}{(β(v), v) : COMPLETE(K_<v) ≠ K}
Function P(K)
    v ← PI(K)
    return COMPLETE(K_<v)
Function ROOTS(G)
    return {COMPLETE(v) : v = min{COMPLETE({v})}}
Function CAND(K)
    return ⋃_{u∈K} N_B(u)
Function CHILDREN(K)
    foreach v ∈ CAND(K) do
        K'_v ← unique maximal BC-clique containing v in G_B[K ∩ N(v) ∪ {v}]
        foreach h ∈ K'_v do
            K''_v ← prefix of K'_v with respect to h, truncated at v
            D ← COMPLETE(K''_v)
            if K = P(D) ∧ h = min D ∧ v = PI(D) then yield D
```

4 Main Algorithm

In this section we describe the proposed algorithm BC-ENUM, using the definitions given in Section 3, to obtain the desired reverse search structure described in Section 2.

Firstly, using the function COMPLETE() we can easily identify the solutions which will be the roots of the forest $\mathcal{Z}$ induced by the parent-child relationship.

Definition 4 (root). *Let K be a maximal* BC*-clique and $h = \min K$ its* head*. Then, K is a root if and only if* $\text{COMPLETE}(\{h\}) = K$.

This definition implies that the number of roots is at most n, and each can be identified by performing $\text{COMPLETE}(v)$ on some node v. We now give definitions of P and CHILDREN as detailed in Algorithm 2.

The parent $\text{P}(K)$ of K is defined as the result of applying COMPLETE to the longest prefix $K_{<v}$ such that this operation does not yield K. We call the element v of K that immediately follows this prefix $K_{<v}$ the *parent index* of K, $\text{PI}(K)$. The definitions of P and root are consistent with Definition 4.

Lemma 2. $\text{P}(K) = \text{NULL}$ *if and only if* $K \in \text{ROOTS}$.

We give a definition of CHILDREN, whose correctness will be proven in Section 4.1. Given a BC-clique K, let CAND(K) be the set of nodes that do not belong to K, but are neighbors of some node of K in G_B. For each such node v, we compute the largest BC-clique K'_v that contains v and is contained in $K \cup \{v\}$ (which corresponds to the connected component containing v in $G_B[K \cap N(v)]$). Then, for each $h \in K'_v$, we consider the *forced* order $\langle k_1, \ldots, k_{|K'_v|} \rangle$ with respect to h (noting that $k_1 := h$), and compute K''_v as the prefix $k_1, \ldots, k_i$ of this order truncated at $k_i = v$.

Finally, we compute $D = \text{COMPLETE}(K''_v)$ and control if D satisfies the check at line 20, which is required to ensure that the parent of D is indeed K and that we did not generate D multiple times from K itself.

4.1 Correctness

In order to prove the correctness of Algorithm 1 using the routines defined in Algorithm 2, we prove that the conditions listed in Lemma 1 are met, recalling that the directed graph $\mathcal{Z}$ induced by the parent function P has the arcs from $\text{P}(K)$ to K for each solution K. By definition of P and by Lemma 2 we get Condition 1. Lemma 3 focuses on Condition 2, and Lemma 4 focuses on Condition 3.

Lemma 3. *The directed graph $\mathcal{Z}$ induced by* P *is a forest rooted in* ROOTS.

Lemma 4. *If $K = \text{P}(S)$, then $S \in \text{CHILDREN}(K)$.*

Proof. Consider an execution of CHILDREN(K), referring to its implementation in Algorithm 2, and let S be an arbitrary maximal BC-clique with $\text{P}(S) = K$. We need to prove that at some point in the execution Algorithm 2 will choose $v = \text{PI}(S)$ and $h = \min(S)$ in lines 15 and 17 respectively, and that this will give $D = S$ on line 19, which means D is yielded in line 20.

Consider the prefix $S_{<v}$ of S. As $K = \text{P}(S) = \text{COMPLETE}(S_{<v})$, clearly $S_{<v} \subset K$ and, since S has a parent, it is not a root and so $S_{<v} \neq \emptyset$. By definition of prefix, $S_{<v} \cup \{v\}$ is a BC-clique, so there must be a black edge between v and a node in $S_{<v}$, and thus to a node in K since $S_{<v} \subset K$, meaning that $v \in \text{CAND}$ and v is considered on line 15.

Consider now the execution of lines 16–20 when $v = \text{PI}(S)$. We have that $S_{<v} \cup \{v\}$ must be a subset of K'_v, the maximal BC-clique in $G_B[K \cap N(v) \cup \{v\}]$ containing v, since $S_{<v} \cup \{v\}$ is a BC-clique containing v and contained in $K \cup \{v\}$. Since $\min(S) \in S_{<v} \subseteq K'_v$, h will be chosen as $\min(S)$ in some iteration of line 17.

Finally, we need to prove that $S_{<v} \cup \{v\}$ is exactly K''_v, i.e., a prefix with respect to h of K'_v. If this were not the case, let d be the earliest element in K'_v (according to the forced ordering with respect to h) that is not in $S_{<v} \cup \{v\}$. Then $S_{<v} \cup \{v, d\}$ is still a BC-clique (as d must have a backwards black edge and all nodes before d are in $S_{<v}$); moreover, $(\beta_{K'_v}(d), d) < (\beta_{K'_v}(v), v)$. Thus d could be chosen by $\text{COMPLETE}(S_{<v} \cup \{v\})$, and since $\text{COMPLETE}(S_{<v} \cup \{v\}) = S$, this would mean that d is in S and occurs before v in its canonical ordering, implying $d \in S_{<v}$, which is a contradiction. □

As a result, we obtain the correctness of BC-ENUM.

Theorem 1. *Algorithm 1 implemented with the methods from Algorithm 2 finds all and only maximal* BC*-cliques exactly once.*

5 Complexity and Implicit Product Graph

In this section, we give the complexity of BC-ENUM, taking into account that G is not a generic graph with white and black edges, but an implicit product graph between H and F that we do not want to materialize, whose size and features depend on H and F.

Recall that each node of G corresponds to a mapping between two nodes of H and F. For any given $v \in V(G)$, let these nodes be respectively $v_H \in V(H)$ and $v_F \in V(F)$. Further recall that Δ_H and Δ_F are the maximum node degree in H and F, while Δ_B is the maximum degree in G_B. By construction of the product graph we have $\Delta_B \leq \Delta_H \cdot \Delta_F$. For brevity, we define Δ as $\Delta_H + \Delta_F$. These parameters are all significantly smaller than the size of G, which has $|V(H)| \cdot |V(F)|$ nodes, and $O(|V(H)|^2 \cdot |V(F)|^2)$ edges, either black or white.

Let X be a BC-clique in G. We denote as X_H and X_F respectively the set of nodes of H and F mapped in X. We keep a dictionary between the nodes of X and those of X_H and X_F, allowing us to retrieve v_H and v_F from v, or vice versa, in $O(1)$ time.[5]

Lemma 5. *Let X be a* BC*-clique in G and v a node in $V(G)$. Testing whether $X \cup \{v\}$ is a* BC*-clique takes $O(\min(|X|, \Delta))$ time and $O(|X|)$ space.*

Proof. As X is a BC-clique in G, in order to check that $X \cup \{v\}$ is a BC-clique in G we need to check that $\{v\}$ is connected to a node in X through a black edge, and to all the others through either white or black edges. This can trivially be done in $O(|X|)$ time by checking adjacency with the nodes of X one by one.

However, a faster solution is possible if we focus on the edges that are *not* in G: for a given node $x \in X$, corresponding to a mapping between $x_H \in V(H)$ and $x_F \in V(F)$, there is *no* edge in G between v and x if either $\{v_H, x_H\} \in E(H)$ and $\{v_F, x_F\} \notin E(F)$, or $\{v_H, x_H\} \notin E(H)$ and $\{v_F, x_F\} \in E(F)$. Otherwise, there is either a black or white edge between v and x.

To check the presence of missing edges between v and nodes of X we can simply iterate over all $x_H \in N_H(v_H) \cap X_H$, and check that each is mapped by X in a node $x_F \in N_F(v_F) \cap X_F$. Then, similarly, iterate over all $x_F \in N_F(v_F) \cap X_F$ and check that they are mapped in some $x_H \in N_H(v_H) \cap X_F$. This can be done in $O(|N_H(v_H)| + |N_F(v_F)|) = O(\Delta)$ time. If no missing edge exists then $X \cup \{v\}$ is a clique in G. As a byproduct, this process finds all black edges between v and X, thus we may check at the same time that there is at least one, and thus that $X \cup \{v\}$ is a BC-clique. □

[5] This data structure will be built at the beginning of a COMPLETE call. As building it takes $O(|X|)$ time and space, it will not affect the final complexity.

Lemma 6. *For any* BC*-clique X in G, computing $\beta_X(v)$ for all $v \in X$ takes $O(|X| \cdot \min(|X|, \Delta_H, \Delta_F))$ time and $O(|X|)$ space.*

Proof. The values of $\beta_X(v)$ corresponds to their distance from the head x of X in $G_B[X]$. This can be done via a BFS of $G_B[X]$ rooted at x. As $G_B[X]$ has $|X|$ nodes, the trivial bound for this traversal is $|X|^2$. Once again, we can exploit the fact that G is the product graph of H and F: indeed, each node v of X corresponds to a mapping of a node v_H of H into *one* node v_F of F. For this reason, while v can have up to Δ_B neighbors in G_B, v_H may have at most $|N_H(v_H)|$ neighbors in X_H.

We can thus iterate on the neighborhood of v in $O(\min(\Delta_H, \Delta_F))$ time by iterating on the neighbors of either v_H in H or v_F in F and then retrieve the corresponding nodes in X. In total, we process $|X|$ nodes, each in $O(\min(\Delta_H, \Delta_F))$ time, or in $O(|X|)$ time using the trivial version of the BFS. The cost follows. □

Lemma 7. COMPLETE(X) *takes $O(q(q + \Delta_B)\Delta) = O(q^2\Delta + q\Delta_B\Delta)$ time and $O(q)$ space.*

Proof. In order to perform COMPLETE(X), we iterate over all nodes that can be added to X, adding the lexicographically smallest, with respect to X and its head x, first. For each node v in X (including those that are added during the procedure), we keep an iterator which will scan in increasing order its black neighbors. Clearly, each node must be considered after the smallest ones, and once it is considered it is either added to X or discarded, thus it does not need to be considered as a candidate anymore.

Given a node $c \notin X$, that has a black neighbor in X, we can see that $\beta_X(c) = \beta_X(v) + 1$, where v is the black neighbor of c in X that minimizes this value. Hence, to select the lexicographically smallest node, we must first consider the black neighbors of the nodes v that minimize $\beta_X(v)$. We thus order the nodes in a priority queue by value of $\beta_X(v)$, breaking ties by the value of the smallest black neighbor yet to consider, so that the first node in the priority queue is the smallest candidate to consider for addition to X.

As X will contain $|X| = O(q)$ nodes, and we will iterate on the $O(\Delta_B)$ black neighbors of each node exactly once, the total cost of this iteration is $O(q\Delta_B)$ time, and will yield up to $q\Delta_B$ nodes. Since by Lemma 5 testing a candidate takes $O(\min(q, \Delta))$ time, the total cost is $O(q\Delta_B \min(q, \Delta))$.

Furthermore, we need to account for the cost of changing *heads*: after we add a node x to X, this becomes the new head of X if its label is smaller than that of the previous head. In this case, we need to update both the values of $\beta_X(v)$, and the priority queue of candidate nodes. By Lemma 6, this can be done in $O(q \min(q, \Delta_H, \Delta_F))$ time. We pay this cost at most q times as we add up to q nodes, for a cost of $O(q^2 \min(q, \Delta_H, \Delta_F))$ which is upper bounded by $O(q^2\Delta)$.

The total cost is thus $O(q\Delta_B \min(q, \Delta) + q^2\Delta) = O(q^2(\Delta_B + \Delta))$ time. □

Lemma 8. CHILDREN(K) *takes $O(q^4\Delta_B(\Delta_B + \Delta)) = O(q^4\Delta_H^2\Delta_F^2)$ time and $O(q)$ space.*

Proof. The cost of CHILDREN(K) is bounded by the cost of lines 19 and 20, times the number of nodes CAND(K), times the number of nodes in K'_v.

Nodes in CAND(K) are at most $|K|\Delta_B = O(q\Delta_B)$ (line 13 of Algorithm 2) and K'_v size is bounded by $O(q)$. In the following, we prove that the time cost of line 20 of Algorithm 2 is $O(q(q+\Delta_B)\Delta)$. Let $\langle d_1, \ldots, d_{|D|}\rangle$ be the canonical ordering of D. By definition, $d_i = \text{PI}(D)$ is the latest element in the canonical order of D such that COMPLETE($D_{<d_i}$) $\neq D$. By the proof of Lemma 3, we have that COMPLETE($D_{<d_i}$) $\leq D$ and COMPLETE($D_{<d_j}$) $= D$ for any $j > i$. To check that v is indeed the parent index of D, we thus simply need to check that COMPLETE($D_{<d_i}$) $\neq D$ and COMPLETE($D_{<d_{i+1}}$) $= D$. Furthermore, if this is the case, COMPLETE($D_{<d_i}$) also gives us the parent P(D) of D. Checking that h is the node of smallest label in D does not affect the cost, thus the total cost is that of calling the COMPLETE() function twice. As Δ_B and Δ are bounded by $\Delta_H \cdot \Delta_F$, the statement follows. □

Looking at Algorithm 1, we can see that the complexity of BC-ENUM is bounded by the cost of the function CHILDREN(K). Furthermore, as shown in Lemmas 5, 6, and 7, the space required is always $O(q)$. By turning the recursion into a stateless iteration (see [6]), no more space is needed as we do not need to store the recursion stack. We also address the *delay* of the algorithm, that is, the maximum elapsed time between two consecutive outputs, by applying the *alternative output* technique in [20]: for each recursive call on K in the recursion tree of Algorithm 1, we output solution K at the *beginning* of the call if its depth is even, and at the *end* if it is odd. In this way, the delay is equal to the cost per solution. We can thus state the main result.

Theorem 2. *Given two graphs H and F,* BC-ENUM *lists all their (isomorphisms corresponding to)* MCCIS*'s in $O(q^4\Delta_H^2\Delta_F^2)$ delay and $O(q)$ space.*

Acknowledgements. Alessio Conte is supported by JST CREST, grant number JPMJCR1401, Japan, and Roberto Grossi, Andrea Marino and Luca Versari are supported by MIUR, Italy.

References

1. D Avis and K Fukuda. Reverse search for enumeration. *Discrete Appl Math*, 65(1):21–46, 1996.
2. Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.
3. Y Cao, T Jiang, and T Girke. A maximum common substructure-based algorithm for searching and predicting drug-like compounds. *Bioinformatics*, 24(13):i366–i374, 2008.
4. S Cohen, B Kimelfeld, and Y Sagiv. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *J Comput Syst Sci*, 74(7):1147–1159, 2008.
5. A Conte, R Grossi, A Marino, L Tattini, and L Versari. A fast algorithm for large common connected induced subgraphs. In *AlCoB*, pages 62–74, 2017.

6. A Conte, R Grossi, A Marino, and L Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In *ICALP*, pages 148:1–148:15, 2016.
7. A. Conte, R. Grossi, A. Marino, and L. Versari. Listing Maximal Subgraphs in Strongly Accessible Set Systems. *ArXiv e-prints*, March 2018. `arXiv:1803.03659`.
8. A Droschinsky, B Heinemann, N Kriege, and P Mutzel. Enumeration of maximum common subtree isomorphisms with polynomial-delay. In *ISAAC*, pages 81–93. Springer, 2014.
9. HC Ehrlich and M Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(1):68–79, 2011.
10. EJ Gardiner, PJ Artymiuk, and P Willett. Clique-detection algorithms for matching 3-dimensional molecular structures. *J Mol Graph Model*, 15:245 – 253, 1997.
11. A Gupta and N Nishimura. Finding largest subtrees and smallest supertrees. *Algorithmica*, 21(2):183–210, 1998.
12. X Huang, J Lai, and S Jennings. Maximum common subgraph: some upper bound and lower bound results. *BMC Bioinformatics*, 7(Suppl 4):S6, 2006.
13. I Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor Comp Sci*, 250(1):1–30, 2001.
14. I Koch, T Lengauer, and E Wanke. An algorithm for finding maximal common subtopologies in a set of protein structures. *J Comp Bio*, 3(2):289–306, 1996.
15. EB Krissinel and K Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Practice and Experience*, 34(6):591–607, 2004.
16. EL Lawler, JK Lenstra, and AHG Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM J Comput*, 9(3):558–565, 1980.
17. G Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO*, 9(4):341–352, 1973.
18. K Makino and T Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, pages 260–272. Springer, 2004.
19. WH Suters, FN Abu-Khzam, Y Zhang, CT Symons, NF Samatova, and MA Langston. A new approach and faster exact methods for the maximum common subgraph problem. In *Computing and combinatorics*, pages 717–727. 2005.
20. Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms, 2003. NII Technical Report NII-2003-004E, Tokyo, Japan.
21. RJP Van Berlo, W Winterbach, MJL De Groot, A Bender, PJT Verheijen, MJT Reinders, and 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):407–432, 2013.
22. R Welling. A performance analysis on maximal common subgraph algorithms. In *15th Twente Student Conference on IT, The Netherlands*, Un. of Twente, 2011.
23. Y Yuan, G Wang, L Chen, and H Wang. Graph similarity search on large uncertain graph databases. *The VLDB Journal*, 24(2):271–296, 2015.