New Polynomial Delay Bounds for Maximal Subgraph Enumeration by Proximity Search

Alessio Conte, Takeaki Uno

National Institute of Informatics, Tokyo, Japan
{conte,uno}@nii.ac.jp

Abstract

In this paper we propose polynomial delay algorithms for several maximal subgraph listing problems, by means of a seemingly novel technique which we call *proximity search*. Our result involves modeling the space of solutions as an implicit directed graph called "solution graph", a method common to other enumeration paradigms such as reverse search. Such methods, however, can become inefficient due to this graph having vertices with high (potentially exponential) degree. The novelty of our algorithm consists in providing a technique for generating better solution graphs, reducing the out-degree of its vertices with respect to existing approaches, and proving that it remains strongly connected. Applying this technique, we obtain polynomial delay listing algorithms for several problems for which output-sensitive results were, to the best of our knowledge, not known. These include Maximal Bipartite Subgraphs, Maximal k-Degenerate Subgraphs (for bounded k), Maximal Induced Chordal Subgraphs, and Maximal Induced Trees. We present these algorithms, and give insight on how this general technique can be applied to other problems.

1 Introduction

Given a set of elements (such as the vertices or edges of a graph) sometimes called the universe, and a property (such as being a clique, or a tree), a listing problem asks to return all subsets of the universe which satisfy the given property.

Listing structures, within graphs or other types of data, is a basic problem in computer science, and it is at the core of data analysis. While many problems can be solved by optimization approaches, e.g., by finding the shortest path, or the largest clique, other require finding several solutions to the input problem: in community detection, for example, finding just one "best" community only gives us local information regarding some part of the data, so we may want to find several communities in order to make sense of the input. Furthermore, in many real world scenarios there may not be a clear objective function to optimize to obtain the best solution: We may define an algorithm which optimizes some desired property, but the optimal solution found may be lacking others or simply not be practical. We thus may want to quickly list several solution which may be suitable, according to some metrics, then analyze them a posteriori to find the desired one.

In these scenarios, listing only the solutions that are *maximal* under inclusion is a common sense requirement where it can be applied,[1] as maximal solutions subsume the information contained in all others, and may be exponentially fewer: For example, a graph may have 2^n cliques, but only

[1] In other problems, we may want *minimal* solutions instead, although this is usually equivalent since it corresponds to requiring the complement of a solution to be maximal.

$3^{n/3}$ maximal ones [22]. For brevity, we call *maximal listing problem* a listing problem where only the maximal solutions should be output.

From a theoretical point of view, listing provides many challenging problems, especially when maximality is required. When dealing with listing algorithms, we are often interested in their complexity with respect to both n, the size input, and N, the size of the output. Algorithms whose complexity can be bounded by a polynomial of these two factors are called *output-sensitive*, or equivalently *output polynomial* or *polynomial total time* [16]. Interestingly, the hardness of listing problems does not seem to be correlated with that of optimization: there are several NP-Hard optimization problems whose corresponding maximal listing problem allows an output-sensitive solution (see, e.g., [31, 1]) while, for other problems, finding an output-sensitive algorithm for listing maximal solutions would imply P=NP, even though one maximal (or maximum) solution can be identified in polynomial time [20].

A long standing question in the area is to find a characterization of which listing problems allow output-sensitive solutions and which do not. Furthermore, within output-sensitive algorithms stricter complexity classes exist, such as *incremental polynomial time*, where the time to output the i-th solution is polynomial in n and i, and *polynomial delay*, where the time elapsed between two consecutive outputs is polynomial in n.

In this work we add a few points to the earlier class, by showing that there exist polynomial delay algorithms for some subgraph listing problems. More formally, we prove Theorem 1 (where max. is short for maximal and sg. is short for subgraphs)

Theorem 1. *The following problems allow polynomial delay listing algorithms by proximity search:*

max. induced bipartite sg.	*max. connected induced bipartite sg.*	*max. (edge) bipartite sg.*
max. induced chordal sg.	*max. connected induced chordal sg.*	*max. obstacle-free convex hulls*
max. induced k-degenerate sg.	*max. (edge) k-degenerate sg.*	*max. connected directed acyclic sg.*

To the best of our knowledge, no output-sensitive result was previously known for these problems. For completeness, we consider both the *induced* and *edge* subgraph cases, where subgraphs are defined by sets of vertices or edges, respectively, as well as the *connected* case where subgraphs are required to be connected.

Furthermore, we abstract a general technique which can be used to obtain similar results on other problems.

We do so by defining a graph whose vertices are the maximal solutions to the listing problem, and with directed edges between pairs of solutions, which we call *solution graph*. The listing problem is solved by traversing the solution graph, and proving that all solutions are found this way. The concept solution graph is common to existing approaches, and general techniques already exist for building them, e.g., [4]. However, the solution graph build with known approaches such as [4] may have too many edges, resulting in a traversal with exponential delay.

The key concept given in this paper is a technique to build a solution graph with fewer edges, while at the same time proving that all solutions are found by its traversal. An interesting property of this approach is that the resulting algorithms are remarkably simple to implement, while the complexity lies in proving their correctness.

We call this technique *proximity search* since at its core lies a problem-specific notion of proximity. This notion acts as a sort of compass on the solution graph built by our algorithm, as given any two solutions S and S^*, we will show that we always traverse an edge from S to another solution S' that has higher proximity to S^*; as S^* has the highest proximity to itself, this implies that a traversal of the solution graph from any solution finds all others. While others, such as [4], already used the principle of reachability in the solution graph, the novelty of this approach lies in the loose

and problem-specific concepts of proximity and neighboring function, which allow us to overcome the exponential burden imposed by the input-restricted problem.

1.1 Related work

The listing problems considered in this paper model solutions as sets of elements (e.g., sets of vertices or edges of a graph), and consist in listing sets of elements with some required property, e.g., inducing a bipartite subgraph, or a tree. We observe that the output is a family of sets, we can associate properties with the corresponding set systems: for example, a property is hereditary when each subset of a solution is a solution, which corresponds to the well known independence systems [20].

In this context, a simple yet powerful technique is recursively partitioning the search space into all solutions containing a certain element, and all that do not. This technique, usually called binary partition or simply backtracking, proves efficient when listing all solutions [26], and can be used to design algorithms that are fast in practice,[2] or that can bound the number of solutions in the worst-case [12]. On the other hand, this strategy rarely gives output-sensitive algorithms when dealing with maximal solutions, as we may spend time to explore a solution subspace which contains many solutions but no maximal one.

To obtain output-sensitive algorithms for maximal solutions, many algorithms rely on the following idea: given a maximal solution S, and some element $x \notin S$, the hardness of listing solutions *maximal within* $S \cup \{x\}$ is linked to the hardness of listing them in a general instance. The earliest mentions of the idea can be found in the influential work by Lawler et al. [20], that aims at generalizing ideas from Paull et al. [24] and Tsukiyama et al. [31], and has been later formally defined as *input-restricted problem* by Cohen et al. [4].

The intuition is that the solutions obtained this way, using a maximal solution S and an element not in S, can be used to generate new maximal solutions of the original problem. We can thus traverse an implicit directed graph, which we will call *solution graph*, where the vertices are the maximal solutions and the out-neighbors are obtained by means of the input-restricted problem.

In particular, [20] showed how solving this problem could yield an output-sensitive and polynomial space listing algorithm for properties corresponding to *independence systems*, assuming the input-restricted problem has a bounded number of solutions. [4] showed that the strategy could be extended to the more challenging *connected hereditary* graph properties (i.e., where *connected* subsets of solutions are solutions) using exponential space, and recently, [5] showed that the same result can be obtained in polynomial space for *commutable* set systems (which include connected-hereditary properties).

The literature contains many more results concerning the enumeration of maximal/minimal solutions, e.g., [1, 28, 13, 15], and in particular regarding challenging problems such as the well known minimal hypergraph transversals/dominating sets problem [17, 14, 10]. However, to the best of our knowledge, there is no general technique which can be used to produce polynomial delay algorithms for classes of problems whose associated input-restricted problem is not solvable in polynomial time.

1.2 Overview

The main contribution of the paper is presenting proximity search, a general technique which can be used to solve several enumeration problems in polynomial delay, even some cases of maximal subgraph listing where the input-restricted problem has exponentially many solutions.

[2] E.g., the Bron-Kerbosh [30] algorithm tends to be faster than output-sensitive ones [6] for maximal cliques.

By using this technique we show polynomial delay algorithms for several maximal listing problems such as maximal bipartite subgraphs and the others mentioned in Theorem 1. Other than providing efficient algorithms, we remark that technique may help gain further insight on which classes of problem allow output-sensitive listing algorithms and which do not.

The paper is organized as follows: First, we introduce some basic concepts and notation in Section 2. Then we proceed bottom-up, focusing on the case of Maximal Bipartite Subgraphs: in Section 3.1 we give a polynomial algorithm for listing Maximal *Connected* Induced Bipartite Subgraphs, and later adapt it to the non-connected and edge-subgraph cases. We then explain in Section 4 how to abstract the structure of the algorithm to solve other enumeration problems. We formally define a class of problems, called *proximity searchable*, which allow a polynomial delay algorithm using the proposed techniques.

The remaining section of the paper, i.e., Sections 5-9, are consider other maximal listing problems for which, to the best of our knowledge, output-sensitive algorithms are now known. We show these problems to be proximity searchable, and thus obtain polynomial delay algorithms for them. Finally, we give our concluding remarks in Section 10.

2 Preliminaries

In this paper we consider enumeration problems on an undirected graph $G = (V(G), E(G))$, whose vertex set is denoted as $V(G)$ and edge set as $E(G)$, or simply V and E when G is clear from the context. The neighborhood of a vertex v is denoted as $N(v)$. For brevity we refer to $|V(G)|$ as n, to $|E(G)|$ as m, and to the maximum degree of a vertex in G as Δ. Furthermore, we assume the vertices to be labelled arbitrarily in increasing order $v_1, \ldots, v_n$, and say that v_i is (lexicographically) smaller than v_j, i.e., $i < j$, if v_i has a smaller label. Furthermore, given a vertex ordering, we say that a neighbor of v_i is a *forward* neighbor if it comes later than v_i in the ordering, and a *backward* neighbor otherwise. For a set of vertices $A \subseteq V(G)$, $E[A]$ denotes the edges of G whose endpoints are both in A, and $G[A]$ the graph $(A, E[A])$, i.e., the subgraph induced in G by A. Similarly, for a set of edges B, $V[B]$ denotes the vertices incident to an edge in B and $G[B] = (V[B], B)$. As common in the literature, we call *induced subgraphs* those of the earlier kind, defined by a set of vertices, and *edge subgraphs* (or simply subgraphs) those of the latter, defined by a set of edges.

When dealing with subgraphs defined by a set of vertices (resp. edges) A, we will sometimes use A to refer to both the vertex set (resp. edge set) and the subgraph $G[A]$ it induces, when this causes no ambiguity. For further notation, we refer to the standard terminology in [9].

For a set of vertices $A \subseteq V(G)$ which corresponds to a (non necessarily maximal) solution of the problem at hand, we define a simple "maximalization" function, named $\textsc{comp}(A)$: this function greedily adds to A the vertex v of minimum label such that the result is still a solution, until A is maximal. It can be proved that this greedy process always returns a maximal solution if the property at hand is hereditary (an independence system) or connected hereditary, and that it runs in polynomial time assuming we can recognize solutions in polynomial time [4]. Note that this is true for all graph properties considered in this paper.

3 Maximal Bipartite Subgraphs

A graph is bipartite if it allows a bipartition of its vertices V_0, V_1 such that $V_0 \cap V_1 = \emptyset$, $V_0 \cup V_1 = V(G)$, and both $G[V_0]$ and $G[V_1]$ are empty graphs. Equivalently, this corresponds to graph with no *odd* cycle. Maximal bipartite subgraphs have also been studied as *minimal odd cycle transversals* [19], as one is the complement of the other.

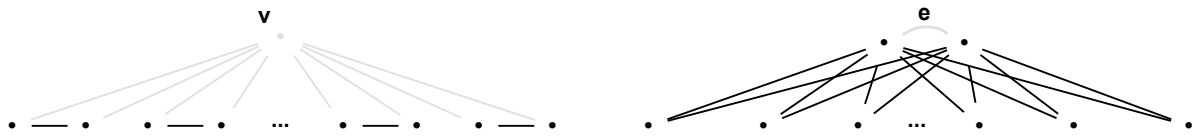

Figure 1: Instances of input-restricted problem for maximal bipartite subgraphs. On the left: the black dots define a maximal bipartite induced subgraph; adding the vertex v creates a graph with exponentially many maximal bipartite induced subgraphs, as we can obtain one by removing a vertex from each connected pair in the bottom in any combination. On the right, a similar case for the highlighted edge subgraph and the edge e.

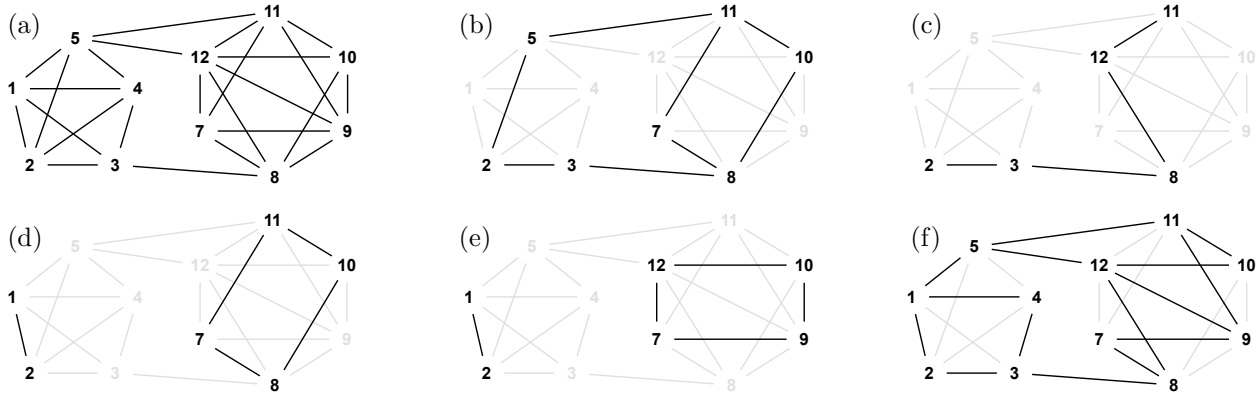

Figure 2: **a**: a graph. **b**,**c**: two maximal connected induced bipartite subgraphs of (a). **d**,**e**: two maximal induced bipartite subgraphs of (a). **f**: a maximal edge bipartite subgraph of (a).

The problem of listing *all* bipartite (and induced bipartite) subgraphs has been efficiently solved in [34]. However, to the best of our knowledge, neither the techniques in [34] or other known ones extend to efficiently listing *maximal* bipartite subgraphs.

Finally, we remark that a subgraph of a bipartite graph is itself bipartite, meaning the property is hereditary, and the connected version connected hereditary.

For completeness, Figure 1 shows two instances of input-restricted problem with exponentially many solutions, meaning that applying the techniques from [20, 4, 5] would not result in polynomial cost per solution. Indeed to obtain polynomial cost per solution we would need to solve the input-restricted problem in polynomial time, which is clearly not possible. The best we could hope for is solving it with polynomial delay, which would yield an incremental polynomial time algorithm. Figure 2 shows examples of the three types of bipartite subgraphs we will consider.

We denote an induced bipartite *subgraph* as a pair of vertex sets B_0, B_1, with $B_0 \cap B_1 = \emptyset$ and $B_0 \cup B_1 \subseteq V(G)$, such that $G[B_0]$ and $G[B_1]$ are empty graphs.

To remove ambiguity and duplication, B_0 is always assumed to be the side of the bipartition containing the vertex of smallest label among those in the subgraph. In case $G[B_0 \cup B_1]$ has multiple connected components, this applies to all components. This way, any bipartite subgraph (connected or not) always has a *unique* representation B_0, B_1. We will sometimes use simply B to refer to the subgraph B_0, B_1.

We define the intersection between two bipartite subgraphs B and B' as the set of all shared vertices, i.e.: $B \cap B' = (B_0 \cup B_1) \cap (B'_0 \cup B'_1)$.

Note that, when performing $\textsc{comp}(B)$, if B is not connected this may move some vertices from B_0 to B_1 and vice versa due to different components becoming connected; even when B is

connected, if a vertex with smaller label than all others in B is added to B_1, then B_0 and B_1 are immediately swapped to preserve the invariant of the smallest vertex being in B_0.

3.1 Listing maximal connected induced bipartite subgraphs

In this section we consider the case of induced connected subgraphs. We will later briefly show how this structure can be adapted to cover the non-connected and non-induced cases with small changes. Let $B = B_0, B_1$ be a maximal induced bipartite subgraph, and v a vertex not in it, i.e., in $V(G) \setminus B$.

We define a key component of the algorithm, i.e., an operation to obtain neighboring solutions:

Definition 2 (neighboring function for maximal induced bipartite subgraphs)**.**

$$\textsc{neigh}(B_i, B_{1-i}, v) = B'_0, B'_1 = \textsc{comp}(\{v\} \cup (B_i \setminus N(u)) \cup B_{1-i})$$

Where i is an index in $\{0, 1\}$, meaning that the formula can be applied as either $\textsc{neigh}(B_0, B_1, v)$ or $\textsc{neigh}(B_1, B_0, v)$.

Note that the set $\{u\} \cup (B_i \setminus N(u)) \cup B_{1-i}$ must still induce a bipartite subgraph, as we are removing the neighbors of u from B_i: all the edges in the remaining subgraph must have one extreme in $(\{u\} \cup (B_i \setminus N(u)))$ and the other in B_{1-i}. We then apply $\textsc{comp}()$ to obtain a maximal solution B'_0, B'_1, different from B_0, B_1 as it contains u.

However, the resulting subgraph may not be connected. To enforce this, we need to remove the vertices that become disconnected from u as well. More formally, we have

Definition 3 (neighboring function for maximal connected induced bipartite subgraphs)**.**

$$\textsc{neigh}(B_i, B_{1-i}, v) = B'_0, B'_1 = \textsc{comp}(\mathtt{cc}_u(\{u\} \cup (B_i \setminus N(u)) \cup B_{1-i}))$$

Where $\mathtt{cc}_u(H)$ denotes the connected component of the graph or subgraph $G[H]$ containing u.

As $\mathtt{cc}_u(\{u\} \cup (B_i \setminus N(u)) \cup B_{1-i})$ is a connected subgraph of a bipartite graph it will be a connected bipartite subgraph. Furthermore, as the solutions to the problem are connected subgraphs, $\textsc{comp}()$ will only add vertices which preserve the connectivity constraint.

Given a solution B_0, B_1, and considering all vertices $u \in V(G) \setminus (B_0 \cup B_1)$, we can thus obtain new solutions, each surely different from B_0, B_1 in that it contains u: these will correspond to the out-neighbors of B_0, B_1 in the solution graph.

It is evident that the out-degree of vertices in this solution graph is polynomial, since we generate at most $2|V(G)|$ neighbors from each solution. On the other hand, solving the input-restricted problem for $B_0 \cup B_1 \cup \{u\}$ would generate up to exponentially many neighbors, as remarked by the example in Figure 1 (left).

Our proposed algorithm essentially performs a depth-first search on this graph, without ever generating the full graph, but keeping track of the vertices already visited (i.e., the solutions found so far) to avoid duplication. We thus need to prove that a traversal of this sparser graph still yields all solutions, and we do so by proving that the graph is strongly connected, which is a sufficient condition.

The pseudo code of the resulting algorithm is given in Algorithm 1

Algorithm 1: Enumerate all maximal induced bipartite subgraphs.

input : A graph $G = (V(G), E(G))$
output: All maximal induced bipartite subgraphs of G
global : Set $\mathcal{S}$ of solutions found, initially empty

```
1 Let v be an arbitrary vertex in V(G)
2 ENUM(COMP({v}))
3 Function ENUM(B_0, B_1)
4     Add (B_0, B_1) to S
5     output (B_0, B_1)
6     foreach v ∈ V(G) \ (B_0 ∪ B_1) do
7         if S does not contain NEIGH(B_0, B_1, v) then ENUM(NEIGH(B_0, B_1, v))
8         if S does not contain NEIGH(B_1, B_0, v) then ENUM(NEIGH(B_1, B_0, v))
```

3.2 Correctness

The proof will proceed as follows: first, we define a binary function on two solutions named "proximity". Intuitively, this can be seen as a sort of similarity measure that is maximized when the two solutions are equal. Then, we prove that for any pair of solutions B and B^* there exists a vertex $v \in V(G)$ such that either $\text{NEIGH}(B_0, B_1, v)$ or $\text{NEIGH}(B_1, B_0, v)$ is *closer* to B^* in terms of proximity. It follows that we may iteratively apply NEIGH operations to finally reach exactly B^*. In turn, this means that the implicit graph whose vertices are the solutions and whose directed edges are defined by the NEIGH operations is strongly connected, and that the proposed algorithm corresponds to performing a visit on this graph.

In the following, we use an uniform notation for simplicity: we consider two solutions $B = B_i, B_{1-i}$ and $B^* = B^*_j, B^*_{1-j}$, where i and j are two indices in with value in $\{0, 1\}$.

For a solution B, we define its *canonical* order as follows:

Definition 4 (canonical order for connected induced bipartite subgraphs)**.**

The canonical *order of a* connected induced bipartite subgraph B *is the sequence* $b_1, \ldots, b_{|B|}$ *of its vertices, where* b_1 *is the vertex of smallest label in* B*, and* b_i *the vertex of smallest label in* $B \setminus \{b_1, \ldots, b_{i-1}\}$ *such that* $\{b_1, \ldots, b_i\}$ *is a* connected *induced bipartite subgraph.*

An important property of this order is that any of its prefixes $\{b_1, \ldots, b_i\}, i \leq |B|$ still correspond to a connected induced bipartite subgraph.

For example, looking at Figure 2, the canonical order of the subgraph (b) would be $\{2, 3, 5, 8, 7, 10, 11\}$ and that of (c) $\{2, 3, 8, 12, 11\}$.

Using the canonical order, we then define the "proximity" between two solutions B and B^*:

Definition 5 (proximity)**.**

Given two solutions S *and* S^**, let* $\{s^*_1, \ldots s^*_{|S^*|}\}$ *be the canonical order of* S^**: the proximity* $S \tilde{\cap} S^*$ *between* S *and* S^* *is the longest* prefix $\{s^*_1, \ldots, s^*_i\}$ *of the canonical order of* S^* *whose elements are all contained in* S*.*

It can be seen that this proximity is maximized if $S = S^*$, where we have $S \tilde{\cap} S^* = S^* = S$, and minimized when the two solutions share no vertex (more in general, when $s^*_1 \notin S$, even if other elements of S^* are in S). However, it should be noted that the operation is *not* symmetrical, i.e., we may have $S \tilde{\cap} S^* \neq S^* \tilde{\cap} S$. For example, let S be the subgraph shown in Figure 2 (b) and S^* that

shown in (c). Considering the canonical orders mentioned above, we can see that $S\tilde{\cap}S^* = \{2, 3, 8\}$, and $S^*\tilde{\cap}S = \{2, 3\}$.

In the following, we will assume B and B^* to be two distinct maximal connected induced bipartite subgraphs, $\{b_1^*, \dots b_{|B^*|}^*\}$ the canonical order of B^*, and denote with $\dot{v}$ the earliest vertex in this order that is not part of the proximity $B\tilde{\cap}B^*$ (i.e., $\dot{v} = b_{i+1}^*$, with $B\tilde{\cap}B^* = \{b_1^*, \dots, b_i^*\}$). Note that since B and B^* are distinct and inclusion maximal (i.e., none may contain the other), $\dot{v}$ is always defined.

The correctness of the approach will be proved via a notion of monotonicity on the "proximity". In other words, we will prove that for any pair of solutions B_0, B_1 and B_0^*, B_1^*, there exists a vertex $v \in V(G) \setminus (B_0 \cup B_1)$ such that for either $B_0', B_1' \leftarrow \textsc{neigh}(B_0, B_1, v)$ or $B_0', B_1' \leftarrow \textsc{neigh}(B_1, B_0, v)$ we will have

$$|B'\tilde{\cap}B^*| > |B\tilde{\cap}B^*|$$

If this is true, then there exist a path in the solution graph from B to B^*. We prove these claims in Lemma 6 and Theorem 7.

Lemma 6. *Let B and B^* be two distinct maximal connected induced bipartite subgraphs, $\dot{v}$ the earliest vertex in the canonical ordering B^* that is not in B, and* NEIGH() *the function in Definition 3. Then $|B'\tilde{\cap}B^*| > |B\tilde{\cap}B^*|$ for either $B' = \textsc{neigh}(B_0, B_1, \dot{v})$ or $B' = \textsc{neigh}(B_1, B_0, \dot{v})$.*

Proof. Firstly, consider the case $|B\tilde{\cap}B^*| = 0$. In this case $\dot{v} = b_1^*$, and since $dv \in B'$ by definition of NEIGH(), $|B'\tilde{\cap}B^*| \geq 1$, which proves the statement.

Otherwise, let $Z = B\tilde{\cap}B^* = \{b_1^*, \dots, b_h^*\}$, and we have $\dot{v} = b_{h+1}^*$. By Definition 4, Z is a connected induced bipartite subgraph, meaning that it allows a unique bipartition Z_0, Z_1 (with Z_0 being the set containing the vertex of smallest label in Z, that is, b_1^*). Since b_1^* is the vertex of smallest label in B^*, it will be in B_0^*, so it follows that $Z_0 \subseteq B_0^*$ and $Z_1 \subseteq B_1^*$.

Let j be the value in $\{0, 1\}$ such that $\dot{v} \in B_j^*$, and observe that $N(\dot{v}) \cap Z_j \subseteq N(\dot{v}) \cap B_j^* = \emptyset$. Furthermore, let i be the value in $\{0, 1\}$ such that $b_1^* \in B_i$, and observe that $Z_i \subseteq B_j^* \cap B_i$ and $Z_{1-i} \subseteq B_{1-j}^* \cap B_{1-i}$.

Finally, let $B' = \textsc{neigh}(B_i, B_{1-i}, \dot{v})$. Z is fully contained in B': the only other vertices of B removed are (i) those in $N(\dot{v}) \cap B_i$, but $N(\dot{v}) \cap B_i \cap Z \subseteq N(\dot{v}) \cap B_i \cap Z_j \subseteq N(\dot{v}) \cap Z_j = \emptyset$, and (ii) the vertices not in the connected component of $\dot{v}$ in $G[\{\dot{v}\} \cup (B_i \setminus N(\dot{v})) \cup B_{1-i}]$, but no such vertex can be in Z as $Z \cup \{\dot{v}\}$ is a prefix of the canonical order of B^*, and thus $G[Z \cup \{\dot{v}\}]$ is connected by definition.

We thus have that $Z \cup \{\dot{v}\} \subseteq B'$, meaning that $\{b_1^*, \dots, b_h^*, b_{h+1}^*\} \subseteq B'\tilde{\cap}B^*$ and thus $|B'\tilde{\cap}B^*| \geq |B\tilde{\cap}B^*| + 1$, proving the statement. □

From this we immediately obtain the correctness of Algorithm 1:

Theorem 7. *Algorithm 1 outputs all maximal connected induced bipartite subgraphs of a given graph G without duplication.*

Proof. The absence of duplication is trivially guaranteed by the set $\mathcal{S}$. Since the result of a COMP() call is always a maximal solution, Algorithm 1 finds at least one maximal solution. Assume now that some solution B^* is not found, and let $B = B_0, B_1$ be the solution found which maximizes $|B\tilde{\cap}B^*|$. By hypothesis $B \neq B^*$, so by Lemma 6 we know that for some v we have $|B'\tilde{\cap}B^*| > |B\tilde{\cap}B^*|$, where B' is either $\textsc{neigh}(B_0, B_1, v)$ or $\textsc{neigh}(B_1, B_0, v)$. This means B' is found by Algorithm 1, which contradicts the hypothesis of B maximizing $|B\tilde{\cap}B^*|$. □

3.3 Listing maximal induced bipartite subgraphs

In this section we briefly how to extend the algorithm to the non-connected case. The correctness proofs are obtained by adapting those for the connected case and can be found in Appendix A.

In short, we prove the strong connectivity between the solutions by showing that we can increase the proximity between a solution B and a target B^* one connected component at a time.

The algorithm for this case is essentially the same, i.e., Algorithm 1, with minimal changes: Firstly, as solutions do not require connectivity, this is reflected on the COMP(B) function, will iteratively add the smallest addible vertex to B, regardless of connectivity. Secondly, the neighboring function used will be that in Definition 2, instead of that in Definition 3.

Finally, for proving the correctness we must provide a suitable proximity function $\tilde{\cap}$. This is obtained by Definition 5, but changing the canonical order as follows:

Definition 8 (canonical order for induced bipartite subgraphs)**.**
Let $C_1^B, \ldots, C_k^B$ be the connected components of an induced bipartite subgraph B, in lexicographical order (i.e., C_i comes before C_j if the vertex of minimum label in C_i is smaller than that in C_j). The canonical order of B is the sequence $b_1, \ldots, b_{|B|}$, obtained by concatenating the canonical orders of its connected components $C_1^B, \ldots, C_k^B$, in this order, where the order of each connected component is obtained by Definition 4.

Thus, taking B as the subgraph shown in Figure 2 (d) and B^* as that shown in (e), the canonical order of B is $\{1, 2, 7, 8, 10, 11\}$, that of B^* is $\{1, 2, 7, 9, 10, 12\}$, and $B \tilde{\cap} B^* = \{1, 2, 7\}$.

We give a proof sketch for the correctness: Consider the solutions B and B^*, a vertex u in $B^* \setminus B$, and let $C_x^{B^*}$ be the connected component of B^* containing u. As all the neighbors of u in B^* must be in its same connected component $C_x^{B^*}$, and the neighbouring function NEIGH(B_i, B_{1-i}, u) (Definition 2) only removes neighbors of u from B, clearly it may not remove from B any vertex of B^* that is *not* in $C_x^{B^*}$. As for $C_x^{B^*}$, which is a connected induced bipartite subgraph, the proximity can be increased by looking at the strategy for the connected case, since each connected component uses the canonical ordering of the connected case. We thus state the result:

Theorem 9. *Algorithm 1, using* NEIGH() *from Definition 2 and relaxing the connectivity requirement in* COMP()*, outputs all maximal induced bipartite subgraphs of a graph G without duplication.*

For a detailed proof, refer to Appendix A.

3.4 Complexity

For brevity, we just show that the algorithms presented have polynomial delay. A finer complexity analysis can be found in Appendix B.

It is easy to see from the pseudo code of Algorithm 1 that we can bound the cost per solution by that of one iteration of ENUM(). Indeed, whenever we find a new solution, the cost of the corresponding iteration can be attributed to it.

Let γ be a bound for the cost checking whether a solution is in $\mathcal{S}$ or adding it to $\mathcal{S}$, and δ a bound for the cost of NEIGH(). The cost per solution of the algorithm will be $O(\gamma + n(\delta + \gamma))$.

Firstly, γ is surely polynomial as we can store $\mathcal{S}$ as, e.g., a binary search tree with access time $O(n)$, as n is the size of the universe on which solutions are defined. Furthermore, δ is dominated by the cost of a COMP() call: as we can check in linear time whether a graph is connected and/or bipartite, COMP() takes polynomial time.

For completeness we remark that the space usage is bounded by the size of $\mathcal{S}$, that is polynomial not only in the size of the input, but in that of the output as well (e.g., a simple binary search tree takes $O(n \cdot |\mathcal{S}|)$ space, where $|\mathcal{S}|$ is the number of solutions).

Finally, since every recursive call generated outputs a solution, we may use the *alternative output* technique [32]: by simply changing the moment in which output is performed, we can reduce the delay of Algorithm 1 to match exactly the cost per solution (up to a constant factor).

We can observe that COMP() can be trivially implemented by testing each of the $O(n)$ vertices for addition in $O(m)$ time, for a total of $O(nm)$ time, giving us a delay bounded by $O(n^2m)$, that is polynomial.

By the more careful analysis shown in Appendix B, however, we obtain the following:

Theorem 10. *Algorithm 1 lists all maximal connected induced bipartite subgraphs with delay $O(nm)$, and all maximal induced bipartite subgraphs with delay $O(n(m + n\alpha(n)))$.*

Where n and m are the number of vertices and edges, and $\alpha()$ is the functional inverse of the Ackermann function [29].

3.5 Maximal Edge Bipartite Subgraphs

We here briefly mention how to adapt the above algorithm to Maximal *Edge* Bipartite Subgraphs, where edge subgraphs are denoted by a set of edges, rather than vertices. A complete description, with correctness and complexity analysis is given in Appendix C.

In the following, let $E(A, B)$ be the set of edges with one endpoint in A and the other in B, where A and B are two disjoint sets of vertices.

Firstly, we observe Maximal Edge Bipartite Subgraphs (MEBS in the following) of a connected graph are always connected (otherwise some edge could be added without creating odd cycles), spann all vertices, and may thus be represented by a bipartition B_0, B_1 of $V(G)$, where the bipartite subgraph corresponds to the edges in $E(B_0, B_1)$.

We also observe that the problem is hereditary and allows a polynomial time computable COMP() function. We define the canonical order of a solution B by taking $b_1, \ldots, b_{|}B|$ as the canonical order of the *vertices* of $G[B]$ (according to Definition 4), then taking the edges of B in increasing order of their *latter* vertex in the vertex order, and breaking ties by increasing order of the earlier extreme. This essentially corresponds to "building" B in a similar fashion as in the induced version, but adding one edge at a time incident to the newly selected vertex.

The principle behind the neighboring function is different but inspired to the induced case: rather than taking a vertex out of the solution and trying to add it to B_0 or B_1, we take an edge $e = \{a, b\}$ with both extremes in the same B_i, and try to move the two vertices a and b to opposite sides of the bipartition.

This can be achieved by including the edge e in the solution, and then, to preserve the subgraph being bipartite, removing either $N_E(a)$ or $N_E(b)$ from it. Finally we apply the COMP() function to obtain a solution that is maximal. More formally:

$$\text{NEIGHBORS}(B) = \bigcup_{e \in E(G) \setminus E(B_i, B_{1-i})} (\text{NEIGH}(B, a, b) \cup \text{NEIGH}(B, b, a))$$

Where

$$\text{NEIGH}(B, a, b) = \text{COMP}((E(B_i, B_{1-i}) \setminus N_E(a)) \cup \{e\})$$

For brevity, the proof of correctness are moved to Appendix C, but it can be proved that, for any pair of distinct solutions B and B^*, there is a solution $B' \in \text{NEIGHBORS}(B)$ such that $|B' \tilde{\cap} B^*| > |B \tilde{\cap} B^*|$, which yields the following resultç

Theorem 11. *Maximal Bipartite (edge) Subgraphs can be listed in $O(m^3)$ time delay.*

4 Generalized algorithm for proximity search

It is worth observing that the proof of correctness for the algorithms above (Theorems 7 and 9) is not based on the specific structure of the problem at hand. We can deduct that the structure of Algorithm 1 is adaptable the enumeration of any type of (maximal) subgraph, as long as suitable $\tilde{\cap}$ and NEIGH() operations are provided to prove a statement equivalent to that of Lemma 6.

In other words, we can obtain a listing algorithm by simply designing a suitable NEIGH() function, and a proximity ($\tilde{\cap}$) notion able to prove that the solution graph induced by NEIGH() is strongly connected.

We devote this section to formalize a class of problem for which this method can be applied.

For convenience, we show in Algorithm 2 the pseudo code corresponding algorithm, which traverses the solution graph, based on the NEIGHBORS() function. As we noted earlier, the algorithms obtained by this structure are remarkably simple, in that we only need to implement the NEIGHBORS() function, while their complexity is mostly hidden behind proving their completeness.

Algorithm 2: Traversal of the solution graph by proximity search.

input : A graph $G = (V(G), E(G))$, a listing problem $\mathcal{P}$
output: All (maximal) solutions of P in G
global : Set $\mathcal{S}$ of solutions found, initially empty

```
1 Let S be one arbitrary solution of P
2 ENUM(S)
3 Function ENUM(S)
4 |   Add S to S
  |   /* output S if recursion depth is even                    */
5 |   foreach S' ∈ NEIGHBORS(S) do
6 |   |   if S does not contain S' then ENUM(S')
  |   /* output S if recursion depth is odd                     */
```

This version of the algorithm generalizes the neighboring function, which corresponds to the set of solutions produced when considering the solution S. Note also that we may equivalently check whether S' has already been found before recurring on it.

To give an example of neighboring function, in the case of Algorithm 1 NEIGHBORS(B) would correspond to all the solutions generated on Lines 6-8, i.e., $\bigcup_{u \in V(G) \setminus B} \{\text{NEIGH}(B_0, B_1, v), \text{NEIGH}(B_1, B_0, v)\}$.

Furthermore, we need one arbitrary (maximal) solution S to start the algorithm. We remark that identifying one maximal (not maximum) solution is usually trivial, and can achieved for example by running COMP(v) for some arbitrary $v \in V(G)$ as in Algorithm 1 when a COMP() function is computable in polynomial time.

We formally define the class of problems which allow a polynomial delay algorithm using this structure as *proximity searchable*:

Definition 12 (Proximity searchable)**.**

Let $\mathcal{P}$ be a listing problem whose solutions are subset of a universe $\mathcal{U}$, and let $\mathcal{S} \subseteq 2^{\mathcal{U}}$ be the set of solutions of $\mathcal{P}$. $\mathcal{P}$ is proximity searchable if it allows a proximity function $\tilde{\cap} : \mathcal{S} \times \mathcal{S} \to 2^{\mathcal{U}}$ and a neighboring function NEIGHBORS() $: \mathcal{S} \to 2^{\mathcal{S}}$ *that returns a set of solutions of $\mathcal{P}$, such that the following holds:*

- *At least one solution of $\mathcal{P}$ can be identified in time polynomial in $|\mathcal{U}|$.*
- *Given any two distinct solutions $S, S^* \in \mathcal{S}$, $\exists S' \in$ NEIGHBORS$(S) : |S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$.*
- NEIGHBORS(S) *is computable in time polynomial in $|\mathcal{U}|$.*

If a problem is proximity searchable, then it is straightforward to see that we obtain a polynomial delay algorithm for it by using the corresponding NEIGHBORS() function in Algorithm 2: A formal proof of correctness is obtained by trivially adapting that of Theorem 7, and the cost per solution (that is, the delay) is bounded by the cost of NEIGHBORS(S), plus the cost of checking membership of each solution found in the $\mathcal{S}$ set (condition of the *if* on Line 6) and adding S to $\mathcal{S}$, both of which can be done in polynomial time as remarked in Section 3.4.

We can thus state the following:

Theorem 13. *All proximity searchable listing problems allow a polynomial delay listing algorithm.*

Furthermore, the following observations are in order:

- As NEIGHBORS(S) needs to be computable in polynomial time, it must return a polynomial number of solutions.
- $\tilde{\cap}$ is not actually used in Algorithm 2, and does *not* need to be computable in polynomial time.
- Proximity search is mainly intended for maximal listing problems, as the non maximal case usually allows simpler techniques, however it is not strictly limited to it.
- Maximal listing problems in which input-restricted problem is computable in polynomial time (as well as the COMP() function) are proximity searchable.[3]
- The delay of the resulting algorithm will be bounded by the cost of Lines 4-6 in Algorithm 2.

In the rest of the paper we show some other maximal listing problems for which, to the best of our knowledge, polynomial delay algorithms were not known. We show that these problems are proximity searchable, and thus allow a polynomial delay listing algorithm by Algorithm 2.

We will use a common notation: S is an arbitrary solution, S^* the "target" solution, and v a vertex.

We also recall that COMP(X) is a function which, given a non-maximal solution X, iteratively adds to X the element with smallest label such that X is still a solution of the problem at hand, until it is maximal.

Finally, as general we combine solutions with vertices to obtain new ones, rather than defining NEIGHBORS(S), we will use NEIGHBORS(S, v), in which case NEIGHBORS(S) is intended to be $\bigcup_{v \in V(G)}$ NEIGHBORS(S, v).

[3] In essence, we obtain as a special case the same solution graph as known algorithms based on the input-restricted problem [31, 20, 4]: For a solution S we compute NEIGHBORS(S) by sequentially taking all elements $v \in \mathcal{U} \setminus S$, solving the input-restricted problem for $S \cup \{v\}$, and applying COMP() on the results.

5 Maximal k-Degenerate Subgraphs

We here consider the enumeration of maximal k-degenerate subgraphs, giving an algorithm that has polynomial delay when k is bounded.

A graph G is k-degenerate if it allows an elimination ordering where each vertex has degree at most k when deleted. Equivalently, it is k-degenerate if no subgraph of G is a $(k+1)$-core, i.e., a graph where each vertex has degree greater or equal to $k+1$. The *degeneracy* d of G is the highest k for which G is k-degenerate.

A *degeneracy ordering* of G is an order of its vertices in which each vertex v has at most d neighbors occurring later than v, where d is the degeneracy of G. It is well known that a degeneracy ordering can be found in $O(m)$ time by recursively removing the vertex of smallest degree. To remove ambiguity, when multiple vertices have the same degree we can remove the one with smallest label.

The degeneracy is a well known sparsity measure [11], and generalizes maximal independent sets (0-degenerate graphs) and maximal trees and forests (connected and non-connected 1-degenerate graphs). Furthermore, degeneracy is linked to planarity as all planar graphs are 5-degenerate, while outerplanar graphs are 2-degenerate [21].

We are interested in listing all maximal k-degenerate subgraphs of a given graph G. An output-sensitive algorithm is known for maximal induced k-degenerate subgraphs if G is chordal [7], but no output-sensitive results are known for general graphs.

5.1 Maximal Induced k-Degenerate Subgraphs

As a subgraph of a k-degenerate graph is k-degenerate, and degeneracy can be computed in linear time, we may compute the COMP() function in polynomial time.

Given a maximal k-degenerate subgraph S, we define its *canonical order* as the *reverse* of its degeneracy ordering, i.e., an ordering $s_1, \ldots, s_{|S|}$, such that $s_{|S|}, \ldots, s_1$ is the degeneracy ordering of S. In the case of non connected subgraphs, this is adapted by considering the connected components one at a time in lexicographical order. Then, the proximity is defined as in Definition 5.

An important property of this ordering is that $|N(s_i) \cap \{s_1, \ldots, s_{i-1}\}| \leq k$, i.e., the neighbors of s_i in S that precede s_i in the canonical order are at most k.

The neighboring function is then obtained as follows:

Definition 14 (Neighboring function for Maximal Induced k-Degenerate Subgraphs)**.**

$$\text{NEIGHBORS}(S) = \bigcup_{v \in V(G)} \text{NEIGHBORS}(S, v)$$

Where :

$$\text{NEIGHBORS}(S, v) = \{\text{COMP}(\{v\} \cup (S \setminus N(v)) \cup K) : K \subseteq (S \cap N(v)) \text{ and } |K| \leq k\}$$

Less formally, when computing NEIGHBORS(S, v), we try to add v to S. As S is maximal, this violates the degeneracy constraint, so we remove all neighbors of v except at most k (the set K). The resulting subgraph $D = \{v\} \cup (S \setminus N(v)) \cup K$ is clearly k-degenerate as any subgraph of D has a vertex of degree at most k: any subgraph of D either contains v, which has degree at most k, or does not, thus it is a subgraph of S (which is k-degenerate) and as such has a vertex of degree at most k.

We finally use the COMP() function to obtain a maximal solution, which is still k-degenerate by definition of COMP().

As for the choice of K, we simply try all possible subsets of $S \cap N(v)$ of size at most k. These are $O(\sum_{i \in \{1,\dots,k\}} \binom{|N(v)|}{i}) = O(n^k)$, which is polynomial when k is bounded.

In this way, we are able to prove the correctness of the algorithm: Indeed, consider two distinct maximal solutions S and S^*, and let $\dot{v}$ be the first element of the canonical order of S^* that is not in S. In other words we have $S \tilde{\cap} S^* = \{s_1^*, \dots, s_h^*\}$ and $\dot{v} = s_{h+1}^*$. By construction of the canonical order, $\dot{v}$ has at most k neighbors in $\{s_1^*, \dots, s_h^*\}$, all of which are in S. Let the set of these vertices be K'; as $|K'| \leq k$, at some point it will be considered as K by the neighboring function NEIGHBORS(S, v) in Definition 14.

When this happens, we will obtain $S' = \text{COMP}(\{\dot{v}\} \cup (S \setminus N(\dot{v})) \cup K')$. Since S contains $S \tilde{\cap} S^*$, and all neighbors of $\dot{v}$ in $S \tilde{\cap} S^*$ are in K', we have that $S \tilde{\cap} S^* \subseteq (S \setminus N(\dot{v})) \cup K' \subseteq S'$. Thus $(S \tilde{\cap} S^*) \cup \{\dot{v}\} = \{s_1^*, \dots, s_h^*, s_{h+1}^*\} \subseteq S'$, which implies $|S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$.

We thus meet the first two requirements of Definition 12. We only need to show that NEIGHBORS(S) can be computed in polynomial time.

Let us first consider the cost of a COMP(X) call. k-degenerate graphs are a hereditary property, meaning that if a vertex is not addible it will not become addible in the future, thus we need to test each $v \in V(G) \setminus X$ for addition at most once. As testing the degeneracy takes $O(m)$ time, we can compute COMP(X) in $O(mn)$ time.

Let us now consider the complexity of NEIGHBORS(S, v): firstly, we need to enumerate each possible $K \subseteq N(v) \cap S$, which can be done in $O(\sum_{i \in \{1,\dots,k\}} \binom{|N(v)|}{i}) = O(|N(v)|^k)$ time. For each, we need to run COMP$(\{v\} \cup (S \setminus N(v)) \cup K)$, which takes $O(mn)$ time. The total cost is thus bounded by $O(n^k + 1m)$ time.

As NEIGHBORS(S) consists in running NEIGHBORS(S, v) n times, the resulting cost is polynomial when k is of constant size. This means the problem is proximity searchable, and the delay of the listing algorithm is given by the cost of NEIGHBORS(S) (which dominates that of maintaining the solution set). More formally:

Theorem 15. *Maximal Induced k-degenerate Subgraphs are proximity searchable when k is constant, and can be enumerated with delay $O(mn^{k+2})$.*

5.2 Maximal Edge k-Degenerate Subgraphs

An algorithm for this case can be obtained by exploiting the structure of the induced one. In the following, let $N_E(v)$ be the *edge neighborhood* of v, i.e., the set of all edges in the input graph G incident to the vertex v.

Note that also edge k-degenerate subgraphs are hereditary, and thus we can compute the COMP() function in polynomial time.

Let S be an edge k-degenerate subgraph, and let $v_1, \dots, v_l$ be the canonical order of the vertices of $G[S]$ (i.e., the graph containing only the edges in S and the vertices incident to them), obtained as in Section 5.1.

The canonical ordering of S is obtained by selecting the edges of B by increasing order with respect to their *later* extreme in the vertex order, and breaking ties by increasing order of the other (earlier) extreme.

This essentially corresponds to selecting the vertices $v_1, \dots, v_l$ in order, and for each adding the edged towards the preceding vertices one by one. Any time all the edges from v_i to the preceding vertices have been added, we can observe that the graph correspond to the subgraph induced in $G[S]$ by the vertices $\{v_1, \dots, v_i\}$. By the canonical order of the vertices defined in Section 5.1, this means v_i has at most k neighbors in $\{v_1, \dots, v_{i-1}\}$.

Finally, the proximity $\tilde{\cap}$ is again defined using the canonical order as in Definition 5.

We can now define the neighboring function:

Definition 16 (Neighboring function for Maximal Edge k-Degenerate Subgraphs)**.**
Let S be a maximal edge k-degenerate subgraph, and $e = \{a, b\}$ and edge not in S. We define:

$$\textsc{neighbors}(S) = \bigcup_{e=\{a,b\} \in E(G) \setminus S} \textsc{neighbors}(S, a, b) \cup \textsc{neighbors}(S, b, a)$$

Where:

$$\textsc{neighbors}(S, a, b) = \{\textsc{comp}(\{e\} \cup (S \setminus N_E(a)) \cup K) : K \subseteq (S \cap N_E(a)) \text{ and } |K| \leq k-1\}$$

In other words, to find neighboring solutions, we add an edge $e = \{a, b\}$ to S, then force either a (or, respectively, b) to have degree at most k, by removing all other edges incident to it except at most $k-1$, as well as adding e. The resulting graph is surely k-degenerate as all its subgraph will either have a (respectively b), which has degree at most k, or will be subgraphs of S, which is k-degenerate, and thus have a vertex of degree at most k.

Let us now consider two solutions S, S^*, with $S \tilde{\cap} S^* = \{e_1, \ldots, e_h\}$, and let $\dot{e} = \{x, y\}$ be the earliest edge in the canonical order of S^* that is not in S, i.e, e_{h+1}.

Furthermore, assume that x comes earlier than y in the canonical ordering of S^*. By definition of canonical ordering of S, y has at most k neighbors preceding it in the (vertex) ordering, i.e, $|\{e_1, \ldots, e_h\} \cap N_E(y)| \leq k$. Furthermore, by the same definition, all edges incident to y that precede $\dot{e}$ in the ordering must be between y and another vertex which comes earlier than x, and thus than y, in the ordering, thus they may be at most $k-1$ (k, including $\dot{e}$ itself, from y to x). Let K' be the set of these edges (not including $\dot{e}$).

When computing $\textsc{neighbors}(S, y, x)$, we will consider all the subset of edges in S incident to y of size at most $k-1$. By what stated above, at some point we will consider exactly K'. In this case, we will obtain $S' = \textsc{comp}(\{\dot{e}\} \cup (S \setminus N_E(y)) \cup K')$. This must contain all edges in $\{e_1, \ldots, e_h\}$, as we only removed edges neighboring y, but all those in $\{e_1, \ldots, e_h\}$ were in K'. Thus we have $\{e_1, \ldots, e_h\} \cup \dot{e} = \{e_1, \ldots, e_h, e_{h+1}\} \subseteq S'$, which implies $|S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$. The case in which x comes after y in the ordering is similarly satisfied by $\textsc{neighbors}(S, x, y)$.

Finally, we only need to show that $\textsc{neighbors}(S)$ takes polynomial time to compute: indeed this is $O(m)$ times the cost of $\textsc{neighbors}(S, y, x)$, which in turn has the cost of computing $\textsc{comp}()$ once for each possible considered set K. These latter are $O(\binom{N_E(y)}{k-1})$, and the $\textsc{comp}()$ can be easily implemented in $O(m^2)$ (as above, testing degeneracy takes $O(m)$ time and each edge needs to be considered at most once for addition since the problem is hereditary), for a total cost that is polynomial when k is constant. We can thus state the following:

Theorem 17. *Maximal k-degenerate Subgraphs are proximity searchable when k is constant, and can be enumerated with delay $O(\binom{n}{k-1} m^3)$.*

6 Maximal Induced Chordal Subgraphs

A graph G is chordal if every cycle in G of length greater than 3 has a chord, i.e., an edge between two non-consecutive vertices in the cycle. Chordal graphs have been widely studied, and it is known that several problems which are challenging on general graphs become easier on chordal graphs (see, e.g., [3, 23, 2]).

In this section we aim at listing all maximal sets of vertices which induce a chordal subgraph in G. Induced subgraphs of chordal graphs are chordal, thus the problem is hereditary, and chordality can be tested in $O(m)$ time [25], thus we can compute the $\textsc{comp}()$ function in polynomial time.

It is well known that graph (or subgraph) is chordal if and only if it allows a *perfect elimination ordering* $\{v_1, \ldots, v_n\}$ of its vertices, that is, an ordering such that $N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ forms a clique. We can obtain such an ordering by recursively peeling simplicial vertices, i.e., those whose neighborhood is a clique in the residual graph.[4] An important remark is that, since the neighbors of a simplicial vertex form a clique, removing a simplicial vertex cannot disconnect them from each other, i.e., a simplicial vertex is never a cut vertex. It is also known that a chordal graph has at most $O(n)$ cliques, and that a vertex v in a chordal graph may be participate in at most $O(|N(v)|)$ maximal cliques [7].

We use this to define the canonical order, which is then combined with Definition 5 to obtain the proximity function $\widetilde{\cap}$.

Definition 18 (Canonical Order for Maximal Induced Chordal Subgraphs)**.**

The canonical order $\{s_1, \ldots, s_{|S|}\}$ of the maximal induced chordal subgraph S is the reverse of its perfect elimination ordering, i.e., such that $\{s_{|S|}, \ldots, s_1\}$ is the perfect elimination ordering.

In this way, the neighbors of a vertex v that precede v in the ordering form a clique. This applies to both the connected and non connected cases. Furthermore, as we remarked earlier simplicial vertices may not disconnect the graph when removed. This implied that, when S is a connected solution, any prefix $\{s_1, \ldots, s_{j \leq |S|}\}$ of the canonical order will correspond to a connected subgraph.

The neighboring function is defined as follows:

Definition 19 (Neighboring function for Maximal (Connected) Induced Chordal Subgraphs)**.**

We define $\textsc{neighbors}(S) = \bigcup_{v \in V(G) \setminus S} \textsc{neighbors}(S, v)$.

For the non connected case $\textsc{neighbors}(S, v)$ *is defined as:*

$\textsc{neighbors}(S, v) = \{\textsc{comp}((S \setminus N(v)) \cup Q) : Q$ *is a maximal clique containing* v *in* $G[S \cup \{v\}]\}$

While for the connected case it is:

$\textsc{neighbors}(S, v) = \{\textsc{comp}(\mathtt{cc}_v((S \setminus N(v)) \cup Q)) : Q$ *is a maximal clique containing* v *in* $G[S \cup \{v\}]\}$

Less formally, we add a vertex v to S, then remove all its neighbors except one maximal clique. In the connected case we further remove vertices not in the connected component of v. We can observe how this yields a chordal graph, since v itself is simplicial and can be removed, leaving an induced subgraph of S which is chordal and thus allows to complete the elimination ordering.

Again, let S and S^* be two solutions, $S \widetilde{\cap} S^* = \{s_1^*, \ldots, s_h^*\}$ and $\dot{v} = s_{h+1}^*$ the earliest vertex in the canonical order of S^* not in $S \widetilde{\cap} S^*$.

By definition of canonical order, $N(\dot{v}) \cap (S \widetilde{\cap} S^*) = N(\dot{v}) \cap \{s_1^*, \ldots, s_h^*\}$ forms a clique. It follows that when computing $\textsc{neighbors}(S, \dot{v})$, as we try all maximal cliques for some Q we will have $N(\dot{v}) \cap (S \widetilde{\cap} S^*) \subseteq Q$. The resulting S' will thus contain all neighbors of $\dot{v}$ in $S \widetilde{\cap} S^*$, and thus all of $S \widetilde{\cap} S^*$, plus $\dot{v}$, meaning that $|S' \widetilde{\cap} S^*| > |S \widetilde{\cap} S^*|$, which proves the correctness the $\textsc{neighbors}()$ function.

Finally, $\textsc{neighbors}()$ can indeed be computed in polynomial time: in order to identify all maximal cliques containing v in $G[S \cup \{v\}]$, we can simply list all maximal cliques in $G[S \cap N(v)]$, and add v to each. While $G[S \cup \{v\}]$ is not chordal, $G[S \cap N(v)]$ is, since it is an induced subgraph of S. Furthermore, $G[S \cap N(v)]$ has at most $|N(v)|$ vertices and $O(|N(v)|^2)$ edges, so it has at most $|N(v)|$ maximal cliques, which can be listed in $O(|N(v)|^2)$ time, as we recall a chordal graph has $O(n)$ cliques and they may be listed in $O(m)$ time (e.g., by computing a perfect elimination ordering [25]).

[4] To remove ambiguity and associate a unique order to any graph, we can assume that ties are broken lexicographically by taking the vertex of smallest label first when more than one may be removed.

The amortized cost of listing all cliques for each $v \in V(G)\setminus S$ is thus bounded by $O(\sum_{v\in V(G)\setminus S} |N(v)|^2) = O(mn)$ time, and the proces yields at most $O(\sum_{v\in V(G)\setminus S} |N(v)|) = O(m)$ maximal cliques. For each clique Q, we must further compute the corresponding COMP() call: as the problem is hereditary, again we only need to test each vertex at most once for addition, and a chordality can be tested in $O(m)$ time, a COMP() call can be implemented in $O(mn)$ time (which dominates the time for checking membership in $\mathcal{S}$). Furthermore, the same bound applied to the connected case, as we simply need to consider vertices for addition only when they become adjacent to the current solution. Scanning the neighborhoods of the vertices that are added to the solution to find these candidates has an additional cost of $O(m)$ which does not affect the $O(mn)$ bound. The total cost will be $O(mn + m \cdot mn) = O(m^2 n)$

We can thus state that:

Theorem 20. *Maximal Induced Chordal Subgraphs and Maximal Connected Induced Chordal Subgraphs are proximity searchable, and can be listed with $O(m^2 n)$ time delay.*

7 Maximal Obstacle-free Convex Hulls

In appplication domains such as robotics planning and routing, a common problem is finding areas, typically convex, in a given environment which are free from obstacles (see, e.g., [8, 27]).

In this section we solve the following formulation of the problem: we are given two sets of elements V and X, which corresponds to points on a 2-dimensional plane. V represents the point of interest for our application, and X represents the obstacles. For short, let $|V| = n$ and $|X| = h$. We are interested in listing all maximal obstacle-free convex hulls (MOCs for short), where an obstacle-free convex hull is a set of elements $S \subseteq V$ such that the convex hull of S does not contain any element of X.

This problem does not concern a graph, but its solutions are modeled as sets of elements, thus our technique may still be applied.

Again, note that the problem is hereditary, i.e., each subset S' of a solution S clearly also admits a convex hull which does not include elements of X (since it will be contained in that of S).

Consider a maximal solution S and an element $v \in V \setminus S$. As S is maximal, there is at least one element $x \in X$ included in the convex hull of $S \cup \{v\}$. This element x casts two "shadows" S_1 and S_2 on S, seen by v: consider the straight line between v and x, S_1 consists of all elements of S above this line, and S_2 of all those below it. It is straightforward to see how both the convex hull of $S_1 \cup \{v\}$ and that of $S_2 \cup \{v\}$ do not contain x. Any element of S that falls exactly on the line may not participate in any solution involving v.[5] Furthermore, any element $x' \in X$ above this line, and still in the convex hull of $S \cup \{v\}$, further casts two shadows on S_1, as any element below this line casts them on S_2. If we repeat this process for all elements of X in the convex hull of $S \cup \{v\}$ we obtain a number of shadows of S which is at most linear in the number of elements of X. Let $\phi(S, v)$ be the set of these shadows. For each of these shadows $S_i \in \phi(S, v)$, we have that the convex hull of $S_i \cup \{v\}$ may not include elements of X, i.e., $S_i \cup \{v\}$ is a (possibly not maximal) solution.

The neighboring function is then obtained as follows:

Definition 21 (Neighboring function for MOCs)**.**

$$\text{NEIGHBORS}(S) = \bigcup_{v\in V(G)\setminus S} \text{NEIGHBORS}(S, v)$$

[5]Note that it may not fall between v and x otherwise the convex hull of S would have included x.

Where NEIGHBORS$(S, v) = \{\text{COMP}(S_i \cup \{v\}) : S_i \in \phi(S, v)\}$

Finally, for two solutions S and S^*, we simply define $S\tilde{\cap}S^*$ as their intersection $S \cap S^*$ between their elements.

Let $I = S \cap S^* = S\tilde{\cap}S^*$, and v any element in $S^* \setminus S$. Since $I \cup \{v\}$ is contained in a MOC, S^*, its convex hull cannot contain any element of X. It follows that I must be fully contained in a single $S_i \in \phi(S, v)$. We have that the neighboring function will return $S' = \text{COMP}(S_i \cup \{v\})$, with $I \cup \{v\} \subseteq S'$, which implies $|S'\tilde{\cap}S^*| > |S\tilde{\cap}S^*|$. The algorithm is thus correct.

As for the complexity, again the problem is hereditary, so we may compute a COMP(S) call by testing each vertex just once in increasing lexicographical order. The convex hull of S can be computed in $O(|S| \log |S|)$ time, and testing that is a solution consists in checking that each vertex of X is not in it, which can trivially be done in $O(|S| \cdot h)$. For a total cost of $O(n(h + \log n))$. For each candidate v, we have at most h neighboring solutions, and since we need to consider at most n candidates, the delay of the algorithm will be $n \cdot h$ times the cost of a COMP() call.

We thus obtain an algorithm with the following complexity:

Theorem 22. *Maximal Obstacle-free Convex Hulls are proximity searchable, and can be listed in* $O(n^2h(h + \log n))$ *time delay.*

Note that the algorithm is also trivially adapted to the case where the sets of points are more than two, as we simply need to set as V the desired set, and as X the union of all others.

8 Maximal Connected Induced Directed Acyclic Subgraphs

For this problem we consider a *directed* graph, thus each edge has a head and a tail, and its direction is from the tail to the head. We call $N^+(v)$ the *out-neighbors* of the vertex v and $N^-(v)$ its *in-neighbors*.

The goal of this section is listing Maximal Connected Acyclic Induced Subgraphs (MCAIS hereafter) of a given directed graph G.

Note that the problem is connected-hereditary (any connected subgraph of a connected acyclic subgraph is itself connected and acyclic), and acyclicity can be tested in linear time, thus the COMP() can be implemented in polynomial time, e.g., in $O(mn)$.

For completeness, we remark that the non-connected version, i.e., Maximal Induced Directed Acyclic Subgraphs, corresponds exactly to listing the complements of minimal feedback vertex sets in a directed graph, and thus is of no interest since an output-sensitive algorithm was given in [28].

Let us define the canonical order:

Definition 23 (Canonical Order for Maximal Connected Acyclic Induced Subgraphs)**.**

Let S be a MCAIS*. The canonical order of S is the order $\{s_1, \ldots, s_{|S|}\}$ such that, for each s_i, $\{s_1, \ldots, s_i\}$ is connected, and either $\{s_1, \ldots, s_{i-1}\} \cap N^+(s_i) = \emptyset$ or $\{s_1, \ldots, s_{i-1}\} \cap N^-(s_i) = \emptyset$. As more than one one order with these property may exist, let the canonical one be the lexicographically minimum among them.*

We now need to show that such an order always exists for S. However, as the algorithm does not actually need to compute $\tilde{\cap}$ or the canonical order, we do not need to show how to compute the lexicographically minimum one. Note that every MCAIS S allows a *topological order*, that is an order where $\{s_1, \ldots, s_{i-1}\} \cap N^+(s_i) = \emptyset$ (i.e., no vertex has backward out-neighbors), however this may not respect the constraint of $\{s_1, \ldots, s_i\}$ being connected. Furthermore, recall that every acyclic subgraph always has at least one vertex with no in-neighbors (a source) and one with no out-neighbors (a target).

Firstly, let us observe an important property of acyclic graphs with a single source:

Lemma 24. *Let G be a single-source acyclic graph, and $v_1, \ldots, v_n$ any topological order of G. Any prefix $v_1, \ldots, v_i$ of this order induces a connected subgraph.*

Proof. By the property of a topological order, no vertex may have a backward out-neighbor. This implies that v_1 is the (only) source. Any other vertex $v_{j>1}$ has at least one in-neighbor, and by the property of the topological order this is a backward neighbor v_z. By recursively following the backward neighbor of v_z, as the graph is acyclic, we must eventually end up in a source, that is v_1. This means that every vertex v_j is reachable from v_1 using only vertices that come earlier than v_j in the ordering, which proves the statement. □

Note that, as reversing edge directions on a single-source acyclic graph yields a single-target acyclic graph, Lemma 24 also implies that the *reversed* topological order (i.e., where vertices have no forward out-neighbors) is such that every prefix induces a connected subgraph.

We also remark that both these orders satisfy Definition 23.

We now use this lemma to show that acyclic graphs allow the defined canonical order. In the following, we define *collapsing* a set of vertices $A \subseteq S$ in x as replacing them with a single vertex x, whose in- and out-neighbors correspond to all vertices in $S \setminus A$ that are respectively in- and out-neighbors of some vertex in A.

Lemma 25. *Every Directed Acyclic Graph allows a canonical order by Definition 23.*

Proof. Let S be a Directed Acyclic Graph. Let v_1 be a source of S, and S_1 be the set of vertices reachable by v_1, including v_1.

Let $s_{1,1}, \ldots, s_{1,|S_1|}$ a topological ordering of S_1. By Lemma 24 every prefix is connected as v_1 is the only source of S_1.

No vertex in S_1 can have an out-neighbor outside of S_1 as otherwise said vertex would be reachable by v_1 and thus be in S_1 itself. They may, however, have in-neighbors: let S_2 be the set of all vertices in $S \setminus S_1$ which can reach some vertex in S_1.

If we collapse S_1 into a single vertex x, we can observe that x is a target and $S_2 \cup \{x\}$ is a single-target acyclic subgraph. Let $x, s_{2,1}, \ldots, s_{2,|S_2|}$ be a reverse topological ordering of $S_2 \cup \{x\}$. Note that x must be the first vertex as it is the target, and by Lemma 24 each prefix is connected. Furthermore, if we replace x with the previously computed order of S_1, we obtain an order $s_{1,1}, \ldots, s_{1,|S_1|}, s_{2,1}, \ldots, s_{2,|S_2|}\}$ which respects Definition 23: Each vertex in $s_{1,1}, \ldots, s_{1,|S_1|}$ has no backward out-neighbor by the topological ordering of S_1; each $s_{2,1}, \ldots, s_{2,|S_2|}$ has no backward in-neighbor by the reverse topological ordering of S_2, and the fact that no vertex of S_1 can have an out-neighbor outside of S_1; finally, every prefix of $s_{1,1}, \ldots, s_{1,|S_1|}$ is connected, and every prefix $s_{1,1}, \ldots, s_{1,|S_1|}, s_{2,1}, \ldots, s_{2,j}$ is connected, as $x, s_{2,1}, \ldots, s_{2,j}$ is connected, meaning that all vertices in $s_{2,1}, \ldots, s_{2,j}$ are connected to some vertex in S_1, that is itself connected.

We may now repeat this step by collapsing $S_1 \cup S_2$ into a single vertex x', and since x' will be a source (any vertex that could reach one of S_2 would be in S_2), take S_3 as all vertices reached by x' in $S \setminus (S_1 \cup S_2)$, and take a topological order of $S_3 \cup \{x'\}$, which we append to the order obtained so far (excluding x').

By iterating this step, we obtain an ordering $s_{1,1}, \ldots, s_{1,|S_1|}, s_{2,1}, \ldots, s_{2,|S_2|}, s_{3,1}, \ldots, s_{3,|S_3|} \ldots, s_{k,1}, \ldots, s_{k,|S_k|}$, with $k \leq |S|$, which contains all vertices of S, and such that any prefix will induce a connected subgraph, and any $s_{i,j}$ will have no backward out-neighbors if i is odd, and no backward in-neighbors if i is even. This order will thus satisfy the conditions of Definition 23. □

Finally, the proximity $\tilde{\cap}$ is defined accordingly using Definition 5.

We define the neighboring function as follows:

Definition 26 (Neighboring Function for Maximal Connected Acyclic Induced Subgraphs)**.** *For a solution S and a vertex $v \in V(G) \setminus S$, we define*

$$\textsc{neighbors}(S) = \bigcup_{v \in V(G) \setminus S} \textsc{neighbors}(S, v)$$

Where:

$$\textsc{neighbors}(S, v) = \{\textsc{comp}(\mathtt{cc}_v(\{v\} \cup S \setminus N^+(v))), \textsc{comp}(\mathtt{cc}_v(\{v\} \cup S \setminus N^-(v)))\}$$

In other words, the function will add v to S. $S \cup \{v\}$ is not acyclic, but all cycles must involve v, so we make it acyclic by removing either all the out-neighbors $N^+(v)$, which makes v a target, or all its in-neighbors $N^-(v)$, which makes v a source. It then takes the connected component containing v and feeds the result to comp(), to surely obtain a mcais.

Consider now two solutions S and S^*, and again let $\dot{v}$ be the first vertex in the canonical order of S^* which is not in $S \tilde{\cap} S^*$. More formally, let $S \tilde{\cap} S^* = \{s_1^*, \ldots, s_h^*\}$ and $\dot{v} = s_{h+1}^*$.

Let $S' = \textsc{comp}(\mathtt{cc}_{\dot{v}}(\{\dot{v}\} \cup S \setminus N^+(\dot{v})))$ and $S'' = \textsc{comp}(\mathtt{cc}_{\dot{v}}(\{\dot{v}\} \cup S \setminus N^-(\dot{v})))$ be the two solutions generated by $\textsc{neighbors}(S, \dot{v})$.

By definition of the canonical order of S^*, we have that $(S \tilde{\cap} S^*) \cup \{\dot{v}\}$ is connected, and either $(S \tilde{\cap} S^*) \cap N^+(\dot{v}) = \emptyset$ or $(S \tilde{\cap} S^*) \cap N^-(\dot{v}) = \emptyset$.

It follows that if $(S \tilde{\cap} S^*) \cap N^+(\dot{v}) = \emptyset$, then $(S \tilde{\cap} S^*) \cup \{\dot{v}\} \subseteq \mathtt{cc}_{\dot{v}}(\{\dot{v}\} \cup S \setminus N^+(\dot{v})) \subseteq S'$, and otherwise we have $(S \tilde{\cap} S^*) \cap N^-(\dot{v}) = \emptyset$, which means $(S \tilde{\cap} S^*) \cup \{\dot{v}\} \subseteq \mathtt{cc}_{\dot{v}}(\{\dot{v}\} \cup S \setminus N^-(\dot{v})) \subseteq S''$.

We thus have that either $|S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$ or $|S'' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$, which gives us the second necessary condition of proximity search.

Finally, it is straightforward to see that $\textsc{neighbors}(S)$ takes polynomial time, as its cost is bounded by $O(n)$ calls to comp(), which can be implemented in $O(mn)$, meaning that all conditions of Definition 12 are satisfied. Theorem 27 follows.

Theorem 27. *Maximal Connected Directed Acyclic Induced Subgraphs are proximity searchable, and can be listed $O(mn^2)$ time delay.*

8.1 Maximal Connected Directed Acyclic Edge Subgraphs

We remark here that the structure can be adapted to the edge case, i.e., Maximal Connected Directed Acyclic Edge Subgraphs (mcaes).

Firstly note that as the problem is still hereditary and acyclic subgraphs can be tested in linear time, we can implement the comp() function in $O(m^2)$ time. Furthermore, we can define a canonical order as follows:

Definition 28 (Canonical order for mcaes)**.** *Given a mcaes S, let $v_1, \ldots, v_{|V[S]|}$ be the canonical ordering of the vertices of $G[S]$ according to Definition 23.*

The canonical ordering of S is obtained by selecting the edges of S by increasing order with respect to their later extreme in the vertex order, and breaking ties by increasing order of the other (earlier) extreme.

We thus obtain a canonical ordering $e_1, \ldots e_{|S|}$ of S with the following properties: take an edge $e_i = \{v_j, v_k\}$, assuming wlog $j < k$. All edges whose latter extreme comes earlier than v_k in the vertex order are by definition preceding e_i in the order, thus all the edges in the induced subgraph $G[\{v_1, \ldots, v_{k-1}\}]$ will be in the prefix $e_1, \ldots e_i$ of the canonical ordering of S. Note that by Definition 23, $G[\{v_1, \ldots, v_{k-1}\}]$ is connected. Finally, the only other edges in $e_1, \ldots e_i$ are those

whose latter extreme is v_k, meaning that their earlier extreme is in $\{v_1, \ldots, v_{k-1}\}$. It follows that each prefix $e_1, \ldots e_i$ forms a connected (edge) subgraph, which is also acyclic as it is a subgraph of the acyclic subgraph S.

Furthermore, it also holds that, for the latter extreme v_k of e_i, either $\{v_1, \ldots, v_{k-1}\} \cap N^+(v_k) = \emptyset$ or $\{v_1, \ldots, v_{k-1}\} \cap N^-(v_k) = \emptyset$. This implies that either $\{e_1, \ldots, e_{i-1}\} \cap N_E^+(v_k) = \emptyset$, or $\{e_1, \ldots, e_{i-1}\} \cap N_E^-(v_k) = \emptyset$, which gives us our neighboring function:

Definition 29 (Neighboring Function for MCAES)**.**

Let S be a MCAES *and $e = \{v_t, v_h\}$ a directed edge in $E(G) \setminus S$ directed from its tail v_t to its head v_h. Furthermore, let $N_E^+(v_h)$ and $N_E^-(v_h)$ be the out-edges and in-edges of v_h, respectively. We define*

$$\text{NEIGHBORS}(S, v_t, v_h) = \{\text{COMP}(\mathtt{cc}_{v_t}(\{e\} \cup (S \setminus N_E^-(v_t))), \text{COMP}(\mathtt{cc}_{v_h}(\{e\} \cup (S \setminus N_E^+(v_h)))\}$$

And thus

$$\text{NEIGHBORS}(S) = \bigcup_{e=\{v_t,v_h\} \in E(G) \setminus S} \text{NEIGHBORS}(S, v_t, v_h)$$

In other words, we add e to S, and try each of the two possibilities to obtain the latter vertex in the canonical order of S^*: if it is the tail v_t of the edge, surely its backward out-neighborhood in the canonical order of S^* is not empty as it contains v_h, so it's in-neighborhood must be, thus we can safely remove $N_E^-(v_t)$ to make $S \cup \{e\}$ acyclic. Conversely, if it is the head v_h we can safely remove $N_E^+(v_h)$. We thus obtain $|S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$ for some $S' \in \text{NEIGHBORS}(S)$.

9 Maximal Induced Trees

In this section we consider the enumeration of maximal induced trees. A tree is an acyclic connected graph. An induced tree in G is a set of vertices $T \subseteq V(G)$ such that $G[T]$ is a tree.

Relaxing the induced constraint we obtain the problem of listing maximal trees, defined as sets of edges, which correspond to spanning trees, and for which efficient algorithms are known [18]. Relaxing the maximality constraint, we obtain the problem of listing all trees, for which output-sensitive algorithms also exist [33]. Finally, relaxing the connectivity constraint we obtain maximal induced forests, which correspond exactly to the complements of minimal feedback vertex sets, which was solved efficiently in [28].

To the best of our knowledge, however, no output-sensitive algorithms are known for the enumeration of maximal induced trees.

Firstly, we observe that the problem is connected-hereditary, and that we can implement $\text{COMP}(X)$ in $O(m)$ time: first, in $O(m)$ we can test that X is indeed a tree and collect all candidate vertices C which are adjacent to some vertex in X. We then iterate C in increasing order; for each vertex c_i, it is sufficient to check that c_i has no more than one neighbor in X, otherwise it can be discarded, and this can be done in $O(|N(c_i)|)$ time. If c_i is added, we need to also take its neighbors that are not in C and add them to it, which also takes $O(|N(c_i)|)$ time. The resulting cost is thus bounded by $O(m + \sum_{c \in V(G)} |N(c)|) = O(m)$.

The *canonical order* of a solution S can be defined as a BFS traversal order of S, starting from the vertex of minimum label. The resulting order $s_1, \ldots, s_{|S|}$ will be such that each prefix is an induced tree, as it will be a subtree of the BFS tree, and each s_i with $i > 1$ will have exactly

one backward neighbor in this order, corresponding to its parent in the BFS tree. In other words, $|\{s_1, \ldots, s_{i-1}\} \cap N(s_i)| = 1$.

The proximity $\tilde{\cap}$ again follows from Definition 5.

All that needs to be defined is the neighboring function $\textsc{neighbors}(S, v)$ for $v \in V(G) \setminus S$.

Definition 30 (neighboring function for Maximal Induced Trees)**.**

$$\textsc{neighbors}(S) = \bigcup_{v \in V(G) \setminus S} \textsc{neighbors}(S, v)$$

Where

$$\textsc{neighbors}(S, v) = \{\textsc{comp}(\mathsf{cc}_v((S \setminus N(v)) \cup \{v, x\})) : x \in N(v) \cap S\}$$

Less formally, we attempt to add v to S. If v has more than one neighbor in S, this would create a cycle, and we break all such cycles by removing all neighbors of v in S except one, x, for which we try all possible choices. Finally, we feed the result to $\textsc{comp}()$ to obtain a maximal solution. Note that if v has no neighbor in S, $\textsc{neighbors}(S, v) = \{\textsc{comp}(\{v\})\}$. Otherwise, v must have at least two neighbor in S, as if it only had one $S \cup \{v\}$ would be an induced tree, and S would have not been maximal.

We also remark that computing $\textsc{neighbors}(S)$ takes polynomial time, since it consists in calling $\textsc{neighbors}(S, v)$ for each $v \in V(G) \setminus S$, and the cost of each $\textsc{neighbors}(S, v)$ is bounded by computing a connected component and calling $\textsc{comp}()$ (both of which take $O(m)$ time) for each vertex in $N(v) \cap S$. The total cost will thus be bounded by $O(\sum_{v \in V(G)}(|N(v)| \cdot m)) = O(m \sum_{v \in V(G)}(|N(v)|)) = O(m^2)$.

Finally, we need to prove the key condition on the proximity $\tilde{\cap}$.

In the following let $S \tilde{\cap} S^* = \{s_1, \ldots, s_h\}$, and $\dot{v} = s_{h+1}$ be the first vertex in the canonical order of S^* not included in S.

Lemma 31. *Given two distinct Maximal Induced Trees S and S^*, there exists some $S' \in \textsc{neighbors}(S)$, such that $|S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$.*

Proof. Recall that $\{s_1, \ldots, s_h\} = S \tilde{\cap} S^*$ is an induced tree, and so is $\{s_1, \ldots, s_h\} \cup \{\dot{v}\}$ as they are both prefixes of the canonical order of S^*. We also know that $\dot{v}$ has exactly one neighbor $\dot{x}$ in $\{s_1, \ldots, s_h\}$ (unless $\dot{v} = s_1$, in which case the proof is trivial as $|S \tilde{\cap} S^*| = 0$).

Let S' be the solution obtained by $\textsc{neighbors}(S, \dot{v})$ when $x = \dot{x}$. Clearly, no vertex of $\{s_1, \ldots, s_h\}$ is removed from S since $\dot{x}$ is the only neighbor of $\dot{v}$ in S, thus S' contains $\{s_1, \ldots, s_h\} \cup \{\dot{v}\}$, proving the statement. □

We can thus state the result:

Theorem 32. *Maximal Induced Trees in a graph are proximity searchable, and can be listed in $O(m^2)$ time delay.*

10 Conclusions

We presented proximity search, a technique for the design of efficient enumeration algorithm, based on defining and traversing a solution graph where solutions have bounded out-degree. We presented several application cases, considering problems that did not allow efficient algorithms with known techniques, and showing that these allow polynomial delay algorithms using proximity search.

Other than being a useful tool for algorithm design, we remark that the technique may bring useful insight in better understanding which classes of problems allow output sensitive algorithms and which do not. Future work involves extending the applicability of the techniques to open listing problems, and investigating whether the technique can be modified to use just polynomial space.

References

[1] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21 – 46, 1996.

[2] Jean R. S. Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In *Graph theory and sparse matrix computation*, pages 1–29. Springer, 1993.

[3] L. Sunil Chandran. A linear time algorithm for enumerating all the minimum and minimal separators of a chordal graph. In Jie Wang, editor, *Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON)*, pages 308–317, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[4] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *Journal of Computer and System Sciences*, 74(7):1147 – 1159, 2008.

[5] A. Conte, R. Grossi, A. Marino, and L. Versari. Listing Maximal Subgraphs in Strongly Accessible Set Systems. *ArXiv e-prints*, March 2018.

[6] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca 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*, pages 148:1–148:15, 2016.

[7] Alessio Conte, Mamadou Moustapha Kanté, Yota Otachi, Takeaki Uno, and Kunihiro Wasa. Efficient enumeration of maximal k-degenerate subgraphs in a chordal graph. In *Computing and Combinatorics - 23rd International Conference, COCOON 2017, Hong Kong, China, August 3-5, 2017, Proceedings*, pages 150–161, 2017.

[8] Robin Deits and Russ Tedrake. *Computing Large Convex Regions of Obstacle-Free Space Through Semidefinite Programming*, pages 109–124. Springer International Publishing, Cham, 2015.

[9] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2005.

[10] Khaled Elbassioni, Kazuhisa Makino, and Imran Rauf. Output-sensitive algorithms for enumerating minimal transversals for some geometric hypergraphs. In *European Symposium on Algorithms*, pages 143–154. Springer, 2009.

[11] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithmics*, 18, 2013.

[12] Fedor V. Fomin, Fabrizio Grandoni, Artem V. Pyatkin, and Alexey A. Stepanov. Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications. *ACM Trans. Algorithms*, 5(1):9:1–9:17, December 2008.

[13] Petr A. Golovach, Pinar Heggernes, Mamadou Moustapha Kanté, Dieter Kratsch, Sigve H. Sæther, and Yngve Villanger. Output-polynomial enumeration on graphs of bounded (local) linear mim-width. *Algorithmica*, 80(2):714–741, Feb 2018.

[14] Petr A Golovach, Pinar Heggernes, Dieter Kratsch, and Yngve Villanger. An incremental polynomial time algorithm to enumerate all minimal edge dominating sets. *Algorithmica*, 72(3):836–859, 2015.

[15] Alain Gély, Lhouari Nourine, and Bachir Sadi. Enumeration aspects of maximal cliques and bicliques. *Discrete Applied Mathematics*, 157(7):1447 – 1459, 2009.

[16] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119 – 123, 1988.

[17] Mamadou Moustapha Kanté, Vincent Limouzy, Arnaud Mary, and Lhouari Nourine. On the enumeration of minimal dominating sets and related notions. *SIAM Journal on Discrete Mathematics*, 28(4):1916–1929, 2014.

[18] Sanjiv Kapoor and Hariharan Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM Journal on Computing*, 24(2):247–265, 1995.

[19] Stefan Kratsch and Magnus Wahlström. Compression via matroids: a randomized polynomial kernel for odd cycle transversal. *ACM Transactions on Algorithms (TALG)*, 10(4):20, 2014.

[20] Eugene L. Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[21] Don R Lick and Arthur T White. k-degenerate graphs. *Canadian J. of Mathematics*, 22:1082–1096, 1970.

[22] John W. Moon and Leo Moser. On cliques in graphs. *Israel journal of Mathematics*, 3(1):23–28, 1965.

[23] Yoshio Okamoto, Takeaki Uno, and Ryuhei Uehara. Linear-time counting algorithms for independent sets in chordal graphs. In Dieter Kratsch, editor, *Graph-Theoretic Concepts in Computer Science: 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, pages 433–444, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[24] Marvin C Paull and Stephen H Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, (3):356–367, 1959.

[25] Donald J Rose, Robert Endre Tarjan, and George S Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, 5(2):266–283, 1976.

[26] Frank Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

[27] Sergei Savin. An algorithm for generating convex obstacle-free regions based on stereographic projection. In *Control and Communications (SIBCON), 2017 International Siberian Conference on*, pages 1–6. IEEE, 2017.

[28] Benno Schwikowski and Ewald Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(1-3):253–265, 2002.

[29] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[30] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[31] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[32] Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms, 2003. NII Technical Report NII-2003-004E, Tokyo, Japan.

[33] Kunihiro Wasa, Hiroki Arimura, and Takeaki Uno. Efficient enumeration of induced subtrees in a k-degenerate graph. In *International Symposium on Algorithms and Computation*, pages 94–102. Springer, 2014.

[34] Kunihiro Wasa and Takeaki Uno. Efficient enumeration of bipartite subgraphs in graphs. In Lusheng Wang and Daming Zhu, editors, *Computing and Combinatorics*, pages 454–466, Cham, 2018. Springer International Publishing.

A Listing maximal induced bipartite subgraphs

In this section we give the full correctness proof for the algorithm described in Section 3.3, that is Algorithm 1, adapted to maximal induced bipartite subgraphs (i.e., without connectivity requirements).

Firstly, recall that the $\textsc{comp}(B)$ function is adapted according to the problem at hand, i.e., it will iteratively add the smallest addible vertex to B, regardless of connectivity. Secondly, the neighboring function is that in Definition 2.

We also recall that the the proximity function $\tilde{\cap}$ is defined accordingly to Definition 5, using the canonical order in Definition 8 (which essentially is the same as that for the connected case, but computed for one connected component at a time, ordering the components lexicographically)

To prove the correctness, we will prove that we are always able to increase the size of the proximity by 1, either by further completing a partially included connected component, or by adding a vertex from the first empty one when all previous ones are completed.

Similarly to above, we consider two distinct maximal solutions B and B^*, with $B\tilde{\cap}B^* = \{b_1^*, \ldots, b_h^*\}$, and define $\dot{v}$ as b_{h+1}^*. Our goal is proving Lemma 6 for the non-connected case:

Lemma 33. *Let B and B^* be two distinct maximal induced bipartite subgraphs, $\dot{v}$ the earliest vertex in the canonical ordering B^* that is not in B, and* $\textsc{neigh}()$ *the function in Definition 2. Then $|B'\tilde{\cap}B^*| > |B\tilde{\cap}B^*|$ for either $B' = \textsc{neigh}(B_0, B_1, \dot{v})$ or $B' = \textsc{neigh}(B_1, B_0, \dot{v})$.*

Proof. We proceed in the same way as for Lemma 6: When $|B\tilde{\cap}B^*| = 0$ we have $\dot{v} = b_1^*$, and $\{\dot{v}\} \subseteq B'\tilde{\cap}B^*$ by definition of $\textsc{neigh}()$, proving the statement.

Otherwise, let $Z = B\tilde{\cap}B^* = \{b_1^*, \ldots, b_h^*\}$, and we have $\dot{v} = b_{h+1}^*$). By Definition 4, Z is an induced bipartite subgraph, with connected components $C_1^Z, \ldots, C_l^Z$. Still by Definition 4, for $1 \leq x < l$ we have $C_x^Z = C_x^{B^*}$, and $C_l^Z \subseteq C_l^{B^*}$.

We consider two cases:

$C_l^Z = C_l^{B^*}$: In this case we have $\dot{v} \in C_{l+1}^{B^*}$, meaning that $\dot{v}$ is not connected to any vertex in Z. As $\textsc{neigh}(B_j, B_{1-j}, \dot{v})$ only removes vertices in $N(\dot{v})$ from B, it follows that B' will fully contain Z and $\dot{v}$.

$C_l^Z \subset C_l^{B^*}$: In this case we have $\dot{v} \in C_l^{B^*}$, and that $C_l^Z \cup \{\dot{v}\}$ is a connected induced bipartite subgraph, meaning that all vertices in $N(\dot{v}) \cap C_l^Z$ are in the same partition B_i of B. It follows that $B' = \textsc{neigh}(B_{1-i}, B_i, \dot{v})$ will place $\dot{v}$ in B_{1-i}, and no vertex of C_l^Z will be removed, nor any vertex of $C_{x<l}^Z$, since they are separate connected components and may not contain neighbors of $\dot{v}$. B' will thus contain Z and $\dot{v}$.

In both cases, B' will contain $Z \cup \{\dot{v}\}$, meaning that $Z \cup \{\dot{v}\} = \{b_1^*, \ldots, b_h^*, b_{h+1}^*\} \subseteq B' \tilde{\cap} B^*$, which proves the statement. □

Thus, by simply retracing the proof of Theorem 7, using Lemma 33 in place of Lemma 6, we obtain the correctness of the algorithm, that is the proof of Theorem 9.

Corollary 33.1. *Algorithm 1, using* $\textsc{neigh}()$ *defined in Definition 2, outputs all maximal induced bipartite subgraphs of a given graph G without duplication.*

B Complexity of Maximal Bipartite Subgraph Enumeration

Let us discuss the complexity of Algorithm 1 for listing maximal (connected) induced bipartite subgraphs.

Firstly, note that before generating a recursive call, we check that the solution produced has not been found already, meaning that the cost of nested recursive calls can be shouldered by the new solutions found in them. This means that the cost per solution is bounded by that of a single recursive call.

The cost per solution will thus be the time required to add a solution to $\mathcal{S}$, plus $O(n)$ times that of performing the $\textsc{neigh}()$ operation, which is that of a $\textsc{comp}()$ call, and checking whether the result is in $\mathcal{S}$.

B.1 Cost of maintaining $\mathcal{S}$

We can store $\mathcal{S}$ as a binary search tree, where vertices are considered in order $v_1, \ldots v_n$ on descending paths, where each node has a child corresponding to the presence of the considered vertex and one corresponding to its absence. A descent in this tree is enough to either add or check existence of a solution in it, and this takes $O(n)$ time (as the depth of the tree is n, in the case of edge subgraphs this would be $O(m)$ time). Furthermore, we can avoid storing empty subtrees which do not contain any solutions, meaning that the space required to store $\mathcal{S}$ will be $O(n \cdot |\mathcal{S}|)$, where $|\mathcal{S}|$ denotes the number of solutions in it, since it will have exactly $|\mathcal{S}|$ leaves, each with at most n ancestors.

B.2 Cost of a comp(X) call

Let us first consider the non connected case: a trivial implementation consist in trying to add each vertex v in $V(G) \setminus X$ at each step and check whether the resulting $X \cup \{v\}$ is a solution: this way we perform $O(n)$ steps, and try $O(n)$ vertices each step. Checking whether a set of vertices is a (connected) bipartite subgraph can trivially be done in $O(m)$ time, thus the cost is bounded by $O(n^2 m)$.

However, we can obtain better bounds. If we keep track of the connected components via union-find, to test a vertex v we must just check that it does not connect to two vertices in different partitions C_0 and C_1 of the same connected component C of X: this can be done in $O(|N(v)|)$. Updating the union-find can be done in total $O(n\alpha(n))$, where $\alpha()$ is the functional inverse of the Ackermann function [29]. Finally, once we tested a vertex, if this was not addible, it will never become addible, thus we only need to test each vertex once (in increasing lexicographical order). The cost will be the sum of their degrees, that is bounded by $O(m)$. The total time is thus $O(m + n\alpha(n))$.[6]

As for the connected case, we must test at each time only vertices adjacent to X, in increasing lexicographical order. All vertices adjacent to X can be found in $O(\sum_{x \in X} |N(x)|)$. Whenever adding a vertex v to X, we can further update the list of vertices adjacent to X with its neighbors in $O(|N(v)|)$. As each vertex is only added once, the cost is bounded by $O(m)$. As above, if a vertex is not addible, it can be immediately discarded, thus each vertex is testes for addibility at most once. Testing addibility of v can be done in $O(|N(v)|)$ by simply checking that it is not adjacent to both vertices in X_0 and X_1, for a total of $O(m)$. The total cost for the connected case will thus be $O(m)$.

B.3 Total cost

The total cost of the algorithm is given by the cost of a recursive call. This is bounded by $O(\gamma + n(\textsc{comp} + \gamma))$, where $\gamma = O(n)$ is the cost of adding a solution to $\mathcal{S}$ or checking whether it is contained in it.

From what discussed above, this is bounded for the non connected case by

$$O(n + n(m + n\alpha(n))) = O(n(m + n\alpha(n)))$$

While, for the connected case, it is bounded by $O(mn)$. We remark that, in real instances, the two bounds are practically equivalent. Finally, since every recursive call generated outputs a solution, we may use the *alternative output* technique [32]: whenever the *recursion depth* of a call in Algorithm 1 is even, we output the solution on Line 5 as shown, but when it is odd, we output the solution at the end of the call (i.e., as if moving Line 5 to after the *for* loop of Lines 6-8).

Thanks to this simple tweak, the *delay* of the algorithm, i.e., the maximum time elapsed between two consecutive outputs, becomes the same as the amortized cost per solution.

C Maximal Edge Bipartite Subgraphs

We here show that we may adapt our proximity search algorithm for Maximal *Induced* Bipartite Subgraphs to listing Maximal *Edge* Bipartite Subgraphs, where edge subgraphs are denoted by a set of edges, rather than vertices.

The problem is inherently different as, for example, the induced bipartite subgraphs of a complete graph have at most 2 vertices, while the edge bipartite graph can have linear size (e.g., a spanning tree of the graph).

However, structural similarities allows us to obtain a canonical order and a suitable neighboring function for this case too.

In the following, let $E(A, B)$ be the set of edges with one endpoint in A and the other in B, where A and B are two disjoint sets of vertices.

[6] As $\alpha(n)$ grows extremely slowly, we observe that $O(m + n\alpha(n))$ is in essence the same as $O(m)$ on real, finite, graphs.

Firstly, we observe that while a Maximal Edge Bipartite Subgraph (MEBS in the following) of a connected graph is a set of edges, it may be represented by two sets of vertices. Indeed:

- Any MEBS is connected, otherwise it could be extended by joining one connected component or isolated vertex to another with a single edge, and the result would still be bipartite.
- Any MEBS spans all vertices of the graph, as otherwise the same logic as above applies to vertices not spanned by the solution.
- Any connected bipartite subgraph allows a single bipartition B_0, B_1 a defined above, and the subgraph is maximal iff the edges it contains are all and only those between B_0 and B_1, i.e., $E(B_0, B_1)$.

Note that, as MEBS are always connected, we do not need to separately consider the connected and non-connected cases.

We will refer in the following to a bipartite subgraphs by the set of edges they contain (e.g., $E(B_0, B_1)$), and to *maximal* bipartite subgraphs by either that, or the bipartition B_0, B_1 of the vertices of G which induces it.

Then, let us observe that the problem is hereditary, and since being bipartite can be tested in $O(m)$ time the COMP() function can be implemented in $O(m^2)$ time. Let us now define the canonical order of a solution:

Definition 34 (Canonical order for MEBS)**.** *Given a* MEBS $B = B_0, B_1$*, let* $b_1, \ldots, b_{|}B|$ *be the canonical ordering of the vertices of* $G[B]$ *according to Definition 4.*

The canonical ordering of B *is obtained by selecting the edges of* B *by increasing order with respect to their later extreme in the vertex order, and breaking ties by increasing order of the other (earlier) extreme.*

In other words, we sequentially considering each vertex b_i in increasing order, and for each we add all edges in B incident to b_i and some $b_{j<i}$, in increasing order of j.

We can observe how this essentially corresponds to "building" B in a similar fashion as in the induced version, but adding one edge at a time incident to the newly selected vertex. It follows that:

Lemma 35. *Given a* MEBS B *and the canonical ordering of its edges* $e_1, \ldots, e_{|E(B_0,B_1)|}$*, any prefix of this order is a connected bipartite subgraph.*

The principle behind the neighboring function is different but inspired to the induced case: rather than taking a vertex out of the solution and trying to add it to B_0 or B_1, we take an edge $e = \{a, b\}$ with both extremes in the same B_i, and try to move the two vertices a and b to opposite sides of the bipartition.

This can be achieved by including the edge e in the solution, and then, to preserve the subgraph being bipartite, removing either $N(a)$ or $N(b)$ from it. Finally we apply the COMP() function to obtain a solution that is maximal. More formally:

Definition 36 (neighboring function for Maximal Bipartite Subgraphs)**.** *Let* B_i, B_{1-i} *be a Maximal Bipartite Subgraph, and* $e = \{a, b\}$ *such that* $\{a, b\} \subseteq B_i$*.*

$$\text{NEIGHBORS}(B) = \bigcup_{e \in E(G) \setminus E(B_i, B_{1-i})} (\text{NEIGH}(B, a, b) \cup \text{NEIGH}(B, b, a))$$

Where

$$\text{NEIGH}(B, a, b) = \text{COMP}((E(B_i, B_{1-i}) \setminus N_E(a)) \cup \{e\})$$

Note that $\{a, b\} \subseteq B_i$ fully characterizes all edges which are not in $E(B_0, B_1)$, as if $a \in B_0$ and $b \in B_1$ or vice versa, then the edge $\{a, b\}$ could be included in B, meaning that B would not be maximal.

Less formally, we force the edge $e = \{a, b\}$ into the solution, then make the graph bipartite by removing either the edges incident to a or those incident to b (except e). Indeed, any odd cycle introduced by the addition of e is broken by either of these operation since either a or b is left with degree one.

Furthermore, we can show that this function satisfies the requirements of proximity search. Firstly note that it is clearly computable in polynomial time, as it consists in $O(m)$ calls to the NEIGHBORS(B_i, B_{1-i}, e) function, whose cost is bounded by that of COMP(), that is $O(m^2)$.

Secondly, consider two solution B and B^*, with $e_1, \ldots, e_{|E(B_0^*, B_1^*)|}$ being the canonical order of B^*. Furthermore, let $B \tilde{\cap} B^* = \{e_1, \ldots, e_h\}$ and $\dot{e} = e_{h+1} = \{a, b\}$ the first edge in the ordering of B^* which is not in B.

By the definition of the canonical ordering, we have that $\{e_1, \ldots, e_h\}$ is a connected bipartite subgraph, meaning that it allows a unique bipartition $B' = B'_0, B'_1$ of its incident vertices. As $\{e_1, \ldots, e_h\} \cup \{\dot{e}\}$ is also a connected bipartite subgraph, for some $j \in \{0, 1\}$ we must have both $N(a) \cap B'_j = \emptyset$ and $N(b) \cap B'_{1-j} = \emptyset$.

Since B' is included in B, we must have either (i) $B'_j \subseteq B_i$ and $B'_{1-j} \subseteq B_{1-i}$ or (ii) $B'_{1-j} \subseteq B_i$ and $B'_j \subseteq B_{1-i}$. Recall now that both a and b are assumed wlog to be in B_i, meaning that $N(a) \cap B_i = N(b) \cap B_i = \emptyset$. In the (i) case, we have $N(b) \cap B_{1-i} \cap B'_{1-j} = \emptyset$, so removing $N_E(b)$ from B may not remove any edge of B'. Thus NEIGH(B, b, a) will contain $(B \tilde{\cap} B^*) \cup \{e\}$.

In the (ii) case, we have $N(a) \cap B_{1-i} \cap B'_{1-j} = \emptyset$, removing $N_E(a)$ may not remove any edge of B'. Thus NEIGH(B, a, b) will contain $(B \tilde{\cap} B^*) \cup \{e\}$.

In both cases, NEIGHBORS(B) will yield a solution S' which contains $(B \tilde{\cap} B^*) \cup \{e\}$, i.e., such that $|S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$.

This means that all the conditions of proximity search (Definition 12) are satisfied, and we can state the result:

Theorem 37. *Maximal Bipartite (edge) Subgraphs are proximity searchable, and can be listed in $O(m^3)$ time delay.*