Listing Acyclic Subgraphs and Subgraphs of Bounded Girth in Directed Graphs

Alessio Conte[1(✉)], Kazuhiro Kurita[2], Kunihiro Wasa[3], and Takeaki Uno[3]

[1] Università di Pisa, Pisa, Italy
conte@di.unipi.it
[2] Hokkaido University, Sapporo, Japan
k-kurita@ist.hokudai.ac.jp
[3] National Institute of Informatics, Tokyo, Japan
{wasa,uno}@nii.ac.jp

Abstract. The *girth* of a directed graph is the length of its shortest directed cycle. We consider the problem of generating all subgraphs of girth at least g in a directed graph G with n vertices and m edges. This generalizes the problem of generating acyclic subgraphs (i.e., with no directed cycle), that correspond to the subgraphs of girth at least $n+1$. The problem of finding the acyclic subgraph with maximum size or weight has been thoroughly studied, however to the best of our knowledge there is no known efficient enumeration algorithm. We propose polynomial delay algorithms for listing both *induced* and *edge* subgraphs with girth g in time $O(n)$ per solution; both improve upon a naive solution, respectively by a factor $O(nm)$ and $O(m^2)$. Furthermore, this work is on the line of existing research for extracting acyclic structures from graphs.

1 Introduction

The problem of extracting directed acyclic structures from graph has been object of study in different forms. Some works, e.g. [5,17], consider the problem of directing the edges of an undirected graph so that the resulting directed graph is acyclic. Berger and Shor [1] considered the problem of finding the acyclic *edge* subgraph with the largest number of edges, while Grotsche et al. [10] studied the more general one of finding the acyclic subgraph of maximum edge weight in a graph with weighted edges. Algorithms that find the best solution with respect to some goal function, e.g., maximize size or weight, are often the tool of choice when a clear goal function can be identified. In real-world situations, however, optimizing some desired properties of the solution may negatively impact other aspects or properties, and a rigorous goal function may not be easy to find. In these situations, a fast enumeration algorithm can be a powerful tool. An enumeration algorithm will report all solutions to the user, letting him judge its goodness with an arbitrarily complex metric. Furthermore, different goal functions require *ad-hoc* algorithms to find the best solution, while an enumeration algorithm may be used in combination with any such function.

This work was supported by JST CREST, Grant Number JPMJCR1401, Japan.

This motivates the problem considered in this work, that is efficiently finding *all* connected acyclic subgraphs of a directed graph G with n vertices and m edges. We solve this problem for both *induced* subgraphs (defined by a subset of the vertices) and *edge* subgraphs (defined by a subset of the edges). Furthermore, we generalize this problem to that of finding connected subgraphs with lower bounded (directed) *girth*, that is the length of the shortest directed cycle in G: a cycle may have length at most n, which makes the subgraphs with girth lower bounded by $n+1$ (i.e., at least $n+1$) exactly the acyclic subgraphs of G. Finally, we will show that the connectivity constraint can be easily dropped from the algorithm, solving the enumeration problem also when the connectivity is not required.

A common way to evaluate the efficiency of an enumeration algorithm is by considering its running time with respect to the number of solutions found. If m is the size of the input, and α the number of subgraphs found by the algorithm, we say that the algorithm runs in *polynomial total time* if the running time is $poly(\alpha, m)$, and *amortized polynomial* if the running time is $\alpha \cdot poly(m)$, i.e., $poly(m)$ *amortized time per solution*. Finally, we say that an algorithm has *polynomial delay* if the time elapsed between finding the i-th and $i+1$-th solution is bounded by $poly(m)$ [13].

We first describe a baseline naive approach which runs in $O(n^2m)$ and $O(m^2n)$ time per solution respectively, for induced subgraphs and edge subgraphs with girth g. We then use structural properties of the problem and support data structures to produce two algorithms, for listing induced and edge subgraphs with girth g, both of which run in $O(n)$ time per solution, i.e., improving the baseline by a factor $O(nm)$ and $O(m^2)$, respectively.

The girth of a graph is related to many fundamental graph properties, e.g., average and minimum degree, diameter, chromatic number, and tree-width [3,6,7]. Many studies consider properties of graphs with the given girth: Thomassen [18] proved that a graph with girth at least five is 3-list-colorable, and Hayes [11] proposed an efficient algorithm for finding a random k-coloring of such graphs. Borodin et al. [2] linked the girth of a graph to its *circular chromatic number*. Furthermore, Galluccio et al. [9] showed that in graphs without a specific minor the circular chromatic number is arbitrarily close to two if the girth is large enough. In addition, several W[1]-hard or W[2]-hard problems, e.g., dominating set, independent set, and set cover become FPT if a graph has large girth [16].

Finding the girth of a graph is a problem that has been studied for decades, but that continues to be object of significant advancement even in recent years. Itai and Rodeh showed the first non-trivial algorithm for finding the girth of an undirected graph in 1978 [12], which runs in $O(mn)$ time. In 2000, Djidjev [8] improved this bound to $O(n^{5/4} \log n)$ for planar graphs. This was further improved by Chang et al. in 2013 [4], by providing a linear time algorithm for finding the girth of planar graphs.

As for directed graphs, Pettie [15] provided an algorithm with running time $O(mn + n^2 \log\log n)$ for finding the girth of weighted directed graphs, improving a “long standing bound obtained by using Dijkstra’s algorithm and Fibonacci heaps”. Orlin and Sedeno-Noda further reduced this to $O(mn)$ time in a recent work [14]. Computing the girth of a directed graph is similar to the shortest path problem. Indeed, Chang et al. used a single source shortest path algorithm in [4] as a subroutine. However, using a shortest path algorithm is not efficient for our listing problem since our problem computes distance between any two vertices many times. Hence, instead of using a shortest path algorithm as a subroutine, our algorithms will exploit matrices which incrementally and efficiently update the distances and reachability among vertices and edges.

In Sect. 3 we describe algorithm `g-is` (for girth g - *i*nduced *s*ubgraphs), which lists all *induced* subgraphs of G having girth g in $O(n)$ time per solution. In particular, by setting $g = n + 1$ `g-is` can be used to list all *acyclic induced* subgraphs of G with the same complexity. In Sect. 4 we describe algorithm `g-es`, which lists all *edge* subgraphs of G having girth g in $O(n)$ time per solution. Table 1 in Sect. 5 summarizes the contributions.

2 Preliminaries

All graphs and edges considered in this work are directed. A graph is represented as $G = (V(G), E(G))$, where $V(G)$ is the set of vertices and $E(G) \subseteq (V(G) \times V(G))$ the set of edges. We denote as (a, b) the edges whose *tail* is a and *head* is b. When edge direction is not important, we write $\{a, b\}$ to refer to either the directed edge (a, b) or (b, a). $N_G(v)$ represents the set of vertices connected to a vertex v in G by an edge in any direction, i.e., the neighborhood of v, and $N^e_G(v)$ represents the set of edges having v as either tail or head, which we call *edge neighborhood.* If no confusion arises, we will drop the subscripts and use a relaxed notation, e.g. referring to the vertex and edge sets as V and E, or the neighborhoods as $N(v)$ and $N^e(v)$.

An *induced subgraph* of G, given a set of vertices $X \subseteq V(G)$, is the subgraph $G[X] = (X, E[X])$. Here, $E[X] = E(G) \cap (X \times X)$. In other words, the subgraph obtained by removing all vertices in $V(G) \setminus X$ and all edges incident to those vertices from G. An *edge subgraph* of G, given a set of edges $F \subseteq E(G)$, is the subgraph $G[F] = (V[F], F)$, where $V[F]$ is the set of vertices incident to an edge in F, i.e., $V[F] = \{x \mid (x, y) \in F \text{ or } (y, x) \in F\}$.

A cycle is a sequence of distinct vertices $C = \{v_1, \ldots, v_k\}$ such that $(v_i, v_{i+1}) \in E(G)$ for $1 < j < k - 1$, and $(v_k, v_1) \in E(G)$. We say that the cycle C has length k, that is the number of vertices involved in C. The *directed girth*, or simply *girth*, of a graph G is the length of its smallest cycle. A graph is *acyclic* if it contains no cycle. If G is acyclic, its girth is defined to be ∞; in all other cases, the girth of G is at most $|V(G)|$, i.e., the maximum possible length of a cycle.

A basic but fundamental property of the girth is that it is *hereditary*, i.e., any subgraph G' (both induced or edge) of a graph G with girth g has girth at least g, as any cycle shorter than g in G' would be present also in G. Figure 1 shows some examples of induced (b) and edge (c), (d) subgraphs of a graph (a).

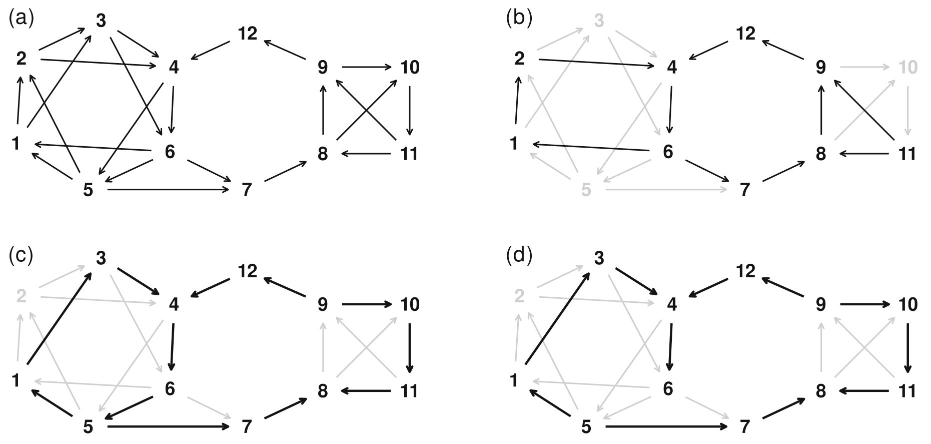

Fig. 1. A graph (a), an induced subgraph with girth 4 (b), an edge subgraph with girth 5 (c), and an acyclic edge subgraph (d). The subgraphs correspond to the vertices and edges in black.

In several cases, properties of graphs with girth g apply also to graphs with larger girth, and so it is common for studies to consider graph with girth $\geq g$ [11], and even to refer to those as simply graph with girth g [18]. We adopt this notation in this work as well, thus we will refer to subgraphs whose girth is *at least* g simply as subgraphs of girth g.

3 Listing Induced Subgraphs with Girth g

Our algorithms enumerate all subgraph with girth g by a simple backtracking procedure that adds vertices to a partial solution $S \subseteq V(G)$ or alternatively removes them from the graph. The vertices removed from the graph are represented by a set X. To give an accurate cost analysis, we will refer to the hypothetical *recursion tree* of its execution, where each recursive call of the algorithm is represented by *recursive node*, or simply *node*, and the nested recursive call inside a recursive node correspond to its children in the recursion tree.

3.1 Basic Algorithm

The basic algorithm `base` is detailed in Algorithm 1. In the beginning, $S = \emptyset$; since a subgraph made by a single vertex is acyclic, it will have girth ∞ and every $v \in V$ will be addible; the algorithm will thus consider all possible subgraphs of a single node (procedure *main*), which will then be further expanded when calling `base`. After a vertex is considered it is conceptually removed from the graph by adding it to X.

The recursive procedure can be seen as a form of binary partition: we identify the set C, called *addible candidate set*, all vertices that may be added to S

Algorithm 1. Enumerating all connected induced subgraphs of girth g in a directed graph $G = (V, E)$

```
1  Procedure main($G = (V(G), E(G)), g$)
2      $X \leftarrow \emptyset$
3      foreach $v \in V(G)$ do
4          base($\emptyset, X, v, g$)
5          $X \leftarrow X \cup \{v\}$

6  Procedure base($S, X, v, g$)
7      $S \leftarrow S \cup \{v\}$
8      Output $S$
9      $C \leftarrow \{x \in V(G) \setminus (S \cup X) \mid G[S \cup \{x\}]$ is connected and has girth $g\}$
10     for $x \in C$ do
11         base($S, X, x, g$)                    // find subgraphs containing $x$
12         $X \leftarrow X \cup \{x\}$                // find subgraphs not containing $x$
13     $S \leftarrow S \setminus \{v\}$; $X \leftarrow X \setminus C$                    // restore S and X
```

without violating the girth or connection constraint. For a vertex $x \in C$, we first consider all the subgraphs of girth g extending S that contain x (Line 11). Then, after these subgraphs have been found, we remove v from the graph by adding it to X and iterate over the next member of C; this corresponds to the "other branch" of the binary partition, when we consider all subgraphs of girth g extending S that do *not* contain v. Thus every cycle of the for loop can be seen as a binary partition step. However, grouping these steps in a single recursive node will aid the analysis of the algorithm. Finally, when all vertices of C have been considered, we output S in Line 8, which corresponds to the choice of *not* adding any $v \in C$; this is also the only solution found in the case $C = \emptyset$.

Induced acyclic subgraphs. As the maximum possible length of a cycle is n, i.e., $|V(G)|$, any graph with girth $\geq n + 1$ is acyclic. Thus we can enumerate all induced acyclic subgraphs by simply using `base` with $g = n + 1$.

Correctness. Proving that each output of `base` is an induced subgraph of girth at least g is trivial, since every vertex added to S has passed the check in Line 9. Is it also straightforward to see that no duplication is possible: all solutions found in sub-calls of Line 11 will contain x, while all solutions found during following cycles of the *for* loop will not contain the same x as it is added to the X set; moreover, every call in Line 11 adds some x to S, thus none of these will output S itself as a solution, which is output in Line 8.

Finally, we only need to show that every subgraph S^* with girth at least g is output by `base`. We prove this by induction: consider as base case for S^* a recursive call in which $S \subseteq S^*$ and $S^* \cap X = \emptyset$. This is trivially true for some S in the beginning of the `main` procedure, in particular the first time a vertex $v \in S^*$ is considered by the *foreach* loop, i.e., with $S = \emptyset$ and $\{v\} \subseteq S^*$ and as no vertex of S^* was previously considered, $S^* \cap X = \emptyset$, thus in the corresponding call of `base` we will have $S = \{v\} \subseteq S^*$ and $X \cap S^* = \emptyset$.

If $S^* \setminus S = \emptyset$, that is if $S = S^*$, then S^* is output in Line 8. Otherwise let $v_1, v_2, \ldots, v_{|C|}$ be the order in which the vertices of C are scanned by the *for* loop, and let v_i be the earliest vertex in the sequence belonging to S^*. When considering v_i, vertices $v_1, \ldots, v_{i-1}$ have been removed from C and not added to S. When the recursive call in Line 11 considers $S' = S \cup \{v_i\}$, all vertices in $S^* \setminus S'$ will still be in C' since any subgraph of S^* also has girth at least g, and adding any vertex from $S^* \setminus S'$ to S' will make a subgraph of S^*. Here, C' is the candidate set for S'. Thus, after the recursive call in Line 11 S' and C' will still respect the inductive hypothesis $S' \subseteq S^*$ and $S^* \setminus S' \subseteq C'$, but $|S' \cap S^*| > |S \cap S^*|$, thus in at most $|S^*|$ such steps there will be a recursive call with $S' = S^*$, that will finally output S^* in Line 8.

Cost analysis. As every recursive call will output a solution in Line 8, the cost per solution is clearly bounded by the cost of a recursive call. This corresponds to the cost of computing C in Line 9. The trivial way to build C is to compute the girth of $G[S \cup \{x\}]$ for each vertex $x \in V(G)$; as the girth can be computed in $O(nm)$ time [14] this yields a total cost of $O(n^2m)$ per solution.

3.2 Improved Algorithm

We considered Algorithm 1, with its complexity of $O(n^2m)$ time per solution, the baseline for the enumeration problem. In the following we show how modify this algorithm to obtain `g-is`, which reduces the cost of `base` by a factor $O(nm)$, obtaining $O(n)$ time per solution. First, consider the following straightforward but fundamental property.

Observation 1. *If, for any vertex $x \notin S$, $\ell(v, S \cup \{x\}) < \ell(v, S)$, then the shortest cycle containing v in $S \cup \{x\}$ must contain x. Here, $\ell(v, S)$ is the length of a shortest cycle containing v in S.*

As this applies to every $v \in S$ and $x \notin S$, this implies a more general property.

Observation 2. *If A is a subgraph of G with girth g, and $A \cup \{x\}$ has girth $g' < g$, the shortest cycle in $A \cup \{x\}$ must involve x.*

Let S_P and C_P be the S and C set computed in the parent call of recursive call R, which correspond to $S_P = S \setminus \{v\}$ and $C_P = \{x \in V(G) \setminus (S_P \cup \{x\}) \mid G[S_P \cup \{x\}]$ is connected and has girth $g\}$. Every addible vertex $x \in C$ for R must either be in C_P or be a neighbor of v, as otherwise the subgraph $G[S \cup \{x\}]$ would either have girth less than g or be disconnected.

We can use this properties to efficiently identify all vertices $x \in C$. C will be made of all the vertices in C_P that still pass the check in Line 9 after adding v to S, and all the vertices in $N(v) \setminus X$ which were not already in C_P: these are only connected to S by v and hence cannot participate in any cycle.

We will also keep in each recursive node a *special distance matrix* for S, that is a matrix M of size $|C| \times |C|$ such that for each pair $x, y \in C$, $M[x, y]$ is equal to the distance between x and y in the induced subgraph $G[S \cup \{x, y\}]$. Clearly, if $M[x, y] + M[y, x] < g$ then $G[S \cup \{x, y\}]$ has a cycle shorter than g. Thanks to M and Observation 2, we can conclude the following.

Lemma 1. *Given C_P the candidate set in the parent recursive call, and the special distance matrix M_P for S_P, Line 9 can be rewritten as follows:*

$$C \leftarrow \{x \in C_P \mid M_P[x, v] + M_P[v, x] \geq g\} \cup (N(v) \setminus (X \cup C_P)) \tag{1}$$

With this technique, C is computed in $O(|C_P|)$ time. After computing C we need to compute M, i.e., the special distance matrix for $S = S_P \cup \{v\}$ which will be passed to the child recursive call. To ease this we will use the M matrix built in the parent recursive call, which we call M_P: we must simply check if the shortest path between two vertices x and y has been improved by adding v to S. In other words, given M_P and C, we can compute M in $O(|M|) = O(|C|^2)$ time as for each $x, y \in C$, $M[x, y] = \min(M_P[x, y], M_P[x, v] + M_P[v, y])$.

As for the first recursive call, with $S = \emptyset$ and $C = V(G)$, M is computed in $O(|M|) = O(|C|^2)$ time, since for each $x, y \in C$, $M[x, y] = 1$ if $(x, y) \in E(G)$, and 0 otherwise. The improved algorithm `g-is` is built by modifying recursive calls of `base` as follows:

- The first recursive call initializes M.
- The C and M are passed to child recursive calls as C_P and M_P.
- The C and M are built using C_P and M_P.
- Line 9 is modified as in Lemma 1.

Cost analysis. Every recursive node of `g-is` will take $O(|C_P| + |C|^2)$ time to compute C and M. While this is trivially bounded by $O(n^2)$, we show that it can be further improved by means of *amortized analysis*: we shift parts of the cost of each recursive node onto other nodes, obtaining a better complexity bound.

Let R be an arbitrary recursive node, which has built the sets S_R, X_R, C_R, and the matrix M_R. Note that R will have exactly $|C_R|$ child recursive nodes and will take $O(|C_P| + |C_R|^2)$ time to execute. However, R will subdivide the $O(|C_R|^2)$ portion of the cost equally among its $|C_R|$ children, for a total of $|C_R|$ each. Every recursive node thus will retain a cost of $O(|C_P|)$ time, and be charged only from its parent of an additional $O(|C_P|)$ time, for a total of $O(|C_P|) = O(n)$ time per each recursive node, i.e., $O(n)$ time per solution found.

Considering the space usage, S, C, and X may be efficiently stored by keeping track of just the difference between the parent and child recursive nodes for an amortized space usage of $O(n)$. As for M, we do not actually need to compute a separate matrix in each recursive node. We simply update the cells of M_P and use M_P as M, accessing only the cells corresponding to indices in C. The depth of the recursion tree, i.e., the number of changes we need to keep track of, is $O(|S|) = O(n)$; the total space usage will thus be $O(|M| \cdot |S|) = O(n^3)$.

As g does not impact the cost, `g-is` can enumerate acyclic subgraphs, i.e., subgraphs with girth at least $n + 1$, in $O(n)$ time per solution as well. Furthermore, distance is meaningless in acyclic subgraphs, as we only care about whether x *can reach* y or not. Each cell of M will thus be updated at most once, for a total space usage of $O(|M|) = O(n^2)$. We can finally state the correctness and complexity of `g-is`.

Theorem 1. `g-is` *lists the induced subgraphs of a graph* G *with girth at least* g *exactly once, using* $O(n)$ *time per solution and* $O(n^3)$ *space.*

Theorem 2. `g-is` *lists the acyclic induced subgraphs of a graph* G *exactly once, using* $O(n)$ *time per solution and* $O(n^2)$ *space.*

3.3 Weighted Case and Non-connected Case

`g-is` is given for unweighted graphs. However, it should be remarked that it is trivially adapted to weighted graphs by simply initializing $M[x, y]$ in the first recursive call to the weight of the edge (x, y), rather than 1, as long as $g > 0$. The approach works in the presence of negative edges and even negative cycles, as a negative cycle can never be added to S since it would cause $g < 0$. `g-is` can also be trivially modified to drop the connectivity constraint, by simply setting $C = V(G)$ in the first recursive call, so that every vertex can be immediately considered for addition (we can then also ignore the addition of vertices in $N(v)$ to C, see Lemma 1). Similar trivial adaptations are possible for all the algorithms proposed in the remainder of the paper.

4 Listing Edge Subgraphs with Girth *g*

In this section we describe an algorithm for listing all *edge* subgraphs of girth at least g, The algorithm, which we call `g-es`, is detailed in Algorithm 2. The structure of `g-es` is in essence that of `g-is`, but with two key differences. Firstly, the solution S, the set of addible candidates C, and excluded elements X are sets of *edges* rather than vertices. Secondly, the order in which candidate edges are selected in `g-es` will play an important role in the complexity of `g-es`. The baseline algorithm, obtained by trivially adapting Algorithm 1 for edge subgraphs, has a complexity of $O(m^2 n)$ time per solution. We will show that `g-es` will improve this bound by a factor $O(m^2)$, obtaining $O(n)$ time per solution.

Like in `g-is`, at any time we consider the current solution as a set of edges $S \subseteq E(G)$, corresponding to a subgraph with girth g, a set X of excluded edges (i.e., conceptually removed from the graph), and the set $C \subseteq E(G) \setminus (S \cup X)$ of edges that are addible to S without violating the girth constraint. In addition, we will subdivide C into C_{in} and C_{ext}: let S_N be the set of vertices incident to edges in S, $\forall e = \{x, y\} \in C$, $e \in C_{in}$ if $\{x, y\} \subseteq S_N$, and $e \in C_{ext}$ otherwise. Again, we find all solutions in a binary partition fashion by selecting an edge $e \in C$, and first considering all subgraphs with contain $S \cup \{e\}$, then removing e from C and considering those that contain S but not e.

We call this algorithm `g-es`, and we can reconstruct its structure by simple modifications of `g-is`: in particular, each cycle of the *for* loop in Line 10 in Algorithm 1 considers an edge $e \in C_{in}$ rather than a vertex. Furthermore, the updated C (Line 9) should be computed as $C \leftarrow \{e' \in E(G) \setminus (S \cup X) \mid$ and $G' = (V[S \cup \{e'\}], S \cup \{e'\})$ is connected has girth $g\}$. Finally, `g-es` will select e from C_{ext} *only if* $C_{in} = \emptyset$. For brevity, we omit the correctness proof which consists in simply retracing that of `g-is`.

Again, we employ a *special distance matrix* M for S; let C_N be the set of vertices incident to at least one edge in C: in this case M will have size $|C_N| \times |C_N|$, and for each pair $x, y \in C_N$, $M[x, y]$ is equal to the distance between x and y in the edge subgraph $G' = (V[S], S)$. For two edges $e_1 = (x, y)$ and $e_2 = (w, z)$ in C, if there is a cycle shorter than g in $G'' = (V[S \cup \{e_1, e_2\}], S \cup \{e_1, e_2\})$, then we will have $M[y, w] + M[z, x] + 2 < g$, since any cycle involving e_1 and e_2 must traverse the vertices y, w, z, and x in this order. After adding $e = \{a, b\}$ to S, the edges in $N^e(a) \cup N^e(b)$ but not in X may enter C, which can thus be computed similarly to how done in Lemma 1 for the induced case, i.e.

$$C \leftarrow \{e' = \{c, d\} \in C_P \cup (N^e(a) \cup N^e(b)) \setminus X \mid M[b, c] + M[a, d] + 2 \geq g\}, \quad (2)$$

where the 2 is added to account using the edges e and e' and can be replaced by their weight for the weighted case. The values of M can also be similarly updated, as after adding $e = \{a, b\} \in C$ to S, we have that $M[y, w]$, i.e., the distance "from" e_1 to e_2 in $G' = (V[S \cup \{e\}, S \cup \{e\})$, was either improved by using e or is unchanged. That is $M[y, w] = \min(M[y, w], M_P[y, a] + M_P[b, z] + 1)$, where the 1 is added to account for using e. Note that we replaced by the weight of e when weighted case. Thus, `g-es` will also follow the structure in Algorithm 1, modifying recursive calls of `base` as follows:

- The first recursive call initializes M.
- The sets C_{in}, C_{ext}, C_N, S_N and M are passed to child recursive calls.
- C, C_N, S_N and M are updated using those passed from the father recursive call.
- The candidates in C_{ext} will be selected only after C_{in} is empty.
- Line 10 is modified as in Eq. (2).

Cost analysis. By implementing the updates in Line 10 similarly to how done in `g-is`, and performing the same amortized analysis, one could easily find that `g-es` has a complexity of $O(m)$ time per solution, which is a factor $O(mn)$ faster than the baseline. In the following, however, we will further reduce the cost of Line 10 and obtain $O(n)$ time per solution.

In particular, let us focus on the update of the C_{in} and C_{ext} sets. When `g-es` selects $e \in C_{ext}$, updating the sets can trivially be done in $O(m)$ time by testing each edge $f \in E(G) \setminus (S \cup X)$ with M as in Eq. 2. However, this can be simplified by means of the following:

Lemma 2. *Let $e = \{a, b\} \in C_{ext}$ be the edge selected and added to S by* `g-es`*, with $C_{in} = \emptyset$. Without loss of generality let $a \in S_N$ and $b \notin S_N$. Then*

- *The updated C_{in} is contained $N^e(b) \setminus (S \cup X)$.*
- *The updated C_{ext} is contained in $C_{ext} \cup N^e(b) \setminus (S \cup X)$.*
- *Both C_{in} and C_{ext} can be updated in $O(\Delta)$ time.*

Proof. Since b is the only new vertex in S_N, the new edges in C_{in} and C_{ext} must be adjacent to b. Any new edge in C_{in} must be removed from C_{ext}. C_{in} and C_{ext} can be computed by scanning $N^e(b)$ and testing each edge $\{b, x\}$ not in S or X in constant time using M, adding the edges that pass the girth test to C_{in} if $x \in S_N$ and to C_{ext} otherwise. This takes $O(\Delta)$ time. □

Algorithm 2. g-es: Enumerating all connected edge subgraphs of girth g in a directed graph $G = (V, E)$

```
1  Procedure main($G = (V(G), E(G)), g$)
2      $X \leftarrow \emptyset$
3      foreach $e = \{x, y\} \in E(G)$ do
4          g-es($\emptyset, X, v, g$)
5          $X \leftarrow X \cup \{v\}$
6  Procedure g-es($S, C_{in}, C_{ext}, C_N, S_N, M, X, e, g$)
       // let $e = \{a, b\}$
7      $S \leftarrow S \cup \{e\}$
8      $S_N \leftarrow S_N \cup \{a, b\}$
9      Output $S$
10     Update $C_N, C_{in}, C_{ext}, M$ for the new $S$ and $X$
11     for $f \in C_{in}$ do
12         g-es($S, C_{in}, C_{ext}, C_N, S_N, M, X, f, g$)     // subgraphs containing $f$
13         $X \leftarrow X \cup \{f\}$                         // subgraphs not containing $f$
14     for $f \in C_{ext}$ do
15         g-es($S, C_{in}, C_{ext}, C_N, S_N, M, X, f, g$)     // subgraphs containing $f$
16         $X \leftarrow X \cup \{f\}$                         // subgraphs not containing $f$
17     $S \leftarrow S \setminus \{v\}$; $X \leftarrow X \setminus C$                  // restore S and X
18     Restore $C_N, C_{in}, C_{ext}, M$ for the restored $S$ and $X$
```

Note that g-es only selects e from C_{ext} once C_{in} is empty, and otherwise it will select it from C_{in}. Selecting e from C_{in} always decreases $|C_{in}|$ by at least 1: indeed no new edge may enter C_{in} since S_N is unchanged, but e itself is removed. When C_{in} is empty and we select e from C_{ext}, $|C_{in}|$ may become at most Δ (Lemma 2). We can thus state that

Lemma 3. *At any time in* g-es*, $|C_{in}| \leq \Delta$.*

When g-es selects the edge e from C_{in}, thanks to Lemma 3 we can also update C_{in} and C_{ext} faster than in $O(m)$ time:

Lemma 4. *Let $e = \{a, b\} \in C_{in}$ be the edge selected and added to S by* g-es*. Then the updated C_{in} is included in $C_{in} \setminus \{e\}$ and can be computed in $O(\Delta)$, and C_{ext} is unchanged.*

Proof. As S_N is unchanged, no edge enters C_{in}, but e is removed. Whether each edge remains in C_{in} can be tested in constant time using M, which takes $O(\Delta)$ time as $|C_{in}| \leq \Delta$ by Lemma 3. Finally, as every edge in C_{ext} still exactly one extreme in S_N, it may not participate in any cycle in $G(V[S_N], S)$, and since S_N is unchanged no edge is either removed from or added to C_{ext}. □

We can now proceed to give the complexity g-es: consider any recursive call P, with its sets S_P, X_P, C_{inP}, C_{extP}, C_{NP} and the matrix M_P as computed in

Line 10, and R, a child recursive call of P (performed in either Line 12 or 15) with its sets S_R, X_R, C_{inR}, C_{extR}, C_{NR} and the matrix M_R.

Thanks to Lemmas 2 and 4, we can update C_{inP} and C_{extP} to obtain C_{inR} and C_{extR} in $O(\Delta)$ time. Furthermore, C_{NR} can be obtained in constant time, and using M_P and C_{NR} we can update M_P to obtain M_R in $O(|C_{NR}|^2)$ time. The total cost of Line 10 will thus be $O(\Delta + |C_{NR}|^2)$.

However, we have that for each edge in C_{inR} and C_{extR} there are two vertices in C_{NR}, which means $|C_{NR}| \leq 2(|C_{inR}| + |C_{extR}|)$. As R has $|C_{inR}| + |C_{extR}|$ children recursive calls, and $|C_{NR}| = O(|C_{inR}| + |C_{extR}|)$, we can give the same amortized analysis as for `g-is`: R will subdivide equally among its children the $O(|C_{NR}|^2)$ time component of its cost, for a total of $O(|C_{NR}|) = O(n)$ for each child. Each recursive node will thus maintain the $O(\Delta)$ time component of the cost, and receive an additional $O(n)$ time component from its parent call, for a total cost of $O(n)$ time per recursive node, i.e., $O(n)$ time per solution.

The space complexity of `g-es`, similarly, is dominated by the space needed to store S, C_{in}, C_{ext} and X, which can be stored in amortized $O(m)$ space (by keeping track of the differences between parent and children recursive calls), and C_N and S_N, which can similarly be stored in $O(n)$ space. Finally, M has $O(n^2)$ cells, and for each cell we must keep track of at most n changes. Indeed, while the depth of the recursion is bounded by m, each value $M[i, j]$ corresponds to a distance between two vertices i and j, which is bounded by n; as the distance is only updated when it is reduced, and each reduction is of at least 1, there can be no more than n updates, which lead to a total space usage of $O(n^3)$[1].

As for acyclic edge subgraphs, there are only two possible values for $M[i, j]$: *can reach* and *cannot reach*. As we only need to keep track of one update, the space usage will be $O(n^2)$. We can finally state the cost of `g-es`:

Theorem 3. *Given a graph $G = (V, E)$,* `g-es` *lists the edge subgraphs of G with girth at least g exactly once, in time $O(n)$ per solution and space $O(n^3)$.*

Theorem 4. *Given a graph $G = (V, E)$,* `g-es` *lists the acyclic edge subgraphs of G, in time $O(n)$ per solution and space $O(n^2)$.*

5 Delay and Final Remarks

While the cost per solution of `g-is` and `g-es` is $O(n)$, their *delay*, i.e., the maximum elapsed time between the output of a solution and the following one, is higher, unless we employ additional techniques. By outputting the solution at the beginning of every recursive call (e.g., moving Line 8 of Algorithm 1 to the top), a solution will be output whenever a recursive call is performed. In this case the delay will be bounded by the cost of updating M in Line 9 in Algorithm 1 for `g-is` (using Lemma 1), and Line 10 in Algorithm 2 for `g-es`, i.e., $O(n^2)$.

[1] This is different in the weighted case, in which distances can be reduced by less than 1, and will thus require using $O(n^2 m)$ space.

However, we can reduce the delay by employing the *output queue* and *alternative output* techniques [19]: Let X be an arbitrary recursion node, T^* be an upper bound on the cost of X, and $\bar{T}$ an upper bound for the ratio

$$\text{(cost of processing the subtree of } X)/\text{(solutions found in the subtree)} \qquad (3)$$

Table 1. Time and space complexity of the proposed enumeration algorithms, including the *output queue* technique preprocessing cost.

`subgraph type`	`algorithm`	`delay`	`pre-processing`	`space usage`
induced with girth g	`g-is`	$O(n)$	$O(n^3)$	$O(n^3)$
induced acyclic	`g-is`	$O(n)$	$O(n^3)$	$O(n^2)$
edge with girth g	`g-es`	$O(n)$	$O(n^3)$	$O(n^3)$
edge acyclic	`g-es`	$O(n)$	$O(n^3)$	$O(n^2)$

To reduce the delay, we will need to use a buffer which stores $\lceil 2 \cdot T^*/\bar{T} \rceil + 1$ solutions. First, we fill the buffer until it is full, then we will out a solution every $O(\bar{T})$ time, obtaining $O(\bar{T})$ delay.

By means of our amortized cost analysis (see Sect. 3) we have that $\bar{T}$ corresponds exactly to the cost per solution, that is $O(n)$ for both `g-is` and `g-es`.

Thus we will have $T^* = O(n^2)$ and $\bar{T} = O(n)$, meaning that we will obtain delay $O(n)$, at the cost of storing $\Theta(n)$ solutions. As a solution of `g-is` is defined by a set of vertices, this translates to a space usage of $O(n^2)$ and a delay of $O(n)$.

As for `g-es`, we would need to store solutions corresponding to sets of edges, which have size $O(m)$ and take $O(m)$ to output. We address this problem with the alternative output technique: this consists in performing the output of a solution as the *first* operation in each recursive node of *even* depth, and as the *last* operation in each recursive node of *odd* depth.

Thanks to this structure, consecutive outputs of the algorithm are performed by recursive nodes at distance at most 3 in the recursion tree (see Fig. 3 in [19]). As each recursive call outputs a solution that differs by 1 edge from those output by its parent and children, consecutively output solutions will differ by at most 3 edges. We can thus output each solution by giving only the difference with the last output solution, which takes constant space (and time), thus the buffer size will take only $O(m)$ space for the first solution, and $O(n)$ space for the subsequent n ones, for a total cost of $O(n)$ delay and $O(m)$ space usage.

In both cases, the space required by the solution buffer does not increase the $O(n^2)$ space usage of `g-is` and `g-es`. However, the output queue technique will add a pre-processing time, that is the time required to fill the buffer: as without the output queue technique the algorithm guarantee a delay of $O(n^2)$ time, the time required to fill a buffer of $\Theta(n)$ solution, that is $O(n^3)$.

Table 1 summarizes the performances of the proposed algorithms.

References

1. Berger, B., Shor, P.W.: Approximation algorithms for the maximum acyclic subgraph problem. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 236–243 (1990)
2. Borodin, O.V., Kim, S.-J., Kostochka, A.V., West, D.B.: Homomorphisms from sparse graphs with large girth. J. Comb. Theory Ser. B **90**(1), 147–159 (2004)
3. Chandran, L.S., Subramanian, C.R.: Girth and treewidth. J. Comb. Theory Ser. B **93**(1), 23–32 (2005)
4. Chang, H.-C., Lu, H.-I.: Computing the girth of a planar graph in linear time. SIAM J. Comput. **42**(3), 1077–1094 (2013)
5. Conte, A., Grossi, R., Marino, A., Rizzi, R.: Listing acyclic orientations of graphs with single and multiple sources. In: Kranakis, E., Navarro, G., Chávez, E. (eds.) LATIN 2016. LNCS, vol. 9644, pp. 319–333. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49529-2_24
6. Cook, R.J.: Chromatic number and girth. Periodica Mathematica Hungarica **6**(1), 103–107 (1975)
7. Diestel, R.: Graph Theory. Graduate Texts in Mathematics, 4th edn. Springer, Heidelberg (2017)
8. Djidjev, H.: Computing the girth of a planar graph. In: 27th International Colloquium on Automata, Languages and Programming, pp. 821–831 (2000)
9. Galluccio, A., Goddyn, L.A., Hell, P.: High-girth graphs avoiding a minor are nearly bipartite. J. Comb. Theory Ser. B **83**(1), 1–14 (2001)
10. Grötschel, M., Jünger, M., Reinelt, G.: On the acyclic subgraph polytope. Math. Prog. **33**(1), 28–42 (1985)
11. Hayes, T.P.: Randomly coloring graphs of girth at least five. In: ACM Symposium on Theory of Computing, pp. 269–278. ACM (2003)
12. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM J. Comput. **7**(4), 413–423 (1978)
13. Johnson, D.S., Yannakakis, M., Papadimitriou, C.H.: On generating all maximal independent sets. Inf. Process. Lett. **27**(3), 119–123 (1988)
14. Orlin, J.B., Sedeno-Noda, A.: An o(nm) time algorithm for finding the min length directed cycle in a graph. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 1866–1879 (2017)
15. Pettie, S.: A new approach to all-pairs shortest paths on real-weighted graphs. Theor. Comput. Sci. **312**(1), 47–74 (2004)
16. Raman, V., Saurabh, S.: Short cycles make W-hard problems hard: FPT algorithms for W-hard problems in graphs with no short cycles. Algorithmica **52**(2), 203–225 (2008)
17. Squire, M.B.: Generating the acyclic orientations of a graph. J. Algorithms **26**(2), 275–290 (1998)
18. Thomassen, C.: 3-list-coloring planar graphs of girth 5. J. Comb. Theory Ser. B **64**(1), 101–107 (1995)
19. Uno, T.: Two general methods to reduce delay and change of enumeration algorithms. NII Technical Report NII-2003-004E, Tokyo, Japan (2003)