Enumerating Connected Induced Subgraphs: Improved Delay and Experimental Comparison

Christian Komusiewicz and Frank Sommer

Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany
{komusiewicz,fsommer}@informatik.uni-marburg.de

Abstract. We consider the problem of enumerating all connected induced subgraphs of order k in an undirected graph $G = (V, E)$. Our main results are two enumeration algorithms with a delay of $\mathcal{O}(k^2 \Delta)$ where Δ is the maximum degree in the input graph. This improves upon a previous delay bound [Elbassioni, JGAA 2015] for this problem. In addition, we give improved worst-case running time bounds and delay bounds for several known algorithms and perform an experimental comparison of these algorithms for $k \leq 10$ and $k \geq |V| - 3$.

1 Introduction

We study algorithms for the following fundamental graph problem.

Connected Induced Subgraph Enumeration (CISE)
Input: An undirected graph $G = (V, E)$ and an integer k.
Task: Enumerate all connected induced subgraphs of order k.

We call a connected subgraph of order k a *solution* in the following. The enumeration of connected subgraphs is important in many applications, such as the identification of network motifs (statistically overrepresented induced subgraphs of small size). A straightforward algorithm to find such motifs is to enumerate all connected induced subgraphs and to count how often each subgraph of order k occurs [6, 14]. A further application arises when semantic web data is searched using only keywords instead of structured queries [5]. Finally, many fixed-cardinality optimization problems can be solved by an algorithm whose first step is to enumerate connected induced subgraphs of order k [8]. This algorithm can solve for example Connected Densest-k-Subgraph, the problem of finding a connected subgraph of order k with a maximum number of edges. Experiments showed that enumeration-based algorithms can be competitive with other algorithmic approaches [9].

At first sight, providing any nontrivial upper bounds on the running time of CISE seems hopeless: As evidenced by a clique on n vertices, graphs may have up to $\binom{n}{k}$ CISE solutions. Even very sparse graphs may have $\binom{n-1}{k-1}$ CISE solutions as evidenced by a star graph with $n - 1$ leaves. It is maybe due to these lower bounds that, despite its importance, CISE has not received too much attention from the viewpoint of worst-case running time analysis.

To appear in *Proceedings of the 45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '19)*, Nový Smokovec, Slovakia, January 2019. © Springer.

One way to achieve relevant running time bounds is to consider degree-bounded graphs where the number of solutions is much smaller than in general.

Lemma 1 ([3, Equation 7]). *Let G be a graph with maximum degree Δ. Then the number of connected induced subgraphs of order k that contain some vertex v is at most $(e(\Delta - 1))^{(k-1)}$. Hence, the overall number of connected induced subgraphs of order k in G is $\mathcal{O}((e(\Delta - 1))^{(k-1)} \cdot (n/k))$.*

This observation can be exploited to obtain an algorithm for CISE that runs in $\mathcal{O}((e(\Delta - 1))^{(k-1)} \cdot (\Delta + k) \cdot (n/k))$ time [8].

A second approach to provide nontrivial running time bounds is to prove upper bounds on the *delay* of the enumeration. The delay is the maximal time that the algorithm spends between the output of consecutive solutions. The *reverse search* framework is a general paradigm for enumeration algorithms with bounded delay. The basic idea is to construct a tree where each node represents a unique element of the enumeration process. By traversing this tree from the root, each element is enumerated exactly once. By using reverse search, one can enumerate all induced subgraphs of order *at most* k with polynomial delay [1]. When we are interested only in solutions of order *exactly* k [4], this algorithm is *not* output polynomial, that is, the running time is not bounded by a polynomial in the input and output size. Hence, it does not achieve polynomial delay either. A different reverse search algorithm, however, achieves delay $\mathcal{O}(k \min(n - k, k\Delta)(k(\Delta + \log k) + \log n))$ [4].

Thus, k and Δ appear to be central parameters governing the complexity of CISE. Motivated by this observation, we aim to make further progress at exploiting small values of Δ and k.

Related Work. Most known CISE algorithms follow the same strategy: starting from an initial vertex set $S := \{v\}$ for some vertex v, build successively larger connected induced subgraphs $G[S]$ until an order-k subgraph is found. Wernicke [13] describes a procedure following this paradigm, which we refer to as *Simple*. The idea is to branch into the different possibilities to add one vertex u from $N(S)$. Another popular enumeration algorithm is *Kavosh* [6] which also considers adding vertices of $N(S)$ but creates one branch for each subset of $N(S)$ that has size at most $k - |S|$.

A slightly different strategy is to first pick a vertex p of the current set S whose neighbors are added in the next step and then branch on the up to $(\Delta - 1)$ possibilities for adding a neighbor of this vertex. The vertex p is called the *active* vertex of the enumeration. The corresponding algorithm, which we call *Pivot*, has a worst-case running time of $\mathcal{O}((4(\Delta - 1))^k \cdot (\Delta + k) \cdot n)$ [7]. A further variant of *Pivot* achieves the running time of $\mathcal{O}(e((\Delta - 1))^{(k-1)} \cdot (\Delta + k) \cdot n/k)$ mentioned above [8]. This variant, which we call *Exgen*, generates *exhaustively* all subsets S' of $N(p) \setminus S$ of size at most $k - |S|$ and creates for each such set S' one branch in which S' is added to S. The final variant that we consider is *BDDE* [11]. For a fixed vertex v, *BDDE* enumerates the connected subgraphs containing v for increasing subgraph orders. The main idea is to use two functions, one to discover new graph edges and one to copy siblings in the enumeration tree.

The above-mentioned algorithms with polynomial delay [4] work differently. They use reverse search and, more generally, the supergraph method [1]. There, for a given graph G and parameter k, the supergraph $\mathcal{G}$ contains a node for each CISE solution in G. Furthermore, two nodes in $\mathcal{G}$ are connected if and only if the corresponding connected subgraphs differ in exactly one vertex. Let $|\mathcal{G}|$ denote the number of vertices in $\mathcal{G}$, that is, the number of CISE solutions. The basic idea is to explore the supergraph $\mathcal{G}$ efficiently. The first variant, which we refer to as *RwD* (*Reverse Search with Dictionary*) has a delay of $\mathcal{O}(k \min(n-k, k\Delta)(k(\Delta + \log k) + \log n))$ and requires $\mathcal{O}(n+m+k|\mathcal{G}|)$ space where m is the number of edges in the input graph G. The second variant, which we refer to as *RwP* (*Reverse Search with Predecessor*), has a delay of $\mathcal{O}((k \min(n-k, k\Delta))^2(\Delta + \log k))$ and requires $\mathcal{O}(n+m)$ space [4]. Hence, algorithm *RwD* admits a better delay but requires exponential space, since $\mathcal{G}$ may grow exponentially with the size of G.

Our Results. We show how to adapt *Simple* and *Pivot* in such a way that the worst-case delay between the output of two solutions is $\mathcal{O}(k^2\Delta)$ and the algorithms requires $\mathcal{O}(n+m)$ space. This improves over the previous best delay bound of *RwD* [4] while requiring only linear space. As a side result, we show that these variants of *Simple* and *Pivot* achieve an overall running time of $\mathcal{O}(e((\Delta-1))^{(k-1)} \cdot (\Delta+k) \cdot n/k)$ and $\mathcal{O}((e(\Delta-1))^{k-1} \cdot \Delta \cdot n)$, respectively. For *Simple* this is the first running time bound, for *Pivot*, this is a substantial improvement over the previous running time bound.

Finally, we compare these algorithms experimentally with implementations of *Kavosh* [6], *Exgen* [8], and *BDDE* [11]. For $k \leq 10$, we observe that *RwD* and *RwP* are significantly slower than the other algorithms. The *Simple* algorithm is faster than *RwD* and *RwP* but substantially slower than the other algorithms. *Kavosh* [6] is the fastest with *Pivot* being surprisingly competitive. For k close to the order of the largest connected component, we observe that our adaptions are necessary to solve these instances. Again, *RwD* and *RwP* are slower than the other algorithms and again, *Kavosh* is the fastest algorithm with *Simple* being second-best but not competitive with *Kavosh*.

Due to lack of space, several proofs are deferred to a long version of the article.

2 Preliminaries and Main Algorithm

Graph Notation. We consider undirected simple graphs $G = (V, E)$. The *order* $n := |V|$ denotes the number of vertices in G and $m := |E|$ denotes the number of edges in G. For a vertex v, $N(v) := \{u \mid \{u, v\} \in E\}$ denotes the *open neighborhood* of v, and $N[v] := N(v) \cup \{v\}$ denotes the *closed neighborhood* of v. For a vertex set $W \subseteq V$, $N(W) := \bigcup_{v \in W} N(v) \setminus W$ denotes the *open neighborhood of* W and $N[W] := N(W) \cup W$ denotes the *closed neighborhood of* W. The graph $G[W] := (W, \{\{u, v\} \in E \mid u, v \in W\})$ is the *subgraph induced by* W. For a set W the graph $G - W := G[V \setminus W]$ is the subgraph of G obtained by deleting the vertices of W. A connected component of G is a maximal subgraph where any two vertices are connected to each other by paths.

Algorithm 1 The main loop for calling the enumeration algorithms; *Enum-Algo* can be any of *Simple*, *Pivot*, *Exgen*, *Kavosh*, and *BDDE*.

```
1: procedure Enumerate($G = (V, E)$)
2:     while $|V(G)| \geq k$ do
3:         choose vertex $v$ from $V(G)$
4:         enumerate all CISE solutions containing $v$ with Enum-Algo
5:         remove $v$ from $G$
```

Enumeration Trees and the Main Algorithm Loop. With the exception of *RwD* and *RwP*, the enumeration algorithms use a search tree method which is called from a main loop whose pseudo code is given in Algorithm 1. Different algorithms, for example *Simple* or *Pivot*, can be used as *Enum-Algo* in Line 4 in Algorithm 1. For each vertex in the graph, Algorithm 1 creates a unique *enumeration tree*. In other words, Algorithm 1 produces a forest consisting of $|V|$ enumeration trees. To avoid confusion, we refer to the vertices of the enumeration trees as *nodes*. Each node represents a connected subgraph $G[S]$ of order at most k. Roughly speaking, a node N is a child of another node M if the subgraph corresponding to M is a subgraph of the subgraph corresponding to N. The exact definition of child depends on the choice of *Enum-Algo*. A *leaf* is a node without any children. Further, a leaf is *interesting* if S has size k; otherwise it is *boring*. A node leads to an interesting leaf, if at least one of its descendants is an interesting leaf.

In the main algorithm loop, we enumerate for each vertex of the input graph all *CISE* solutions containing the vertex v by calling the respective enumeration procedures; the first call of the enumeration procedure is the *root* of the enumeration tree and it represents the connected subgraph $G[\{v\}]$. After enumerating all solutions containing v, the vertex v is removed from the graph.

Cleaning the Graph. The removal of v may create connected components of order less than k. If *Enumerate* chooses all vertices from such connected components, then we will not achieve the claimed delays. Hence, we show how to remove these connected components quickly.

Lemma 2. *Let G be a graph such that each connected component has order at least k and let v be an arbitrary vertex of G. In $\mathcal{O}(k^2 \Delta)$ time we can delete every vertex of $G - \{v\}$ that is in a connected component of order less than k.*

3 Polynomial Delay with Simple

We now adapt *Simple* to obtain a polynomial delay algorithm; the pseudo code is shown in Algorithm 2. In *Simple*, we start with a single vertex v and find successively larger connected subgraphs containing v. The vertex set of a subgraph set is denoted by P. Further, the set X, called *extension set*, contains those neighbors of P which can be added to P to enlarge this subgraph. When putting u in the set P, we remove u from X and add to X each neighbor of u which is not in $N[P]$. Lines 10 and 11 of Algorithm 2 are not part of the plain version of

Algorithm 2 The *Simple* algorithm; the initial call is *Simple*$(\{v\}, N(v))$.

```
1: procedure SIMPLE($P, X$)
2:     if $|P| = k$ then
3:         output $P$
4:         return
5:     while $X \neq \emptyset$ do
6:         $u :=$ choose arbitrary vertex from $X$
7:         delete $u$ from $X$                    ▷ The current set $P$ will be extended
8:         $X' := X \cup (N(u) \setminus N[P])$
9:         Simple($P \cup \{u\}, X'$)
10:        if output of Simple($P \cup \{u\}, X'$) was empty then
11:            return                  ▷ Stop recursion if no new solution found
12:    return
```

Simple [13]. Without these two lines *Simple* is not a polynomial delay algorithm for CISE.

We now present a pruning rule (Lines 10 and 11 of Algorithm 2) that will establish polynomial delay. Consider a path $T_1, \ldots, T_i$ from the root T_1 to a node T_i of an enumeration tree. We denote the subgraph set of a node T_i by P_i and its extension set by X_i. To avoid some unnecessary recursions, we check after each recursive call of *Simple* in node T_i whether this call reported a new solution. If not, we return in T_i to its parent T_{i-1}. First, we prove that this pruning rule is correct. Recall that a leaf T_j is called interesting if the corresponding subgraph set P_j is a solution for CISE and that T_j is called boring otherwise.

Lemma 3. *If the output of a recursive call of Simple in node T_i is empty, then no subsequent recursive call of Simple in node T_i leads to an interesting leaf.*

Now we prove that *Simple* achieves a polynomial delay. To this end, we present a new data structure to store the extension set during the algorithm. In the following, we denote by p_i the vertex which was added to the subgraph set P_i when T_i is created. In other words, if T_{i-1} is the parent of T_i, then $p_i \in P_i \setminus P_{i-1}$. First, we prove that for a node T_i in the enumeration tree we need $\mathcal{O}(\Delta)$ time to either compute its next child T_{i+1} or to restore its parent T_{i-1}.

Lemma 4. *Simple can be implemented in such a way that for every node T_i of the enumeration tree, we need $\mathcal{O}(\Delta)$ time to either compute the next child T_{i+1} or to restore the parent T_{i-1} and that the overall space needed is $\mathcal{O}(n+m)$.*

Proof. We describe the data structures that we use to fulfill the running time and space bounds of the lemma. To check whether a vertex is in some extension set, we color some vertices of with $k+1$ colors $c_0, \ldots, c_k$ as follows. For a node T_i, we call the *exclusive neighbors of* p_i the vertices which are in $N[P_i] \setminus N[P_{i-1}]$ where T_{i-1} is the parent of T_i. These are exactly the vertices that are added to X_{i-1} in Line 7 of Algorithm 2 to construct the set X_i for the node T_i. Throughout the algorithm we maintain the following invariant: The vertex p_1 has color c_0. A

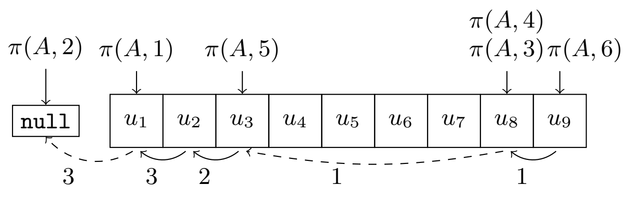

Fig. 1. An example for the pointer movement: Pointer $\pi(A, 6)$ points to u_9, an exclusive neighbor of p_6. Before adding u_9 to the subgraph set P_6, we move pointer $\pi(A, 6)$ to the left to p_8, an exclusive neighbor of vertex $\pi(A, 3)$. Hence, we move $\pi(A, 6)$ to the position of pointer $\pi(A, 5)$, since T_5 is the parent of T_6. Next, we create a child of T_6 by adding u_9 to the subgraph set P_6. The next time we are in node T_6, we move $\pi(A, 6)$ one to the left to vertex u_2 create a child of T_6 by adding u_3 to P_6. After returning from this child, we move $\pi(A, 6)$ to vertex u_1 which is an exclusive neighbor of vertex p_1. Hence, we move $\pi(A, 6)$ to the position of $\pi(A, 2)$, since T_2 is the parent of T_3. Afterwards, we create a child by adding u_2 to P_6. The next time we come back to node T_6, we delete pointer $\pi(A, 6)$, since $\pi(A, 6)$ points to `null`, and return to the parent T_5 of node T_6.

vertex has color c_i, $i \geq 1$, if and only if it is an exclusive neighbor of p_i. In a nutshell, the colors $c_0, \ldots, c_j$ represent the vertices in $N[P_j]$. It is necessary to use $k + 1$ different colors to determine in which node a vertex was added to the extension set. Note that every vertex may have at most one color.

The extension sets of all nodes on the path from the root T_1 to an enumeration tree node T_i are represented by an array A of length $k\varDelta$ with up to k pointers pointing to positions of A. There is one pointer $\pi(A, i)$ corresponding to T_i and one pointer $\pi(A, j)$ for each ancestor T_j of T_i. An entry of A is either empty or contains a pointer to a vertex of the extension set X_i. New vertices for the extension set replace empty entries in the back. Pointer $\pi(A, i)$ points to the vertex x in the extension set X_i which will be added to P_T in the *next* recursive call of *Simple* in node T_i. If at node T_i already all children of T_i have been created, then $\pi(A, i)$ points to `null`. Hence, we may check in constant time whether T_i has further children and return to the parent of T_i if this is not the case.

In addition to A, we use two further simple data structures: The subgraph set P_i at a node T_i is implemented as stack Q that is modified in the course of the algorithm with the top element of the stack being p_i. Also, for each node T_i, we create a list L_i of its exclusive neighbors. This list is necessary to undo some later operations. We now describe how these data structures are maintained throughout the traversal of the enumeration tree.

Initialization. At the root T_1 of the enumeration tree, we initialize A as follows: add all neighbors of the start vertex $p_1 := v$ to A, set pointer $\pi(A, 1)$ to the last non-empty position in A. Hence, the initial extension set is represented by all vertices from the first vertex in A to the initial position of pointer $\pi(A, 1)$. These are precisely the vertices of the exclusive neighborhood of v. The stack Q consists of the vertex v and L_1 contains all neighbors of v.

Creation of new children. As discussed above, a node T_i has a further child T_{i+1} if it points to an index containing some vertex x. We create child T_{i+1} as follows:

1. move the pointer $\pi(A, i)$ to the left,
2. check whether x is an exclusive neighbor of p_i, and remove x from A if this is the case, and
3. create the child T_{i+1} with $p_{i+1} = x$ and enter the recursive call for T_{i+1}.

We now specify how to *move the pointer* $\pi(A, i)$ *to the left* when it currently points to vertex x of color c_ℓ. Note that if x is an exclusive neighbor of p_i, we have $i = \ell$. If x is contained in the first entry of A, then redirect $\pi(A, i)$ to `null`. Otherwise, decrease the position of $\pi(A, i)$ by one. If $\pi(A, i)$ now points to a position containing a vertex y of color c_j such that $\pi(A, j)$ also points to y, then move $\pi(A, i)$ to the position that $\pi(A, \ell - 1)$ points to. Observe that if $j = \ell - 1$ this means that the pointer does not move in the second step.

We now describe how the algorithm creates a child T_{i+1} of T_i after fixing $p_{i+1} := x$ as described above. If node T_{i+1} is an interesting leaf, that is, if $i = k - 1$, we output $P_{i+1} \cup \{x\}$ and return to node T_i. Otherwise, we add vertex x to the stack Q representing the subgraph set and create an initially empty list L_{i+1}. Then we update A so that it represents X_{i+1}. For each neighbor u of x, check if u has some color c_j. If this is not the case, then color u with color c_{i+1} and add u to L_{i+1}. Now store the vertices of L_{i+1} in the left-most non-empty entries of A. Finally, create the pointer $\pi(A, i + 1)$ and let it point to the last non-empty position in A. Observe that this procedure runs in $\mathcal{O}(\Delta)$ time.

Restoring the parent. Finally, we describe how the algorithm returns to the parent T_{i-1} of a node T_i. Note that the case that T_i is an interesting leaf was already handled above, hence, assume that T_i is not an interesting leaf. When returning to T_{i-1}, first delete the last element of stack Q. Then, for each vertex in L_i, we remove its color c_i. Finally, remove pointer $\pi(A, i)$ from array A. Observe that this can be done in $\mathcal{O}(\Delta)$ time as well. Hence, the overall running time is $\mathcal{O}(\Delta)$ as claimed. Moreover, the size of stack Q is bounded by k, array A has a length of $\min(k\Delta, n)$, and the sum of the sizes of all lists L_i is at most $\min(k\Delta, n)$. Hence, *Simple* needs $\mathcal{O}(n+m)$ space. The proof of the correctness of the algorithm is deferred to a long version of the article. □

With this running time bound to we may now prove the claimed delay.

Theorem 1. *Enumerate with Simple solves CISE for any graph G where each connected component has order at least k and the maximum degree is Δ with delay $\mathcal{O}(k^2\Delta)$ and space $\mathcal{O}(n + m)$.*

Proof. *Enumerate* chooses an arbitrary start vertex v. According to Lemma 2, after the deletion of vertex v, we can delete every vertex of each connected component with less than k vertices in $\mathcal{O}(k^2\Delta)$ time. Thus it is sufficient to bound the time which is needed to output the next solution within *Simple.*

Consider a node T_i in the enumeration tree of one call of *Enumerate* with *Simple* and its associated sets P_i (the subgraph set of node T_i) and X_i (the extension set of node T_i). Every time we call *Simple* recursively, we add exactly one vertex to the subgraph set. Hence, we need at most k iterations to reach a leaf T_j. If T_j is interesting, that is, if we find a solution for CISE, then we have a delay of $\mathcal{O}(k\Delta)$. If T_j is boring, then according to Lemma 3 the pruning rule applies to each node T_ℓ on the path from T_j to T_i since no other subsequent child of node T_ℓ yields a path to an interesting leaf. Hence, we will return in altogether $\mathcal{O}(k\Delta)$ time to the parent T_{i-1} of node T_i. Now, we are in the same situation as above. Either the first path from node T_{i-1} to a leaf leads to a solution for CISE or the pruning rule applies and we return to the parent of T_{i-1}. The crucial difference is that the depth of node T_{i-1} in the enumeration tree is one less than the depth of node T_i. Since the depth of the enumeration tree is bounded by k, we can go up at most k times until we return from the root (which finishes this call to *Simple*). Each time, we either report a new solution in $\mathcal{O}(k\Delta)$ time or go up once more. Hence, the overall delay is $\mathcal{O}(k^2\Delta)$. The space complexity follows from Lemma 3. □

We can use Lemma 4 also to bound the overall running time of the algorithm.

Proposition 1. *Enumerate with Simple has running time* $\mathcal{O}((e(\Delta - 1))^{k-1} \cdot (\Delta + k) \cdot n/k)$.

4 Polynomial Delay with Pivot

We now adapt *Pivot* of Komusiewicz and Sorge [7] to obtain polynomial delay and a better running time bound. In *Pivot*, in each enumeration tree node, the vertex set of the subgraph set is partitioned into two sets P and S. The set P contains those vertices whose neighbors may still be added to extend the subgraph set and set S contains the other vertices of this subgraph, that is, no neighbor of S may be added to the subgraph. Moreover, we have a set F containing further vertices that may not be added to the connected subgraph. In the original algorithm [7] each node in the enumeration tree has an *active* vertex of the set P whose neighbors will be added to the subgraph. After adding each possible neighbor, the vertex becomes *inactive* and is added to set S. This version of the algorithm has a running time of $\mathcal{O}(4^k(\Delta - 1)^k n(n + m))$ [7] and no polynomial delay.

We improve this algorithm such that the number of enumeration tree nodes will be worst-case optimal and the algorithm has polynomial delay. The pseudo code of *Pivot* with improved running time and with pruning rule can be found in Algorithm 3. Consider a path $T_1, \ldots, T_i$ from the root T_1 to a node T_i of the enumeration tree. We will not associate enumeration tree nodes with active vertices. Instead, with each node T_i we associate P_i which is the subset of the subgraph set which can have further neighbors, S_i which is the remaining subgraph set, and F_i which is the set of forbidden vertices. Hence, we are using a Line 5 instead of creating a new child for each new active vertex. Now we do the following until P_i is empty: Pick an arbitrary $p \in P_i$. Next, for each neighbor v of p that is not in $P_i \cup S_i \cup F_i$, create a child node T_{i+1} in which v is added to P_i. After recursively solving the subproblem of T_{i+1}, move v to F_i. Consequently,

Algorithm 3 The *Pivot* algorithm; the initial call is $\mathit{Pivot}(\{v\}, \emptyset, \emptyset)$.

```
 1: procedure $\mathit{Pivot}(P, S, F)$
 2:     if $|P \cup S| = k$ then
 3:         output $P \cup S$
 4:         return
 5:     while $P \neq \emptyset$ do
 6:         $p :=$ choose element of $P$
 7:         for each $z \in N(p) \setminus (P \cup S \cup F)$ do
 8:             $\mathit{Pivot}(P \cup \{z\}, S, F)$
 9:             $F := F \cup \{z\}$
10:             if output of $\mathit{Pivot}(P \cup \{z\}, S, F)$ was empty then
11:                 return                    ▷ Stop recursion if no solution was found
12:         $P := P \setminus \{p\}$
13:         $S := S \cup \{p\}$
14:     return
```

v is contained in F_i in all subsequent children of T_i. Finally, after creating a child for each neighbor of p, remove p from P_i and put it into S_i. With this simple improvement, the number of enumeration tree nodes is now exactly the number of connected subgraphs of order at most k.

Lemma 5. *For each connected induced subgraph $G[U]$ of order at most k containing v, there is exactly one node T of the enumeration tree created by Pivot$(\{v\}, \emptyset, \emptyset)$ such that $P_T \cup S_T = U$.*

To obtain polynomial delay we add in Lines 10 and 11 a similar pruning rule to *Pivot* as for *Simple*: After each recursive call of *Pivot* in node T_i we check whether the call of node T_{i+1} outputs at least one solution for CISE. If not, we return in node T_i to its parent T_{i-1} of the enumeration tree. These two lines were not part of the original algorithm.

Lemma 6. *Let T_i be a node in the enumeration tree in a call of Pivot. If the output of a recursive call of Pivot in node T_i is empty, then no subsequent recursive call of Pivot in node T_i yields a path to an interesting leaf.*

Next, we prove that with suitable data structures for maintaining the sets P, S, and F during the enumeration, we can quickly traverse the enumeration tree.

Lemma 7. *Pivot can be implemented in such a way that for every node T_i of the enumeration tree, we need $\mathcal{O}(k\Delta)$ time to either compute the next child T_{i+1} or to restore the parent T_{i-1} and that the overall space needed is $\mathcal{O}(n + m)$.*

Proof. To check in constant time whether a vertex belongs to P_i, S_i, or F_i at an enumeration tree node T_i, we color some vertices of the graph with colors c_F, c_P, and c_S. For a node T_i the set of c_F-colored vertices represents the forbidden vertices F_i, the set of c_P-colored vertices represents the set of vertices P_i which can have new neighbors, and the set of c_S-colored vertices represents the set of vertices S_i which have no new neighbors. At the root of the enumeration tree, no vertex has color c_F or c_S. Only the single vertex v in P has color c_P. Testing if a vertex has color c_P, c_S, or c_F can be done in constant time.

To represent the partition of the subgraph set of node T_i into P_i and S_i we use an array A of length k. The array A contains $i = |P_i \cup S_i|$ nonempty elements. In A, we first save all vertices of S_i. Then the vertices of P_i follow. Further, a pointer $\pi(A, i)$ points to the vertex p of P_i with minimal index in A. Vertex p is the vertex which was chosen in Line 6 of *Pivot* and the vertex one position to the right of p will be chosen next. Hence, in node T_i altogether $|P_i \cup S_i|$ many pointers (one for node T_i and one for each of its ancestors) point to positions of A. To represent the set of forbidden vertices, we use a list L_i for each node T_i. The union of all vertices in lists $L_1, \ldots, L_i$ represents the set F_i of forbidden vertices in node T_i. List L_i contains all vertices in $F_i \setminus F_{i-1}$. List L_i is used to restore F_{i-1} when we return from node T_i to its parent T_{i-1}.

Initialization. If we call *Pivot* with the chosen start vertex v we create the root T_1 of the enumeration tree. The first and only non-empty entry of A contains v, pointer $\pi(A, 1)$ points to v, and list L_1 is empty. Now, we describe how to update these data structures in order to the next child T_{i+1}, or restore the parent T_{i-1} of any node T_i in $\mathcal{O}(k\Delta)$ time.

Determining the next child of T_i. Do the following while $\pi(A, i)$ points to a vertex p. Check in $\mathcal{O}(\Delta)$ time whether p has a neighbor u which has none of the colors c_P, c_S, or c_F. If yes, we have determined that by adding u to P_i we can create a new child T_{i+1} of node T_i. Otherwise, all neighbors of p have some color, and we remove color c_P from p, recolor p with c_S, and move pointer $\pi(A, i)$ one position to the right. If pointer $\pi(A, i)$ points to an empty entry of A, then P_i is empty and T_i contains no more children. Overall, we need $\mathcal{O}(k\Delta)$ time to determine the vertex to add for the next child T_{i+1} of T_i.

Creating a new child. To create T_{i+1}, we update the data structure to represent the sets P_{i+1}, S_{i+1}, and F_{i+1}: We replace the empty entry with minimal index in A by vertex u, we color u with c_P, and create pointer $\pi(A, i+1)$ which points to the same vertex as $\pi(A, i)$. Further, we create the list L_{i+1}. This list is empty since $F_{i+1} = F_i$. Thus, the child can be created in constant time.

Restoring a parent. Now, we prove that we can restore the parent T_{i-1} in $\mathcal{O}(k\Delta)$ time when we have determined that T_i has no further children: We need to restore the sets P_{i-1}, S_{i-1}, and F_{i-1} of the parent T_{i-1} of node T_i. All vertices in list L_i are forbidden vertices which were added in node T_i. In other words: $L_i = F_i \setminus F_{i-1}$. Removing color c_F from these vertices and deleting list L_i afterwards needs $\mathcal{O}(k\Delta)$ time, since the set P_i can have at most $k\Delta$ neighbors and hence, node T_i can have at most $k\Delta$ children. Next, we remove pointer $\pi(A, i)$ from A in constant time. Afterwards, we remove the last non-empty vertex x from A, add x to list L_{i-1}, and change the color of x to color c_F. To restore the coloring of P_{i-1} and S_{i-1} we use the position of pointer $\pi(A, i-1)$. More precisely, all vertices from $\pi(A, i-1)$ to the last non-empty entry of A get color c_P, and all other vertices of A get color c_S. Overall, we need $\mathcal{O}(k\Delta)$ time for this step.

As shown above, the algorithm has the claimed running time. Moreover, array A has length k and the sum of the list sizes is $\min(k\Delta, n)$. Hence, the algorithm needs $\mathcal{O}(n+m)$ space. $\square$

Together with the pruning rule, the above gives a delay of $\mathcal{O}(k^3\Delta)$.

Proposition 2. *Pivot can be implemented in such a way that Enumerate with Pivot solves* CISE *for any graph G where each connected component has order at least k and the maximum degree is Δ with delay $\mathcal{O}(k^3\Delta)$ and space $\mathcal{O}(n+m)$.*

Next, we will improve the delay to $\mathcal{O}(k^2\Delta)$. The bottleneck in the delay provided by Proposition 2 is that when we have a node T_i that does not lead to an interesting leaf, we may have to go up $\Theta(k)$ levels before reaching a node that leads to an interesting leaf, each time needing $\Theta(k^2\Delta)$ time to check if the current node T_i leads to an interesting leaf. We will do the following: Before generating child T_{i+1} of node T_i, we invest $\mathcal{O}(k\Delta)$ time to check if the next child T'_{i+1} of T_i yields a path to an interesting leaf. This will be done by coloring at most $k-i$ vertices with a new color c_t. If and only if $k-i$ vertices received color c_t the next child T'_{i+1} yields a path to an interesting leaf. Afterwards, color c_t will be removed from each vertex to use color c_t for the next node in the enumeration tree. With this we can prove the following delay bound.

Theorem 2. *Pivot can be implemented in such a way that Enumerate with Pivot solves* CISE *for any graph G where each connected component has order at least k and the maximum degree is Δ with delay $\mathcal{O}(k^2\Delta)$ and space $\mathcal{O}(n+m)$.*

Finally, we can prove a better running time bound for *Pivot*.

Proposition 3. *Enumerate with Pivot has running time $\mathcal{O}((e(\Delta-1))^{k-1}\cdot\Delta\cdot n)$.*

5 An Experimental Comparison

We implemented *Simple*, *Pivot*, *Exgen*, and *Kavosh* with and without the pruning rules. Note that adding the pruning rule to *Exgen* and *Kavosh* does not make them polynomial delay algorithms. We also implemented *BDDE* [11], the Reverse Search with dictionary (*RwD Old*), and the Reverse Search with predecessor (*RwP Old*) algorithm [4]. For reverse search-based algorithms we also implemented another method to determine neighbors in the supergraph (*RwD New* and *RwP New*).

Each experiment was performed on a single thread of an Intel(R) Xeon(R) Silver 4116 CPU with 2.1 GHz, 24 CPUs and 128 GB RAM running Python 2.7.14 with *igraph* (`http://igraph.org/python/`) as the general graph data structure and *NetworkX* (`https://networkx.github.io/`) as the data structure for maintaining the enumeration tree in *BDDE.* [1] As benchmark data set we used 30 sparse social, biological, and technical networks obtained from the Network Repository [12], KONECT [10], and the 10th DIMACS challenge [2] and 20

[1] The source code of our program *Enucon* is available at `www.uni-marburg.de/fb12/arbeitsgruppen/algorithmik/software/`.

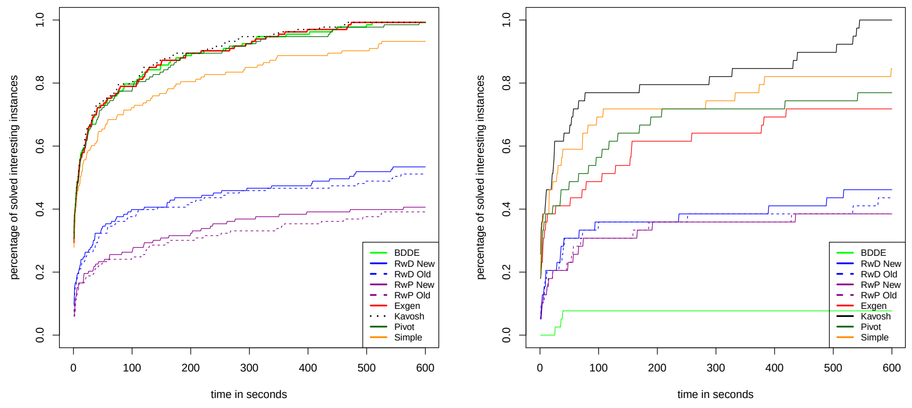

Fig. 2. Comparison for $k \in \{3, \ldots, 10\}$ (left) and $k \in \{n_c - 1, n_c - 2, n_c - 3\}$ (right) on interesting instances.

random graphs generated in the $G_{n,p}$ model with $n \in \{100, 200, \ldots, 1000\}$ and $p \in \{0.1, 0.2\}$. The real-world networks range from very small (up to 500 vertices) to very large networks (up to 500 000 vertices).

Each algorithm was run on each instance with a time limit of 600 seconds. An instance is *interesting* if at least one of the 14 algorithms solved it within the time limit. For *Simple*, *Pivot*, *Exgen*, and *Kavosh* only the variant with the pruning rule is plotted in Fig. 2 since these variants were the fastest. Fig. 2 shows the result for $k \in \{3, \ldots, 10\}$. Both versions of *RwD* and *RwP* only solve half as many instances as the other algorithms. All instances solved by *RwD* were solved by the remaining algorithms in 20 seconds. *Simple* is a factor 2 slower than *Pivot*, *BDDE*, *Exgen*, and *Kavosh*; *Kavosh* is slightly faster than *Pivot*, *BDDE*, and *Exgen*. Hence, for small k, one should use *Kavosh*.

Fig. 2 shows the result for $k \in \{n_c - 1, n_c - 2, n_c - 3\}$ where n_c is the order of the largest connected component in the graph. Since *BDDE* stores the enumeration tree, it produced many memory errors and solved only the smallest instances. All instances solved by *RwD* or *RwP* were solved by *Pivot*, *Simple*, *Exgen*, and *Kavosh* with pruning rules in less than 100 seconds. The versions of the algorithms without the pruning rule only solved the same number of instances as *BDDE*. Hence, adding these pruning rules was necessary to solve CISE for large k. Again, *Kavosh* is the fastest algorithm, despite the fact that adding the pruning rule to *Kavosh* does not yield polynomial delay. Hence, for large k, also *Kavosh* should be used. It seems that *Pivot* is slower for large k because it may spend $\Theta(k\Delta)$ time before creating the next child.

References

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

[2] Bader, D.A., Kappes, A., Meyerhenke, H., Sanders, P., Schulz, C., Wagner, D.: Benchmarking for graph clustering and partitioning. In: Encyclopedia of Social Network Analysis and Mining, pp. 73–82. Springer (2014)

[3] Bollobás, B.: The Art of Mathematics – Coffee Time in Memphis. Cambridge University Press (2006)

[4] Elbassioni, K.M.: A polynomial delay algorithm for generating connected induced subgraphs of a given cardinality. Journal of Graph Algorithms and Applications **19**(1), 273–280 (2015)

[5] Elbassuoni, S., Blanco, R.: Keyword search over RDF graphs. In: Proceedings of the 20th ACM Conference on Information and Knowledge Management, (CIKM '11). pp. 237–242. ACM (2011)

[6] Kashani, Z.R.M., Ahrabian, H., Elahi, E., Nowzari-Dalini, A., Ansari, E.S., Asadi, S., Mohammadi, S., Schreiber, F., Masoudi-Nejad, A.: Kavosh: a new algorithm for finding network motifs. BMC Bioinformatics **10**, 318 (2009)

[7] Komusiewicz, C., Sorge, M.: Finding dense subgraphs of sparse graphs. In: Proceedings of the 7th International Symposium on Parameterized and Exact Computation (IPEC '12). Lecture Notes in Computer Science, vol. 7535, pp. 242–251. Springer (2012)

[8] Komusiewicz, C., Sorge, M.: An algorithmic framework for fixed-cardinality optimization in sparse graphs applied to dense subgraph problems. Discrete Applied Mathematics **193**, 145–161 (2015)

[9] Komusiewicz, C., Sorge, M., Stahl, K.: Finding connected subgraphs of fixed minimum density: Implementation and experiments. In: Proceedings of the 14th International Symposium on Experimental Algorithms (SEA '15). Lecture Notes in Computer Science, vol. 9125, pp. 82–93. Springer (2015)

[10] Kunegis, J.: KONECT: the Koblenz network collection. In: Proceedings of the 22nd International World Wide Web Conference (WWW '13). pp. 1343–1350. International World Wide Web Conferences Steering Committee / ACM (2013)

[11] Maxwell, S., Chance, M.R., Koyutürk, M.: Efficiently enumerating all connected induced subgraphs of a large molecular network. In: Proceedings of the First International Conference on Algorithms for Computational Biology (AlCoB '14). Lecture Notes in Computer Science, vol. 8542, pp. 171–182. Springer (2014)

[12] Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI '15). pp. 4292–4293. AAAI Press (2015), `http://networkrepository.com`

[13] Wernicke, S.: A faster algorithm for detecting network motifs. In: Proceedings of the 5th International Workshop on Algorithms in Bioinformatics (WABI '05). Lecture Notes in Computer Science, vol. 3692, pp. 165–177. Springer (2005)

[14] Wernicke, S.: Combinatorial algorithms to cope with the complexity of biological networks. Ph.D. thesis, Friedrich Schiller University of Jena (2006), `http://d-nb.info/982598882`