Enumerating Connected Induced Subgraphs: Improved Delay and Experimental Comparison[⋆]

Christian Komusiewicz, Frank Sommer[1]

Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany

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. Moreover, we show that these two algorithms can be adapted to give algorithms for the problem of enumerating all connected induced subgraphs of order at most k with a delay of $\mathcal{O}(k+\Delta)$. Finally, we perform an experimental comparison of several enumeration algorithms for $k \leq 10$ and $k \geq |V| - 3$.

1. Introduction

We study algorithms for the following fundamental graph problem.

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

We call a connected subgraph of order k a *solution* in the following. The enumeration of connected subgraphs is important in many applications. It is used, for example, in 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 [8, 19]. A further application arises when semantic web data is searched using only keywords instead of structured queries [6]. In this application, one is interested in all connected subgraphs not only induced ones. E-CISE can be used as a subroutine here, however, since all connected subgraphs can be obtained by enumerating all connected subgraphs of each

[⋆]An extended abstract of this work appeared in the proceedings of the 45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '19) held in Nový Smokovec, Slovakia.

Email addresses: `komusiewicz@informatik.uni-marburg.de` (Christian Komusiewicz), `fsommer@informatik.uni-marburg.de` (Frank Sommer)

[1]Frank Sommer was supported by the DFG, project MAGZ (KO 3669/4-1).

E-CISE solution. Finally, many fixed-cardinality optimization problems can be solved by an algorithm whose first step is to enumerate connected induced subgraphs of order k [11]. 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, 12].

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

One way to achieve relevant running time bounds is to consider degree-bounded graphs. Here, the number of solutions is much smaller than in general as shown by the following bound due to Bollobás [4].

Lemma 1 ([4, 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 $\mathcal{O}((e(\Delta-1))^{(k-1)} \cdot (n/k))$ where n is the number of vertices of G.*

This observation can be exploited to obtain an algorithm for E-CISE that runs in $\mathcal{O}((e(\Delta-1))^{(k-1)} \cdot (\Delta+k) \cdot (n/k))$ time [11].

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 solution 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 [2]. When we are interested only in solutions of order *exactly* k, one may use the algorithm for enumerating solutions of order at most k and simply filter those of order less than k. The running time of this algorithm, however, is *not* output-polynomial, that is, it is not bounded by a polynomial in the input and output size. Consequently, the algorithm does not achieve polynomial delay either. A different reverse search algorithm, however, achieves delay $\mathcal{O}(k \min(n-k, k\Delta)(k(\Delta+\log k)+\log n))$ [5].

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

Related Work. Most known E-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 [19] describes a very simple 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 $N(S)$, the neighborhood of S. Another popular enumeration algorithm is *Kavosh* [8] 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 $\mathcal{O}((4(\Delta - 1))^k \cdot (\Delta + k) \cdot n)$ [10]. 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 [11]. This variant of Pivot, 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. A further algorithm for E-CISE is *BDDE* [15]. 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 already explored parts of the enumeration tree.

An output-sensitive algorithm for E-CISE with running time $\mathcal{O}(\sum_{G^* \in \mathcal{S}} |G^*|)$, where $\mathcal{S}$ is the set of all E-CISE solutions and $|G^*|$ is the total size of G^*, was presented by Ferreira [7]. This running time is optimal in terms of the overall running time when the task is to fully output all solutions, not only their vertex sets. The basic idea of this algorithm, which is closely related to *Simple*, is to create a binary search tree whose nodes represent connected sets S of G and whose leafs represent solutions. In each search tree node, the algorithm selects a vertex v from $N(S)$ and branches into two cases: first it enumerates the solutions that contain $S \cup \{v\}$, then those containing S but not v. Further, a certificate is used to ensure that in each node of the search tree, there exists at least one solution that contains S. Ferreira [7] does not bound the delay of this algorithm.

The known algorithms with polynomial delay [5] work differently. They use reverse search or, more generally, the supergraph method [2]. There, for a given graph G and parameter k, the supergraph $\mathcal{G}$ contains a node for each E-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 E-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 [5]. Hence, *RwD* has a better delay than *RwP* but requires exponential space, since $|\mathcal{G}|$ may grow exponentially with the size of G.

For the problem of enumerating all connected induced subgraphs of G that have order at most k, an algorithm with delay $\mathcal{O}(nm)$ was presented already in the work that introduced the reverse search framework [2]. The more recent

RSSP algorithm enumerates all connected induced subgraphs with delay $\mathcal{O}(n_c)$ where n_c is the size of the largest connected component of G [1]. The basic idea of *RSSP* is also to use reverse search but with a more strict neighborhood definition. Finally, when the running time for outputting the solution is not counted, then all connected induced subgraphs can be enumerated in amortized time $\mathcal{O}(1)$ per connected induced subgraph [18].

Our Results. We show how to adapt *Simple* and *Pivot* for E-CISE in such a way that the worst-case delay between the output of two solutions is $\mathcal{O}(k^2\Delta)$ and that the algorithm requires $\mathcal{O}(n+m)$ space. This improves over the previous best delay bound of *RwD* [5] while requiring only linear space. As a side result, we show that these variants of *Simple* and *Pivot* achieve an overall running time of $\mathcal{O}((e(\Delta-1))^{(k-1)} \cdot (\Delta+k) \cdot n/k)$. For *Simple* this is the first running time bound; for *Pivot* this is a substantial improvement over the previous running time bound. In addition, we further explore the connections between *Simple* and *Pivot* and show that a certain implementation of *Pivot* is in fact just a variant of *Simple*. Furthermore, we improve the delay of *RwP* to $\mathcal{O}(k^2 \min(n-k, k\Delta) \cdot \min(k\Delta, (n-k)(\Delta+\log k)))$.

Moreover, we give delay bounds for the problem of enumerating all connected induced subgraphs of order *at most k*. More precisely, we show that *Simple* and *Pivot* achieve a delay of $\mathcal{O}(k+\Delta)$. Hence, the best known delay bound is much lower than for E-CISE.

We then implement these algorithms in Python and compare them experimentally with Python implementations of *Kavosh* [8], *Exgen* [11], and *BDDE* [15]. For $k \leq 10$, we observe that *RwD* and *RwP* are significantly slower than the other algorithms, which behave quite similarly in terms of the overall running time. For very large k, that is, for k close to the order of the largest connected component of G, *RwD* and *RwP* are again slower than the other algorithms and the differences between these algorithms are larger in this case. Here, *Pivot* has the best overall running times. Finally, for the variant where we aim to output all solutions of order at most k, we included *RSSP* in the comparison and excluded *RwD* and *RwP* since they were not designed for this variant. The main result is that all algorithms behave roughly similar in this case.

2. Preliminaries and Main Algorithm

Graph Notation. We consider undirected simple graphs $G = (V, E)$. The *order* of a graph is the number of its vertices. We let n and m denote the order of G and the number of edges in G, respectively. 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. Let $W \subseteq V$ be a vertex set. Then $N(W) := \bigcup_{v \in W} N(v) \setminus W$ 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. 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

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 E-CISE solutions containing $v$ with Enum-Algo
5:         remove $v$ from $G$
```

a maximal subgraph where any two vertices are connected to each other by at least one path.

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 pseudocode 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 $|V| - k + 1$ enumeration trees. To avoid confusion, we refer to the vertices of the enumeration trees as *nodes*. Each node represents a connected induced subgraph $G[S]$ of order at most k. We refer to the vertex set S of this subgraph as *subgraph set* of the enumeration tree node. Roughly speaking, a node N is a child of another node M if the subgraph corresponding to M is an induced 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. Furthermore, 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 E-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.*

Proof. The only vertices of $G - \{v\}$ that are in connected components of order less than k, are those that are in the same connected component as v in G. We may thus check for each connected component of $G - \{v\}$ which contains at least one neighbor of v whether this component has order at most $k - 1$. Using

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 True
5:     hasIntLeaf := False
6:     while $X \neq \emptyset$ do
7:         $u$ := choose last vertex from $X$
8:         delete $u$ from $X$                    ▷ The current set $P$ will be extended
9:         $X' := X \cup (N(u) \setminus N[P])$
10:        if Simple($P \cup \{u\}, X'$) = True then
11:            hasIntLeaf := True
12:        else
13:            return hasIntLeaf          ▷ Stop recursion if no new solution found
14:    return hasIntLeaf
```

depth-first search, this check needs $\mathcal{O}(k^2)$ time per neighbor. Moreover, for each neighbor which is in a connected component of order less than k, we can remove the connected component from G in $\mathcal{O}(k^2)$ time. Since v has at most Δ neighbors in G, the total running time of the algorithm is $\mathcal{O}(k^2\Delta)$. □

3. Polynomial Delay with Simple

We now adapt *Simple* to obtain a polynomial delay algorithm; the pseudocode is shown in Algorithm 2. In *Simple*, we start with a single vertex v and find successively larger connected subgraphs containing v. The subgraph set is denoted by P. Furthermore, the set $X \subseteq N(P)$, called *extension set*, contains those neighbors of P which can be added to P to enlarge the subgraph $G[P]$. In Lines 7-9, when putting $u \in X$ in the set P, we remove u from X and add to X each neighbor of u which is not in $N[P]$. Lines 5 and 10–13 of Algorithm 2 and returning the Boolean flag hasIntLeaf are not part of the plain version of *Simple* [19]. In these lines, a new pruning rule is performed; this rule is necessary to establish polynomial delay for E-CISE.

To describe the pruning rule, we introduce some notation. Let $T_1, \dots, T_i$ denote the nodes of a path from the root T_1 to a node T_i of the enumeration tree. We denote the subgraph set of a node T_i by P_i and its extension set by X_i. To avoid 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 interesting if the corresponding subgraph set P_j is a solution for E-CISE, that is $|P_j| = k$, and that T_j is boring otherwise.

Lemma 3 *Let T_i be a node in the enumeration tree of Simple. 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.*

Proof. Let $T_i, \dots, T_j$ denote the path in the enumeration tree from node T_i to a leaf T_j, where node $T_{\ell+1}$ is the first child of node T_ℓ for each $\ell \in \{i+1, \dots, j-1\}$. By assumption, the leaf T_j is boring, that is, $|P_j| < k$. This implies that T_j has an empty extension set X_j. Hence, the number $j-i$ of vertices that are added between node T_i and leaf T_j is equal to the number of all vertices in the graph $G - (N[P_i] \setminus X_i)$ which are in a connected component with the vertices of the subgraph set P_i of node T_i. In other words, adding all possible vertices in node T_i does not give a connected subgraph of order at least k.

After returning from node T_{i+1} to node T_i, the vertex u that was added to the subgraph set P_{i+1} of node T_{i+1} is removed from the extension set X_i. Hence, the number of vertices in $G-(N[P_i]\setminus X_i)$ which are in the same connected component with all vertices of the subgraph set P_i is at least one more than the number of vertices in $G - (N[P_i] \setminus \{X_i \setminus \{u\}\})$ which are in the same connected component with all vertices of the subgraph set P_i. Thus, also the next child of node T_i, and any further child of T_i will not lead to an interesting leaf. $\square$

By the above, we can return to the parent T_{i-1} of T_i as soon as one of the recursive branches fails to output a solution.

To check efficiently whether the pruning rule applies, we do the following: Each enumeration tree node T has a Boolean flag hasIntLeaf initialized with `False` in Line 5. As soon as at least one recursive call has an interesting leaf, the Boolean flag hasIntLeaf is set to `True`. Therefore, the algorithm correctly returns whether or not T leads to an interesting leaf. If some recursive call does not lead to an interesting leaf, then the algorithm returns immediately to the parent of node T which is correct due to Lemma 3. The overall overhead for performing the pruning is only a constant factor.

We now show that *Simple* achieves a polynomial delay when this pruning rule is performed. To achieve the claimed delay of $\mathcal{O}(k^2\Delta)$, 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 $P_i \setminus P_{i-1} = \{p_i\}$. First, we prove that for a node T_i in the enumeration tree we need $\mathcal{O}(\Delta)$ time to either compute the sets P_{i+1} and X_{i+1} of its next child T_{i+1} or to restore the sets P_{i-1} and X_{i-1} of its parent node 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 G with $k+1$ colors $c_0, \dots, c_k$ as follows. Following the notation of Wernicke [19], for a node T_i, we call the vertices which are in $N[P_i] \setminus N[P_{i-1}]$ where T_{i-1} is the parent of T_i the *exclusive neighbors of p_i*. These are exactly the vertices that are added to X_{i-1} in Line 9 of Algorithm 2 to construct the extension set X_i for the node T_i. Throughout the algorithm we maintain the following invariant:

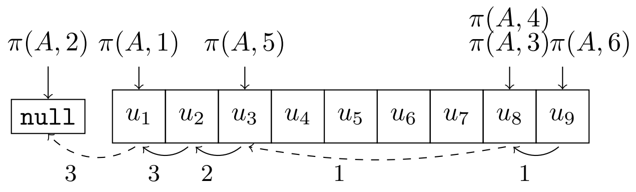

Figure 1: An example for the pointer movement: Pointer $\pi(A, 6)$ points to u_9, an exclusive neighbor of p_6. Before adding u_9 to the subgraph set P_6, we move pointer $\pi(A, 6)$ to the left to u_8, an exclusive neighbor of vertex p_3. Since T_5 is the parent of T_6 we move $\pi(A, 6)$ to u_3 which is the position of pointer $\pi(A, 5)$. 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 and 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. Since T_2 is the parent of T_3 we move $\pi(A, 6)$ to the position of $\pi(A, 2)$. 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.

> Let $T_1, \ldots, T_i$ for $i \leq k$ be the path from the root T_1 to a node T_i. The vertex p_1 has color c_0. A vertex v has color c_i, $i \geq 1$, if and only if v is an exclusive neighbor of p_i.

Altogether, for $0 \leq j \leq k$, 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 $\min(k\Delta, n)$ 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. In Line 9 of Algorithm 2 when the new extension set X' is created from X, the new vertices of $N(u) \setminus N[P]$ replace the left-most empty entries of A. Pointer $\pi(A, i)$ points to the vertex x in the extension set X_i which will be added to P_i in the *next* recursive call of *Simple* in node T_i. If for 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 restore the sets P_{i-1} and X_{i-1} to the parent T_{i-1} 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, and 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 $\pi(A, i)$ 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; an example of the pointer movement is given in Figure 1. The vertex x is an exclusive neighbor of p_ℓ for some $\ell \leq i$. 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 $j < \ell$, then move $\pi(A, i)$ to the position that $\pi(A, \ell-1)$ points to. Since y is an exclusive neighbor of p_j pointer $\pi(A, j)$ points to y. 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 $\pi(A, i+1)$ 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. In the following, assume that T_i is not an interesting leaf. When returning to T_{i-1}, first delete the last element p_i 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.

We conclude that 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.

We now prove that the pointer structure faithfully represents the extension sets during the course of the algorithm. More precisely, we show that each pointer $\pi(A, i)$ visits all the vertices that are contained in the extension set X_i when T_i is created. To this end, we define inductively, for each $\pi(A, j)$ pointing on the array A, a subset $A_{\leq j}$ of the entries of A: The set $A_{\leq 1}$ contains all entries to the left of $\pi(A, 1)$ including the entry that $\pi(A, 1)$ points to. The set $A_{\leq j}$

contains all entries of A that are to the left of $\pi(A, j)$ and either in $A_{\leq j-1}$ or exclusive neighbors of T_j. We now claim the following invariant during the algorithm by induction over the operations of the algorithm.

> Let T_i be a enumeration tree node, and let $T_1, \ldots, T_i$ denote the nodes on the path from the root T_1 of the enumeration tree to node T_i. At any point in time $A_{\leq i}$ contains exactly the set X_i.

The claim is obviously true for the root T_1: Initially, A contains exactly the vertices of $N(p_1)$ and all of them are to the left of the position of $\pi(A, 1)$. Every time the pointer $\pi(A, 1)$ moves to the left, it moves by exactly one position and removes exactly one vertex from X_1 as prescribed in Line 8 of Algorithm 2.

Now, assume by induction that the claim holds for all T_j with $j < i$. When creating T_i, the extension set X_i may, according to the pseudocode of *Simple*, contain all vertices which are in X_{i-1} or exclusive neighbors of p_i. When creating T_i, the pointer $\pi(A, i)$ points to the rightmost non-empty position in the array A. Thus, $A_{\leq i}$ contains all vertices which are exclusive neighbors of p_i or are contained in $A_{\leq i-1}$. By the inductive hypothesis, the latter set contains exactly the vertices of X_{i-1}. When the pointer $\pi(A, i)$ is moved to the left, this corresponds to removing the element that $\pi(A, i)$ points to from X_i as prescribed in Line 8 of Algorithm 2 (where we remove u). To prove the claim that the procedure of moving to the left is correct, we show that the pointer stops at the rightmost position of A containing an element of X_i. We distinguish three cases:

Case 1: $\pi(A, i)$ now points to some vertex with color c_i. Then this vertex is an exclusive neighbor of T_i and the pointer has moved exactly one position to the left. Hence, the pointer stops at the rightmost position of A containing an element of X_i.

Case 2: $\pi(A, i)$ now points for the first time to some vertex x with color c_j such that $j < i$. Then $\pi(A, j)$ points to the same position as $\pi(A, i)$, since x is the rightmost remaining vertex of the extension set X_j. If $j = i - 1$, then the pointer position is not changed further and $\pi(A, i)$ now points to the rightmost exclusive neighbor of T_{i-1} which is the rightmost vertex of X_i in A since A contains no exclusive neighbors of T_i anymore. Otherwise, the pointer jumps to the position of $\pi(A, i - 1)$. Since all elements of X_{i-1} are contained in $A_{\leq i-1}$ and since, by induction, $\pi(A, i - 1)$ points to the rightmost element of X_{i-1} we have that $\pi(A, i)$ points at the rightmost position of A containing an element of X_i.

Case 3: $\pi(A, i)$ was moved from a vertex with color c_j such that $j < i$ to the left. In that case, the movement of $\pi(A, i)$ is exactly the same as the movement of $\pi(A, i - 1)$ when the algorithm visits node T_{i-1}. By induction, we assume that this movement visits all the vertices of X_{i-1}.

Hence, when moving the pointer to the left, we do not miss an element of X_i and we only stop at elements of X_i. □

With this running time bound to compute the next child or to restore the parent at hand, we may now prove the claimed delay.

Theorem 1 *Enumerate with Simple solves* E-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*.

We compute the sets P_1 and X_1 in $\mathcal{O}(\Delta)$ time. Since Lemma 2 was applied, by adding $k-1$ vertices, we obtain a solution for E-CISE. Outputting the corresponding connected induced subgraph needs $\mathcal{O}(k)$ time. Hence, the first solution for a call of *Enumerate* will be output in $\mathcal{O}(k\Delta)$ time.

Next, we show that in $\mathcal{O}(k^2\Delta)$ time we either find a new solution for E-CISE or end this call of *Enumerate*. By induction we will show that in $\mathcal{O}((k-j)k\Delta)$ time we can restore the sets P_j and X_j of node T_j for some $j<k$. Let T_k be an interesting leaf. According to Lemma 4, the sets P_{k-1} and X_{k-1} of node T_{k-1} can be restored in $\mathcal{O}(\Delta)$ time. Assume the sets P_j and X_j are restored in $\mathcal{O}((k-j)k\Delta)$ time. 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_ℓ. If T_ℓ is interesting, that is, if $\ell=k$, then we output the corresponding connected induced subgraph in $\mathcal{O}(k)$ time. Thus, in this case the algorithm outputs the next solution in $\mathcal{O}((k-j+1)k\Delta)$ time. If T_ℓ is boring, that is, if $\ell<k$, then according to Lemma 3 the pruning rule applies to each node T_q on the path from T_ℓ to T_j since no other subsequent child of node T_q yields a path to an interesting leaf. Hence, we will return in altogether $\mathcal{O}(k\Delta)$ time to the parent T_{j-1} of node T_j. Hence, the sets P_{j-1} and X_{j-1} can be restored in $\mathcal{O}((k-j+1)k\Delta)$ time. Thus, in $\mathcal{O}(k^2\Delta)$ time we either output the next solution or return in T_1 and end this call of *Enumerate*. Hence, the overall delay is $\mathcal{O}(k^2\Delta)$. The space complexity follows from Lemma 4. $\square$

Observe that it is necessary to delete vertices of components with less than k vertices after one call of *Enumerate* to obtain the claimed delay of $\mathcal{O}(k^2\Delta)$: Otherwise, after some calls to *Simple* we may end up with a graph containing many vertices which are not contained in solutions. Starting the enumeration from these vertices one after the other would not give a delay that is polynomial in k and Δ.

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

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

Proof. Each connected induced subgraph with at most k vertices is output exactly once [19]. Hence, for two different nodes T and Q in the enumeration tree, we have $P_T \neq P_Q$. In other words, each enumeration tree node corresponds to a different connected subgraph of order at most k. According to Lemma 1, the overall number of these subgraphs containing some vertex v is $\mathcal{O}(\sum_{i=1}^{k}(e(\Delta-1))^{i-1}) = \mathcal{O}((e(\Delta-1))^{k-1})$ where the equality follows from the fact that $2(e(\Delta-$

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

```
1: procedure Pivot$(P, S, p, F)$
2:     if $|P \cup S| = k$ then
3:         output $P \cup S$
4:         return
5:     if $p =$null then
6:         if $P \neq \emptyset$ then
7:             $p :=$ choose some element of $P$
8:         else
9:             return
10:    for $z \in N(p) \setminus \{P \cup S \cup F\}$ do
11:        Pivot$(P \cup \{z\}, S, p, F)$
12:        $F := F \cup \{z\}$
13:    Pivot$(P \setminus \{p\}, S \cup \{p\},$ null$, F)$
14:    return
```

$1))^{i-1} < (e(\Delta-1))^i$ for $\Delta \geq 2$. Consequently, the total number of enumeration tree nodes in all calls of *Enumerate* with *Simple* is $\mathcal{O}((e(\Delta-1))^{k-1} \cdot n/k)$.

Now, we bound the time per node T in the enumeration tree. Let p_T be the vertex that was added to the subgraph set to create the node T. Determining the neighbors of vertex p_T and adding the exclusive neighbors of p_T, that are those without a color, to X_T needs $\mathcal{O}(\Delta)$ time. For each vertex z in X_T we make a recursive call where we add z to the subgraph set. The recursive call includes updating the subgraph set and the extension set and can be done in $\mathcal{O}(\Delta)$ time per call. By charging this running time to the corresponding child in the enumeration tree, we obtain a running time of $\mathcal{O}(\Delta)$ per enumeration tree node. Since outputting a solution needs $\mathcal{O}(k)$ time we obtain a running time of $\mathcal{O}(k + \Delta)$ per enumeration node. The overall running time follows. $\square$

4. From Pivot to a Variant of Simple

We now adapt *Pivot* of Komusiewicz and Sorge [10] to obtain polynomial delay and a better running time bound. More precisely, we show that a careful implementation of an adaption of *Pivot* actually turns it into a variant of *Simple*. The pseudocode of *Pivot* as described by Komusiewicz and Sorge [10] can be found in Algorithm 3; the algorithm works as follows. In each enumeration tree node, 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. 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 removed from P and added to S. This version of the algorithm

Algorithm 4 An adaptation of *Pivot* without active vertex; the initial call is *Pivot*$(\{v\}, \emptyset, \emptyset)$.

```
1:  procedure 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 first element of $P$
7:          for each $z \in N(p) \setminus (P \cup S \cup F)$ do
8:              Pivot$(P \cup \{z\}, S, F)$
9:              $F := F \cup \{z\}$
10:         $P := P \setminus \{p\}$
11:         $S := S \cup \{p\}$
12:     return
```

has a running time bound of $\mathcal{O}(4^k(\Delta - 1)^k n(n + m))$ [10] and no polynomial delay.

The pseudocode of the adaption of *Pivot* can be found in Algorithm 4. This variant already has, up to polynomial factors, a worst-case optimal overall running time. Since our implementation of this algorithm eventually leads to a variant of *Simple*, we omit the proof and show running time bounds only for the final version. Nevertheless, we believe it is instructive to discuss this intermediate version of the algorithm. 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 the set P_i which is the subset of the subgraph set which can have further neighbors, the set S_i which is the remaining subgraph set, and the set F_i which is the set of forbidden vertices. Now, instead of creating a new child when choosing a new active vertex we are using the while-loop starting in Line 5 in Algorithm 4. In this while-loop we do the following until P_i is empty: Pick the vertex $p \in P_i$ that was added first to P_i. That is, if p_i is the ith vertex that was added in the creation of the node T_i, then p_i becomes the active vertex p exactly after p_{i-1} moves from P to S. Otherwise, the algorithm is the same as the original one. Observe that when creating a child in Line 8, the active vertex (which is now only implicitly given) remains the same by the choice of p in Line 6.

We now further modify this variant of *Pivot*. Assume that the current vertex p is the ith vertex p_i that was added to P. The main idea is to show that the set $N(p) \setminus (P \cup S \cup F)$ in Line 7 from which the next vertex z is chosen can be computed already when p_i is added to P. By the convention that p_i is chosen as the first vertex of P, the set S when p_i becomes active is predetermined, it is exactly $\{p_1, \ldots, p_{i-1}\}$. Moreover, whenever a vertex p_j, $j < i$, moves from P to S in Line 10, every neighbor of p_j is either in $P \cup S$ or in F. Hence, the set $N(p) \setminus (P \cup S \cup F)$ for $p = p_i$ is exactly the set $N(p_i) \setminus N[P \cup S]$. Thus, instead of saving the set F of forbidden vertices, we may work with a representation

Algorithm 5 The *Simple-Forward* algorithm, an implementation of *Pivot*; the initial call is *Simple-Forward*$(\{v\}, N(v))$.

```
1:  procedure Simple-Forward($P, X$)
2:      if $|P| = k$ then
3:          output $P$
4:          return True
5:      hasIntLeaf := False
6:      while $X \neq \emptyset$ do
7:          $u$ := choose first vertex from $X$
8:          remove $u$ from $X$              ▷ The current set $P$ will be extended
9:          $X' := X.\text{append}(N(u) \setminus N[P])$
10:         if $Pivot(P \cup \{u\}, X')$ = True then
11:             hasIntLeaf := True
12:         else
13:             return hasIntLeaf      ▷ Stop recursion if no new solution found
14:     return hasIntLeaf
```

of an extension set X as in *Simple.* When adding a vertex z to the subgraph set, we add exactly those neighbors of z to the extension set X that are not neighbors of any vertex in $P \cup S$, that is, we add the vertices in $N(z) \setminus N[P \cup S]$.

As for *Simple*, the extension set X will be represented by an array A with several pointers pointing on this array. The difference is that these pointers move through A in forward direction, that is, from low indices to high indices. The subgraph set will be denoted by P only, that is, there is no need to store the set S anymore, it is implicitly represented by the position of a pointer on A. The same is true for the set F which is also not stored explicitly anymore. The pseudocode of this new implementation of *Pivot* is shown in Algorithm 5; in light of the previous discussion, we will call it *Simple-Forward.* To highlight the differences between *Simple* and *Simple-Forward*, we emphasize that X has an order by using list notation, that is, initially X is a list of the neighbors of $v = p_1$, new vertices in X are appended and the next vertex is chosen from the front of the list instead of from the back as in *Simple.* Before proving the running time bounds for *Simple-Forward*, let us remark that an inspection showed that the implementation of *Simple* in the FANMOD tool of Wernicke and Rasche [20] is essentially the same as *Simple-Forward* except for the parts that are relevant for the pruning rule which is needed to establish polynomial delay (Lines 5 and 10–13 and returning the Boolean flag hasIntLeaf).

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

Lemma 5 *Simple-Forward 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 use the same data structures as described in Lemma 4: an array A

for the representation of X, for each i, a list L_i representing the exclusive neighborhood of a vertex p_i, a stack Q representing the subgraph set P, and a coloring of the vertices to allow for an $\mathcal{O}(1)$-time test for containment in the exclusive neighborhood of some P_i. There is one difference, however: Instead of using $k+1$ colors, one for each exclusive neighborhood, we use only one color c for all vertices in $N[P_i]$. Next, we describe how these data structures are maintained during 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 $A[1]$. 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. A node T_i has a further child T_{i+1} if $\pi(A,i)$ points to an index of A containing some vertex x. We create child T_{i+1} as follows: If x is the last entry of A, redirect $\pi(A,i)$ to `null`. Otherwise, move $\pi(A,i)$ one position to the right. Afterwards, create the child T_{i+1} with $p_{i+1} := x$ as follows. If node T_{i+1} is an interesting leaf, that is, if $i = k-1$, we output $P_i \cup \{x\}$ 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 color c. If this is not the case, then color u with color c 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 same position as pointer $\pi(A,i)$, then enter the recursive call for T_{i+1}. Observe that this procedure runs in $\mathcal{O}(\Delta)$ time.

Restoring the parent. We describe how the algorithm returns to the parent T_{i-1} of a node T_i; 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. This procedure runs in $\mathcal{O}(\Delta)$ time.

The overall space complexity of $\mathcal{O}(n+m)$ follows by the same arguments as in the proof of Lemma 4.

It remains to show that the pointer structure faithfully represents the extension sets during the course of the algorithm. We have to show that each pointer $\pi(A,i)$ visits all vertices contained in the extension set X_i when node T_i is created. By $A_{>i}$ we denote the set of vertices in A beginning at $\pi(A,i)$ and ending at $A[|L_1| + \ldots + |L_i|]$. Note that $A[1,\ldots,|L_1| + \ldots + |L_i|]$ represents $N[P_i] \setminus \{p_1\}$. As for *Simple*, we claim the following invariant during the algorithm by induction over the operations of the algorithm.

> Let T_i be a enumeration tree node, and let $T_1,\ldots,T_i$ denote the nodes on the path from the root T_1 of the enumeration tree to T_i. At any point in time $A_{>i}$ contains exactly the set X_i.

As for *Simple*, the statement is obviously true for the root T_1 with the difference that $\pi(A,1)$ initially points to $A[1]$. Now, assume the claim holds for all T_j with $j < i$. When node T_i is created, the corresponding extension set is $X_i := X_{i-1} \cup N(P_i) \setminus N[P_{i-1}]$. According to the induction hypothesis $A_{>i-1}$ correctly represents X_{i-1}. Furthermore, $L_i = N[P_i] \setminus N[P_{i-1}]$. Hence, when T_i is created, $A_{>i}$ correctly represents X_i. Each time a child T_{i+1} of node T_i is created, pointer $\pi(A, i)$ is moved exactly one position to the right. Hence, pointer $\pi(A, i)$ visits all vertices in X_i. □

The correctness of the pruning rule for *Simple-Forward* is similar to the proof of the correctness of the pruning rule for *Simple* in Lemma 3 . Moreover, the proof of the delay bound of *Simple* in Theorem 1 also applies to *Simple-Forward*. Hence, we obtain directly obtain the following bound.

Corollary 1 *Enumerate with Simple-Forward solves* E-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, the proof of Proposition 1 which bounds the overall running time for *Simple* applies also to *Simple-Forward*.

Corollary 2 *Enumerate with Simple-Forward has running time $\mathcal{O}((e(\Delta-1))^{k-1} \cdot (\Delta + k) \cdot n/k)$.*

5. Enumeration via Reverse Search

In this section, we describe the reverse search algorithms of [5]. Moreover, we present a small modification of one of the algorithms that leads to an improved delay bound for the case of small k.

5.1. Reverse Search with Dictionary (RwD)

The reverse search method enumerates all solutions by traversing the supergraph $\mathcal{G}$ where every solution corresponds to exactly one node of $\mathcal{G}$. The pseudocode of the first of the two algorithms, *RwD*, is shown in Algorithm 6. During the algorithm each node in $\mathcal{G}$ gets the following labels: *visited* means that the node was visited by the algorithm, *discovered* means that the node was not yet visited but is a neighbor of an already visited node, and a node is *not discovered* otherwise. The algorithm stores all visited and discovered nodes in a set $\mathcal{K}$. Furthermore, all discovered nodes are put in a queue Q.

In the first step, we have to determine an initial connected order-k subgraph in G, so we determine a start node T in $\mathcal{G}$. This can be done with depth-first search. Hence, only node T is assigned with the label discovered and all remaining nodes have the label not discovered. So, initially queue Q and set $\mathcal{K}$ consist of node T.

As long as the queue Q is not empty we do the following: We remove the first node T of Q. Let S denote the solution represented by T. Next, we output S. In a next step, we have to determine all neighbors of T in the supergraph $\mathcal{G}$ and add them to Q if they are not already discovered or visited. To this end, for

Algorithm 6 The *RwD* algorithm.

```
1: procedure $RwD(G, k)$
2:     Queue $Q := \emptyset$, $\mathcal{K} := \emptyset$          $\triangleright$ $\mathcal{K}$ stores the enumerated solutions
3:     for each connected component $C$ in $G$ do
4:         $S :=$ lexicographically largest solution in $C$
5:         $Q.\text{enqueue}(S)$, $\mathcal{K} := \mathcal{K} \cup \{S\}$
6:         while $Q \neq \emptyset$ do
7:             $S := Q.\text{dequeue}()$
8:             output $S$
9:             for vertex $v \in S$ do
10:                $S' := S \setminus \{v\}$
11:                $N :=$ common neighborhood of connected components of $S'$
12:                for vertex $w \in N$ do
13:                    $S'' := S' \cup \{w\}$
14:                    if $S'' \notin \mathcal{K}$ then
15:                        $Q.\text{enqueue}(S'')$, $\mathcal{K} = \mathcal{K} \cup \{S''\}$
```

each vertex v of S we try to find connected subgraphs containing $S' := S \setminus \{v\}$ and avoiding v. Now, the subgraph $G[S']$ may be disconnected. To obtain a connected subgraph, we first determine the connected components of $G[S']$. Afterwards, we determine the set N containing all vertices of $V \setminus S$ that have at least one neighbor in each connected component of $G[S']$. We call this set the *common neighborhood* of the connected components of $G[S']$. In the algorithm of Elbassioni [5], the common neighborhood is determined as follows: Check each vertex of $V \setminus S$ and determine whether it has at least one neighbor in each connected component of S'. Each vertex in N extends $G[S']$ to another solution. To output each solution exactly once, we use the set $\mathcal{K}$, that is, we check whether a solution has already been discovered or visited. A node is added to the queue Q only if it is not discovered.

For this algorithm, we implemented another method to determine the common neighborhood N of the connected components of S'. Instead of checking for each vertex of $V \setminus S$ whether it has at least one neighbor in each connected component of $G[S']$, we compute for each connected C component of $G[S']$ the set of vertices N_C that are from $V \setminus S$ and have at least one neighbor in C. Then, we intersect all of these sets. The resulting set is N. This does not change the worst-case delay of *RwD*.

5.2. Reverse Search with Predecessor (RwP)

This algorithm is almost the same as algorithm *RwD*. The main difference is the following: Instead of having the set $\mathcal{K}$ which stores all visited and discovered nodes (which requires exponential space) it uses a *predecessor function* for solutions. The basic idea of this method in this context is the following: All solutions are sorted lexicographically and each solution has a unique *predecessor*. We determine neighbors of a node S in $\mathcal{G}$. If we determine a new candidate B

for a connected order-k subgraph, we only put B into the queue of discovered nodes if the predecessor of B is S.

Now, we explain this method in more detail: We apply depth-first search for the graph G and every vertex in G is assigned a number when it is discovered by this depth-first search. More precisely, the first vertex is assigned with the highest number $|G|$ and all following vertices get a smaller number. The depth-first ordering of the vertices implies a lexicographical ordering of the solutions. The initial connected order-k subgraph for the enumeration is the lexicographically largest subgraph. Next, we define the predecessor function f: The lexicographically largest subgraph is its own predecessor. For all other nodes, f is defined by $f(S) := (S \setminus \{v\}) \cup \{w\}$ where (v, w) is the lexicographically smallest pair of numbers such that $(S \setminus \{v\}) \cup \{w\}$ is connected and lexicographically smaller than S.

For the correctness, it is necessary to consider an ordering based on depth-first search instead of an arbitrary ordering [5]. As in the *RwD* algorithm, we determine the connected components of $G[S \setminus \{v\}]$ by a union-find structure with path compression and union-by rank and the common neighbors of these components are computed similarly. The pseudocode of this algorithm can be found in Algorithm 7.

The original algorithm used DFS to find all solutions for E-CISE. To achieve the claimed delay it was necessary to distinguish between nodes of odd and even depth. We implemented this algorithm with BFS. In other words, we use a queue to store subgraphs that are discovered but not processed . Hence, our implemented algorithm has a space bound of $\mathcal{O}(m+n+k|\mathcal{G}|)$, instead of a linear one.

A Slightly Improved Delay for RwP for small k. For *RwP*, we obtain an improved delay in the case $k < n/2$. To this end, we decrease the time which is needed to determine the predecessor of a solution. In the original *RwP* algorithm [5] this step needs $\mathcal{O}(k(\Delta + \log k) \min(n - k, k\Delta))$ time. We show that this step can be done in $\mathcal{O}(k^2\Delta)$ time using the new approach for computing the common neighborhood of some connected components that we proposed for *RwD*.

Proposition 2 *The predecessor of a connected order-k subgraph can be determined in $\mathcal{O}(k^2\Delta)$ time.*

Proof. Let S be a connected order-k subgraph. To determine the predecessor of S, we find the vertex $u \in S$ with lowest index and the vertex $v \notin S$ with highest index such that $S \setminus \{u\} \cup \{v\}$ is connected as follows: We test for each vertex w of S in increasing index order whether removing w from S gives the predecessor. Each time, we do the following: In $\mathcal{O}(k \min(k, \Delta))$ time we determine the connected components of $G[S \setminus \{w\}]$. Afterwards, we determine the common neighborhood N of the connected components in $G[S \setminus \{w\}]$ in $\mathcal{O}(k\Delta)$ time as follows: Compute for each connected C component of $G[S \setminus \{w\}]$ the set of vertices N_C that are from $V \setminus S$ and have at least one neighbor in C. Then, intersect all of these sets in $O(k\Delta)$ time; the resulting set is N. Finally,

Algorithm 7 The *RwP* algorithm.

```
procedure RwP(G, k)
    Queue $Q := \emptyset$
    for each connected component $C$ in $G$ do
        $S :=$ lexicographically largest solution in $C$
        $Q.\,\mathrm{enqueue}(S)$
        while $Q \neq \emptyset$ do
            $S := Q.\,\mathrm{dequeue}()$
            output $S$
            for vertex $v \in S$ do
                $S' := S \setminus \{v\}$
                $N :=$ common neighborhood of connected components of $S'$
                for vertex $w \in N$ do
                    $S'' := S' \cup \{w\}$        ▷ Now determine predecessor of $S''$
                    for vertex $u \in S''$ according to ascending index sorting do
                        $S^* := S'' \setminus \{u\}$
                        $M :=$ common neighborhood of connected components of $S^*$
                        $x :=$ vertex with highest index in $M$
                        if $u = w$ and $x = v$ then
                            $Q.\,\mathrm{enqueue}(S^* \cup \{x\})$
```

pick the vertex u with highest index in N. The set $(S \cup \{u\}) \setminus \{w\}$ is the predecessor of S. The overall running time is $\mathcal{O}(k^2\Delta)$ since we consider at most k possibilities for w. □

The bottleneck for the delay of *RwP* is the time spent in the for-loop starting in Line 9. By Proposition 2, each execution of the predecessor check (Lines 14–19) takes $\mathcal{O}(k^2\Delta)$ time. Moreover, this check is called at most $k \cdot \min(n-k, k\Delta)$ times: we consider each vertex in S as a candidate for removal from S and check the common neighborhood. Overall, we arrive at the following.

Corollary 3 *The modified RwP algorithm solves* E-CISE *for any graph G with maximum degree Δ and integer k with polynomial delay $\mathcal{O}(k^3\Delta\min(n-k, k\Delta))$.*

If $k \leq n/2$, then $\min(n-k, k\Delta) \geq k$. Hence, our modified *RwP* algorithm has a better delay than the original *RwP* algorithm in this case. Consequently, combining both approaches and choosing one of both algorithms depending on the value of parameter k leads to the following delay.

Corollary 4 *The modified RwP algorithm solves* E-CISE *for any graph G with maximum degree Δ and integer k with polynomial delay $\mathcal{O}(k^2\min(n-k, k\Delta) \cdot \min(k\Delta, (n-k)(\Delta + \log k)))$.*

6. Polynomial Delay for Enumerating Connected Subgraphs of Size at most k

We now consider the problem of enumerating all connected subgraphs of order *at most k*.

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

Avis and Fukuda described the first algorithm with polynomial delay for B-CISE. This algorithm is based on reverse search and has a delay of $\mathcal{O}(nm)$ [2]. Recently, the *RSSP* algorithm achieved a delay of $\mathcal{O}(n_c)$ [1], where n_c is the order of the largest connected component of G, for the special case when $k = n$, that is, when there is no size restriction. It is possible to adapt this algorithm to the case of arbitrary k which gives a delay bound of $\mathcal{O}(n_c + k)$ for B-CISE. We omit the details of this adaption and instead proceed to show that by adapting *Simple* and *Simple-Forward*, we obtain algorithms with delay $\mathcal{O}(k + \Delta)$ that need $\mathcal{O}(n + m)$ space.

Theorem 2 *Enumerate with Simple or Simple-Forward solves* B-CISE *for any graph G with maximum degree Δ with delay $\mathcal{O}(k + \Delta)$.*

Proof. As shown in Lemmas 4 and 5, *Simple* and *Simple-Forward* both spend $\mathcal{O}(\Delta)$ time at each enumeration tree node before either creating the next child T_{i+1} or returning to the parent T_{i-1} of the current enumeration tree node T_i.

To achieve the claimed delay bound, we adapt each enumeration algorithm as follows: *Enumerate* chooses an arbitrary start vertex v. After the enumeration of all connected induced subgraphs of order at most k containing vertex v, vertex v and all incident edges can be deleted in $\mathcal{O}(\Delta)$ time. Furthermore, we do not use the introduced pruning rules.

We output solutions for B-CISE according to the alternative output rule [17]: Consider a node T_i in the enumeration tree with subgraph set P_i. If i is odd, then output P_i when node T_i is created. Otherwise, if i is even, then output P_i when the algorithm returns to the parent T_{i-1} of node T_i. In the following, a node T_i with i odd is called an *odd* node. Otherwise, node T_i is called *even*.

To prove that this adaption leads to a delay of $\mathcal{O}(k + \Delta)$ for B-CISE, we bound the time between outputting two consecutive solutions for B-CISE. Clearly, the first solution of B-CISE is output after $\mathcal{O}(1)$ time, since the vertex v chosen by *Enumerate* is a solution. Now, assume we just output a solution for B-CISE in some node T_i.

Case 1: Node T_i is odd. Then, node T_i was created directly before P_i was output.

Case 1.1: Node T_i has a further child T_{i+1}. The algorithm needs $\mathcal{O}(\Delta)$ time to construct this node. If node T_{i+1} has a further child T_{i+2}, then the algorithm constructs this odd node in $\mathcal{O}(\Delta)$ time. Since T_{i+2} is odd, the algorithm immediately outputs the corresponding subgraph set P_{i+2} in $\mathcal{O}(k)$ time. Otherwise, node T_{i+1} has no child and we return in $\mathcal{O}(\Delta)$ time to node T_i. Since T_{i+1} is even, the algorithm then outputs the subgraph set P_{i+1} in $\mathcal{O}(k)$ time.

Case 1.2: Node T_i has no further child. Hence, the algorithm returns in $\mathcal{O}(\Delta)$ time to the even node T_{i-1}. If node T_{i-1} has a further child, the algorithm constructs in $\mathcal{O}(\Delta)$ time the next child T_i' which is odd. Hence, subgraph set P_i' is output in $\mathcal{O}(k)$ time directly after the construction of T_i'. Otherwise, if node T_{i-1} has no further child, the algorithm returns to its parent T_{i-2} and since node T_{i-1} is even, the subgraph set P_{i-1} is then output in $\mathcal{O}(k)$ time.

Case 2: Node T_i is even. Then, P_i was output directly before the algorithm returns to the parent T_{i-1} of T_i in $\mathcal{O}(\Delta)$ time.

Case 2.1: Node T_{i-1} has a further child T_i'. The algorithm constructs this node in $\mathcal{O}(\Delta)$ time. If node T_i' has a child, the algorithm computes the first child T_{i+1}' of node T_i' in $\mathcal{O}(\Delta)$ time. Since node T_{i+1}' is odd, the algorithm immediately output the subgraph set P_{i+1}' in $\mathcal{O}(k)$ time. Otherwise, the even node T_i' has no children, and the algorithm outputs the subgraph set P_i' in $\mathcal{O}(k)$ time directly before returning to T_{i-1} .

Case 2.2: Node T_{i-1} has no further child. Hence, the algorithm constructs in $\mathcal{O}(\Delta)$ time its even parent T_{i-2}. If T_{i-2} has a further child, the algorithm computes in $\mathcal{O}(\Delta)$ time its next child T_{i-1}'. Since T_{i-1}' is odd, the algorithm outputs P_{i-1}' in $\mathcal{O}(k)$ time. Otherwise, node T_{i-2} has no further child. Since T_{i-2} is even, the algorithm outputs the subgraph set P_{i-2} in $\mathcal{O}(k)$ time directly before returning to T_{i-3}.

In all cases, the delay between two consecutive outputs of a solution for B-CISE is $\mathcal{O}(k + \Delta)$. □

Since $\Delta < n_c$, *Simple* and *Simple-Forward* give improved delay bounds for small k and Δ while achieving the same delay bound as *RSSP* in the previously considered case $k = n_c$.

7. An Experimental Comparison

We now present an experimental comparison of *Simple*, *Pivot*, *Simple-Forward*, *Exgen*, *Kavosh*, *RwD*, *RwP*, and *BDDE*. For a detailed description of *Exgen*, *Kavosh*, and *BDDE*, refer to the appendix.

7.1. Experimental Setup

We implemented our algorithms[2] in Python 3.6.8 using the graph data structure *igraph*; the core modules of igraph are written in C.[3] Each experiment was performed on a single thread of an Intel(R) Xeon(R) Silver 4116 CPU with 2.1

[2]The source code of our implementation is available at `https://box.uni-marburg.de/index.php/s/RGTekb95tTEOCRC`

[3]`http://igraph.org/python/`

GHz, 24 CPUs and 128 GB RAM. For the *BDDE* algorithm, we used NetworkX (`https://networkx.github.io/`) as graph library for building the enumeration tree. This choice is due to the fact that the graph modification operations of igraph are inefficient. The reported running times include the time needed to write the output to the hard drive.

As benchmark data set we used 30 sparse social, biological, and technical networks obtained from the Network Repository [16], KONECT [14], and the 10th DIMACS challenge [3]. We group the real-world instances into three subsets of size 10: small networks with $n < 500$, medium-size networks with $500 \leq n < 5000$ and large networks with $n \geq 5000$. An overview of the instance properties and names is given in Table 1.

In addition, we performed experiments on random instances generated in the $G(n, p)$ model where n is the number of vertices and each edge is present with probability p. We generated one instance for each $n \in \{100, 200, \dots, 1000\}$ and $p \in \{0.1, 0.2\}$. For each network, we considered each $k \in \{3, 4, \dots, 10\}$ and $k \in \{n_c - 1, n_c - 2, n_c - 3\}$. For each instance, we set a running time threshold of 10 minutes.

7.2. Implementation Details

To speed up the enumeration for large k for E-CISE, we implemented the following pruning rule for *Exgen*, *Kavosh*, *Simple* and *Simple-Forward*: We store the order of each connected component of G. Let T_i be a node in an enumeration tree of one of these four algorithms with subgraph set P_i and set F_i of forbidden vertices. Without loss of generality, assume that the vertices of P_i are contained in a connected component C of size $|C|$ of G. To avoid some unnecessary recursions, we check if $|C| - |F_i| < k$. If yes, we return in T_i to its parent T_{i-1}. Now, we prove that this pruning rule is correct.

Lemma 6 *Let T_i be a node in an enumeration tree of Exgen, Kavosh, Simple and Simple-Forward. Let P_i be the corresponding subgraph set in a connected component C of G and let F_i be the corresponding set of forbidden vertices. If $|C| - |F_i| < k$, then no subsequent recursive call in node T_i leads to an interesting leaf.*

Proof. For each subsequent child T_j with forbidden vertex set F_j we have $F_i \subset F_j$. In other words, in node T_j no vertex of F_i can be part of the subgraph set P_i. Since $|C| - |F_i| < k$, adding all possible vertices of component C to the subgraph set does not yield to an induced subgraph of size k. Thus, we can abort this branch and return to the parent T_{i-1} of node T_i. $\square$

In the following we refer to this rule as the *k-component* rule. This rule does *not* improve the delay of these algorithms. We now give some further details on how we implemented the algorithms.

We implemented *Exgen*, *Kavosh*, *Simple* and *Simple-Forward* recursively and iteratively. Preliminary experiments showed that for each k the iterative variants of the algorithms are at least a factor of two faster compared to the

Table 1: Networks used for our experiments.

Size	Name	n	m
Small	moreno-zebra	27	111
	ucidata-zachary	34	78
	contiguous-usa	49	107
	dolphins	62	159
	ca-sandi-auths	86	124
	adjnoun_adjacency	112	425
	arenas-jazz	198	2 742
	inf-USAir97	332	2 126
	ca-netscience	379	914
	bio-celegans	453	2 025
Medium	bio-diseasome	516	1 188
	soc-wiki-Vote	889	2 914
	arenas-email	1 133	5 451
	inf-euroroad	1 174	1 417
	bio-yeast	1 458	1 948
	ca-CSphd	1 882	1 740
	soc-hamsterster	2 426	16 630
	inf-openflights	2 939	15 677
	ca-GrQc	4 158	13 422
	inf-power	4 941	6 594
Large	soc-advogato	6 541	51 127
	bio-dmela	7 393	25 569
	ca-HepPh	11 204	117 619
	ca-AstroPh	17 903	196 972
	soc-brightkite	56 739	212 945
	coAuthorsCiteseer	227 320	814 134
	coAuthorsDBLP	299 067	977 676
	coPapersCiteseer	434 102	16 036 720
	soc-twitter-follows	404 719	713 319
	coPapersDBLP	540 486	15 245 729

recursive variant. This factor increases for large k. Hence, we only compare the iterative variants. Furthermore, preliminary experiments showed that *Kavosh* outperforms *Exgen* on almost every instance. Hence, we do not report the results for *Exgen*. Each of *Kavosh*, *Simple* and *Simple-Forward* is implemented with and without the k-component rule, and the corresponding pruning rules.

Enumerate. We did not implement the removal of the vertex v after the call of *Enum-Algo* as described in the proof of Lemma 2, because *igraph* was relatively inefficient with respect to graph modifications. Instead we assign an index to every vertex and process the vertices in descending index order. Then, in the call of *Enum-Algo* where we enumerate all solutions containing v, we remove all vertices with higher index than v when constructing the set of neighbors of any vertex.

Simple. The set P is implemented as a list. When creating a new node of the enumeration tree in *Simple*, we add the new vertex u to this list. When the algorithm returns to its parent, we remove the vertex u from the list. The extension set is implemented as described in Lemma 4. We have two additional arrays. The first array B is used for applying the pruning rule described in Lemma 3, that is, it is used for implementing the Boolean flag hasIntLeaf. When we create a new child T_{i+1} of a node T_i, the values $B[i]$ and $B[i+1]$ are set to 0. If an interesting leaf is detected, each entry of B is set to 1. If a boring leaf is detected, the last vertex from the subgraph set is removed, until a node T_j is reached with $B[j] = 1$. In the second array we store the orders of the connected components of the graph G to test the k-component rule.

Simple-Forward. We implemented the old variant of *Pivot* described in Algorithm 3, the adapted variant of *Pivot* described in Algorithm 4 and *Simple-Forward.* For *Simple-Forward* we implemented the data structures described in Lemma 5. As in *Simple*, we have an additional array to store the result of testing the pruning rule and an additional array to store the orders of the connected components of the graph G to test the k-component rule.

Kavosh. We implemented *Kavosh* as described in Algorithm 9. To compute each $M \subseteq X$ for a fixed set X we used *itertools.* The original implementation used the *revolving door ordering* for this step [13, 8] . As in *Simple* and *Simple-Forward*, we have an additional array to store the result of testing the pruning rule described in Proposition 4 and an additional array to store the orders of the connected components of the graph G to test the k-component rule.

RwD and RwP. We implemented our versions of *RwD* and *RwP* as described in Section 5.

7.3. Results for E-CISE

In each plot only the fastest variant of *Simple*, *Simple-Forward*, and *Kavosh* with and without the k-component rule and with and without the corresponding pruning rule is plotted. The left part of Figure 2 shows the result for $k \in \{3, \ldots, 10\}$. There, only the new versions of *RwD* and *RwP* are plotted since they are roughly 20% faster than the corresponding original algorithms. There was almost no difference in the running times of *Simple* with and without the k-component rule and with and without the pruning rule described in Lemma 3. *Simple* without the k-component rule and without the pruning rule was slightly faster than the other three variants of *Simple.* For *Kavosh* we obtained similar results. All four variants (with and without the k-component rule and with or without the pruning rule described in Proposition 4) have almost the same running time, with the plain version of *Kavosh* without the pruning rule and without the k-component rule being the fastest. We obtained similar results for *Simple-Forward.* To conclude, for small k the plain versions of the algorithms are the fastest.

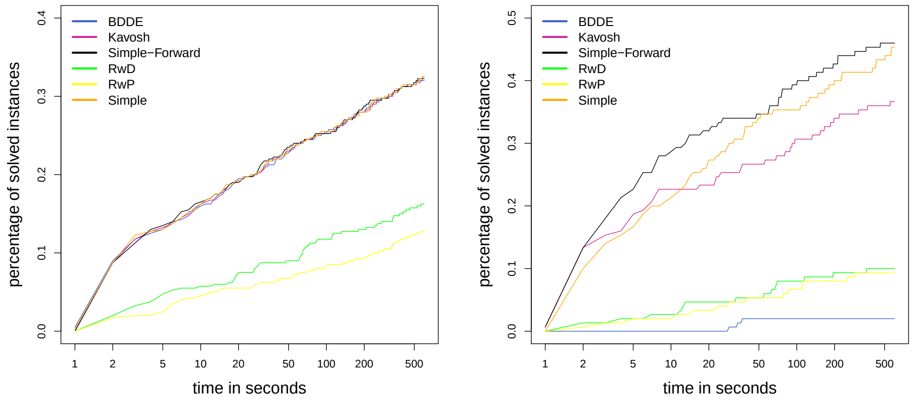

Figure 2: Comparison for E-CISE. Left: Comparison for $k \in \{3, \dots, 10\}$. Right: Comparison for $k \in \{n_c - 1, n_c - 2, n_c - 3\}$ where n_c is the order of the largest connected component in the graph.

Table 2: Average running times for E-CISE on instances that are solved by all algorithms for small k.

Category	*BDDE*	*Kavosh*	*Simple*	*Simple-Forward*
Small	40.9	39.3	37.5	34.2
Medium	91.4	87.7	86.2	78.5
Large	151.3	151.9	153.9	149.4

RwD is five times faster than *RwP*. Furthermore, all instances solved by *RwD* within the time limit of 600 seconds, were solved by the other four algorithms within 15 seconds. In other words, *RwD* is 40 times slower than the other four algorithms. There was almost no difference in the running times for *BDDE*, *Kavosh*, *Simple* and *Simple-Forward*. *Simple-Forward* is slightly faster than the other three algorithms. *Simple* solved 130 out of 400 instances, one more than *Simple-Forward*. The average running time of a solved instance for each algorithm in our comparison can be found in Table 2. On average, *Simple-Forward* is the fastest algorithm in all three categories. Hence, for small k, one should use the plain version of *Simple-Forward*.

The right part of Figure 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. Again, only the new versions of *RwD* and *RwP* are plotted, since they are slightly faster than the corresponding old versions of the algorithms. *Kavosh*, *Simple* and *Simple-Forward* without the corresponding pruning rules and without the k-component rules and *BDDE* solve only three out of 150 instances. Furthermore, many memory errors occurred in *BDDE*. This is not surprising since *BDDE* stores huge parts of the enumeration tree to copy branches. *RwD* has

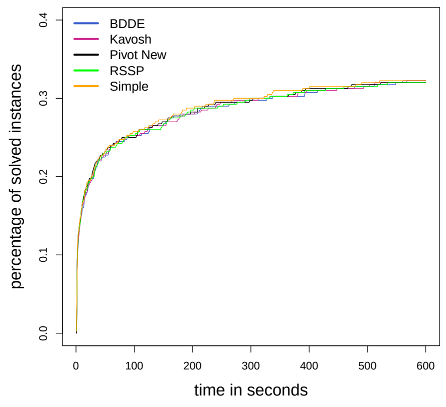

Figure 3: Comparison for B-CISE for $k \in \{3, \ldots, 10\}$.

no memory errors since the cutoff time of 600 seconds was so small that *RwD* only enumerated a few solutions.

Kavosh with the pruning rule and without the k-component rule is much faster than *Kavosh* without both rules. We cannot estimate a concrete speed-up factor since *Kavosh* without both rules solves only three instances. The variant of *Kavosh* with the k-component rule and with the pruning rule is roughly 30% faster than *Kavosh* with the k-component rule and without the pruning rule. Both versions are 20 times faster than *Kavosh* only with the pruning rule. *Kavosh* without the pruning rule and with the k-component rule is only slightly slower than *Kavosh* with the pruning rule and with the k-component rule. *Simple* with the pruning rule and without the k-component rule is much faster than the plain version. Similar to *Kavosh* we cannot estimate a speed-up factor since the plain version solved only very few instances. *Simple* with the k-component rule and with the pruning rule is slightly faster than *Simple* with the k-component rule and without the pruning rule. Both variants are roughly 30 times faster than *Simple* only with the pruning rule. We obtained similar results for *Simple-Forward* as for *Simple*. In our experiments the k-component rule gives a much higher speedup than the corresponding pruning rules since k was at least $n_c - 3$. To conclude, for large k the versions of the algorithms with the pruning rule and with the k-component rule are the fastest.

RwD is roughly two times faster than *RwP*. All instances solved by *RwD* are solved by *Kavosh* in less than two seconds. *Simple* is roughly three times faster than *Kavosh* and *Simple-Forward* is roughly two times faster than *Simple*. *Simple-Forward* solved 69 out of 150 instances. Hence, for large k, one should use *Simple-Forward* with the pruning rule and the k-component rule.

Table 3: Average running times for B-CISE on instances that are solved by all algorithms for small k.

Category	*BDDE*	*Kavosh*	*RSSP*	*Simple*	*Simple-Forward*
Small	45.9	45.4	44.6	39.7	44.9
Medium	76.5	75.3	62.4	68.6	70.7
Large	141.6	145.4	159.0	141.7	140.6

7.4. Results for B-CISE

We tested the plain versions of *Kavosh*, *Simple* and *Simple-Forward* for B-CISE. We compare these three algorithms with *BDDE* and *RSSP* [1]. *BDDE* is slightly slower than *Kavosh*, which is slightly slower than *RSSP*. *Simple-Forward* is slightly faster than *RSSP* and *Simple* is slightly faster than *Simple-Forward*. Overall, *Simple* is roughly 20% faster than *BDDE*. *Simple* solved 129 out of 400 instances. The average running time of a solved instance for each algorithm in our comparison can be found in Table 3. *Simple* is the fastest algorithm for small instances, *RSSP* is the fastest algorithm for medium instances and *Simple-Forward* is the fastest algorithm for large instances. To conclude: To solve B-CISE one should use *Simple* since it is overall the fastest algorithm for this task.

8. Outlook

We have used the implementations of *Simple*, *Simple-Forward*, and *Kavosh* as the basic enumeration methods in an efficient generic algorithm for fixed-cardinality optimization problems in graphs [9]. For this application, *Simple-Forward* turned out to be the most efficient implementation. Concerning future work for E-CISE, one goal is to further improve the delay bounds. It seems that a bound of $\mathcal{O}(k\Delta)$ is within reach. Moreover, it would be interesting to study the algorithm of Ferreira [7] from the viewpoint of bounded delay. Since the depth of the enumeration tree in this algorithm is not bounded by k, one would need a different analysis to prove delay bounds for this algorithm. A further interesting route to obtain better enumeration algorithms could be to replace the maximum degree Δ by provably smaller parameters such as the degeneracy of the graph. Finally, it would be interesting to determine systematically whether putting restrictions on the connected induced subgraphs that shall be enumerated gives better delay bounds. For example, can we achieve a substantially better delay bound when we are only interested in enumerating induced paths of length exactly k?

References

[1] Alokshiya, M., Salem, S., Abed, F., 2019. A linear delay algorithm for enumerating all connected induced subgraphs. BMC Bioinformatics 20-S, 319:1–319:11.

[2] Avis, D., Fukuda, K., 1996. Reverse search for enumeration. Discrete Applied Mathematics 65, 21–46.

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

[4] Bollobás, B., 2006. The Art of Mathematics – Coffee Time in Memphis. Cambridge University Press.

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

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

[7] Ferreira, R., 2013. Efficiently Listing Combinatorial Patterns in Graphs. Ph.D. thesis. Dipartimento di Informatica, Università degli Studi di Pisa. Available at https://arxiv.org/abs/1308.6635.

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

[9] Komusiewicz, C., Sommer, F., 2020. FixCon: A generic solver for fixed-cardinality subgraph problems, in: Proceedings of the Symposium on Algorithm Engineering and Experiments, (ALENEX '20), SIAM. pp. 12–26.

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

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

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

[13] Kreher, D.L., Stinson, D.R., 1998. Combinatorial Algorithms: Generation, Enumeration, and Search (Discrete Mathematics and Its Applications). CRC Press.

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

[15] Maxwell, S., Chance, M.R., Koyutürk, M., 2014. 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), Springer. pp. 171–182.

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

[17] Uno, T., 2003. Two general methods to reduce delay and change of enumeration algorithms. Technical Report. National Institute of Informatics.

[18] Uno, T., 2015. Constant time enumeration by amortization, in: Proceedings of the 14th International Symposium on Algorithms and Data Structures (WADS '15), Springer. pp. 593–605.

[19] Wernicke, S., 2006. Efficient detection of network motifs. IEEE/ACM Transactions on Computational Biology and Bioinformatics 3, 347–359.

[20] Wernicke, S., Rasche, F., 2006. FANMOD: a tool for fast network motif detection. Bioinformatics 22, 1152–1153.

Algorithm 8 The *Exgen* algorithm; the initial call is *Exgen*$(\{v\}, \emptyset, \emptyset)$.

```
procedure Exgen($P, S, F$)
    if $|P \cup S| = k$ then
        output $P \cup S$
        return True
    hasIntLeaf := False
    $p$ := choose element of $P$
    $X := N(p) \setminus (P \cup S \cup F)$
    for $i$ from $k - |P \cup S|$ down to 0 do
        hasIntLeaf$_i$ := False
        for each $M \subseteq X$ such that $|M| = i$ do
            if Exgen($P \cup M \setminus \{p\}, S \cup \{p\}, F \cup X$ = True then
                hasIntLeaf$_i$ := True; hasIntLeaf := True;
        if hasIntLeaf$_i$ = False then
            return hasIntLeaf ▷ Stop recursion if no new solution was found
    return hasIntLeaf
```

9. Appendix: Pseudocode and Analysis of Further Algorithms Included in the Experimental Comparison

In this section, we the describe the other algorithms that we include in our experimental comparison. For *Exgen* and *Kavosh*, we provide a pruning rule that will be useful in the case where k is large. For *Kavosh* and *BDDE*, we provide the the first running time bounds based on Lemma 1. Afterwards, we describe the *RSSP* algorithm [1].

9.1. Exgen

The *Exgen* algorithm is a variant of *Pivot*. The pseudocode of *Exgen* (including the new pruning rule (Lines 13 and 14)) is shown in Algorithm 8. The *Exgen* algorithm was described by Komusiewicz and Sorge [11] since, compared to *Pivot*, it was easier to bound the number of recursive calls by the number of E-CISE solutions. The sets P, S, and F are defined as in Section 4, that is, P contains the vertices of the subgraph set whose neighbors may still be added, S contains the other vertices of the subgraph set, and F contains the vertices which may not be added to the subgraph anymore. In each recursive call, we choose one pivot vertex p from the set P and determine the set X of its neighbors which are not in $P \cup S \cup F$. Next, we move the pivot vertex p from set P to set S, since in the recursive calls of *Exgen*, no further neighbors of p may be added to the subgraph. Afterwards, for each subset M of X which has at most $k - |P \cup S|$ vertices, we call *Exgen* recursively with M added to the subgraph set P (the size bound comes from the fact that we search for subgraphs of order k only). Note that $M = \emptyset$ is a valid choice, since a solution does not need to contain a neighbor of p.

Next, we introduce a similar pruning rule as we did it for *Simple* and *Pivot*.

To apply the rule, we enumerate all subsets M of the possible neighbors X in the following way: We start by creating children for subsets of size $i := k - |P \cup S|$. If none of the children which correspond to these choices for M leads to an interesting leaf, then we prune the enumeration tree, that is, we return to the parent of the current enumeration tree node. Otherwise, we decrease the size i of the subsets M of X that we want to generate by one and continue.

Proposition 3 *The pruning rule performed in Lines 13–14 of Algorithm 8 is correct.*

Proof. Consider a node T in the enumeration tree with vertex sets P_T, S_T, and F_T and current pivot vertex p. Furthermore, let X_T be the set of neighbors of vertex p which are not in $P_T \cup S_T \cup F_T$. Now assume that for some size m, for each size-m set $M \subseteq X_T$, the recursive call for $P_T \cup M$ does not output any solution. Now consider a child R of T for which $|P_R| - |P_T| = m - 1$ holds, that is, P_R is obtained by adding a set M_R of size $m - 1$ to P_T. By the choice of m, there exists some child Q of T obtained by adding M_Q to P_T such that $M_Q \setminus M_R = \{v\}$ for some vertex v and Q does not lead to an interesting leaf. Now suppose that R leads to an interesting leaf. Consider a vertex $u \notin M_R \cup P_T$ that is a leaf of some spanning tree of the corresponding subgraph. Removing u and adding the vertex v to this subgraph gives a connected subgraph that has to be enumerated in the enumeration subtree rooted at Q. This contradicts that Q does not lead to an interesting leaf. Hence, node R cannot lead to an interesting leaf. Consequently, no child obtained by adding a set M of size $m-1$ to P_T leads to an interesting leaf. By applying this argument inductively, we have that no child obtained by adding a set M of size smaller than m leads to an interesting leaf. $\square$

This pruning rule is much weaker since it does not lead to a polynomial delay: Consider a node T in the enumeration tree. Furthermore, let X_T be the set of possible neighbors, where $|X_T| \leq \Delta$. The first time that we may return to the parent of T is if no branch for $M \subseteq X$ with $|M| = k - |P \cup S| - 1$ leads to an interesting leaf. There may be $\Theta(\Delta^{k-1})$ possibilities for choosing M which is not polynomial if k is not a constant. Nevertheless, the pruning rule proved very useful in the case of large k.

The following running time bound was observed by Komusiewicz and Sorge [11] and is stated only for the sake of comparison with the other running time bounds.

Theorem 3 ([11]) *Enumerate with Exgen enumerates each connected subgraph of order at most k exactly once and has a worst-case running time of $\mathcal{O}((e(\Delta - 1))^{k-1} \cdot (\Delta + k) \cdot n/k$ time.*

9.2. Kavosh

The next algorithm in our comparison is *Kavosh* [8]. It was introduced for the computation of network motifs and is in some sense a mixture of *Simple* and *Exgen*; the pseudocode is shown in Algorithm 9.

Algorithm 9 The *Kavosh* algorithm; the initial call is *Kavosh*$(\{v\}, \emptyset, \emptyset)$.

```
procedure Kavosh($P, S, F$)
    if $|P \cup S| = k$ then
        output $P \cup S$
        return True
    hasIntLeaf := False
    $X := N(P) \setminus (P \cup S \cup F)$
    ▷ Add at least one vertex to the subgraph
    for $i$ from $k - |P \cup S|$ down to 1 do
        hasIntLeaf$_i$ := False
        for each $M \subseteq X$ such that $|M| = i$ do
            if Kavosh($M, S \cup P, F \cup X$) then
                hasIntLeaf$_i$ := True; hasIntLeaf := True;
        if hasIntLeaf$_i$ = False then
            return hasIntLeaf ▷ Stop recursion if no new solution was found
    return hasIntLeaf
```

In each enumeration tree node, we have the sets P, S, and F as defined in *Pivot* and *Exgen*, that is, P contains the vertices of the subgraph set whose neighbors may still be added, S contains the other vertices of the subgraph set, and F contains the vertices which may not be added to the subgraph anymore. The basic idea of *Kavosh* is that instead of choosing one pivot vertex, we extend the subgraph set by creating all possible subsets of the neighborhood of P. In other words, we now determine the set X of neighbors of P which are not in $P \cup S \cup F$. Then, for each non-empty set $M \subseteq X$ of size at most $k - |P \cup S|$, we call *Kavosh* recursively with M being the vertex set whose neighbors are now considered, P being added to S, and with X being added to F; in this child we aim to enumerate those subgraphs extending $P \cup S$ that contain all vertices of M and no further vertices of X.

We provide a similar pruning rule for *Kavosh* as we did it for *Exgen*. That is, we create the sets M in decreasing order of size; if for some size M, we do not obtain interesting leafs for any of the recursive calls, then we return immediately to the parent of the current enumeration tree node. The proof of correctness of this pruning rule follows by similar arguments as the proof for *Exgen*.

Proposition 4 *The pruning rule performed in Lines 13–14 of Algorithm 9 is correct.*

Proof. Consider a node T in the enumeration tree with vertex sets P_T, S_T, and F_T. Furthermore, let X_T be the set of neighbors of P_T which are not in $P_T \cup S_T \cup F_T$. Furthermore, consider some m such that for each subset M of X_T of size m the recursive call of *Kavosh*$(M, P_T \cup S_T, F_T \cup X_T)$ does not lead to an interesting leaf. We show that any recursive call for a set $M' \subseteq X$ of size less than m does not lead to interesting leaves. Let R be a child of T which was obtained by such a recursive call and assume R leads to an interesting leaf. Clearly, node T contains a child L created by the recursive call $Kavosh(M, P_T \cup S_T, F_T \cup X_T)$ where $M' \subsetneq M$ where $|M| = m$. Since L does not lead to an interesting leaf and $P_R \subset P_L$, $S_L = S_R$, and $F_L = F_R$ we have that R cannot lead to an interesting leaf. □

As in the case of *Exgen*, this pruning rule does not make *Kavosh* a polynomial delay algorithm for E-CISE. For example, consider the star graph with one vertex v of degree $n-1$ and assume v is added in the root of the enumeration tree. After trying all subsets of size $k-1$ of $N(v)$, the algorithm tries to add each subset of size $k-2$, none of which gives a solution. The number of these subsets is $\binom{n-1}{k-2}$ which is not polynomial if k is not a constant. Hence, there is a superpolynomial delay between the output of the last solution and the termination of the algorithm. Nevertheless, in the experiments, this pruning rule proved to be critical in the case of large k.

We conclude by bounding the overall running time of *Enumerate* with *Kavosh*.

Lemma 7 *Enumerate with Kavosh has a worst-case running time of* $\mathcal{O}((e(\Delta - 1))^{(k-1)} \cdot \Delta \cdot n)$.

Proof. *Enumerate* with *Kavosh* enumerates each connected subgraph of order at most k exactly once [8]. This implies that for each pair of different nodes T and Q in the enumeration tree with the respective sets P_T, S_T, P_Q, and S_Q we have that $P_T \cup S_T \neq P_Q \cup S_Q$. That is, each enumeration tree node corresponds to a different connected subgraph of order at most k. According to Lemma 1, the overall number of nodes in the enumeration trees over all calls to *Kavosh* is $\mathcal{O}((e(\Delta - 1))^{(k-1)} \cdot (n/k))$.

It remains to bound the time per node T in the enumeration tree. Determining the neighbors X_T of the subgraph set P_T needs $\mathcal{O}(k\Delta)$ time since $|P_T| \leq k$ and each vertex has up to Δ neighbors. For each subset M of X_T of size at most $k - |P_T \cup S_T|$ we make a recursive call with parameters $M, P_T \cup S_T, F_T \cup X_T$. The recursive call includes computing the three sets which can be done in $\mathcal{O}(k\Delta)$ time per call. By charging this running time to the corresponding child in the enumeration tree, we obtain a running time of $\mathcal{O}(k\Delta)$ per enumeration tree node. Outputting a solution needs $\mathcal{O}(k)$ time. Hence, we obtain a delay of $\mathcal{O}(k\Delta)$ per node in the enumeration tree. The overall running time follows. □

9.3. BDDE: Breadth-First Discovery, Depth-First Extension

A further, seemingly more involved, enumeration algorithm in our comparison is *BDDE* [15]. The pseudocode of *BDDE* is shown in Algorithm 10. The idea is to start with a vertex v and to enumerate all subgraphs of order at most k such that each connected subgraph S that contains v is enumerated before S' if $S \subseteq S'$. This algorithm was used to enumerate all connected subsets S such that $f(S) \geq t$ for a given function f and a fixed threshold t. In our case of enumerating all connected subgraphs of order k, $f(S) = |S|$ and $t = k$.

To enumerate for a given vertex v all connected induced subgraphs of order at most k containing v in the order of their inclusion relation, the algorithm will

Algorithm 10 The *BDDE* algorithm. Here, $Tree$ is the enumeration tree which is initially empty. The initial call is $Depth([\,], v, [\,])$.

```
procedure $Depth(P, u, C)$
    $X := N(u) \setminus N[P]$
    $P := P \cup \{u\}$
    $x := Tree.$number_of_vertices
    $Tree.$add_vertex($x, id = u$)
    ▷ The $id$ of $x$ is the vertex $u \in V(G)$ represented by $x$
    if $|P| = k$ then
        output $P$
        return $-1$
    $C' := [\,]$
    for $y \in C$ do
        $x' := Breadth(P, y, X)$
        if $x' \neq -1$ then
            $Tree.$add $\_edge(x, x')$
            $C'.$append($x'$)
    for $z \in X$ do
        $x' := Depth(P, z, C')$
        if $x' \neq -1$ then
            $Tree.$add_edge($x, x'$)
            $C'.$append($x'$)
    return $x$

procedure $Breadth(P, x, X)$
    if $Tree.$vertex($x$)[$id$] $\in X$ then
        return $-1$
    $P := P \cup \{Tree.$vertex($x$)[$id$]$\}$
    if $|P| := k$ then
        output $P$
        return $-1$
    $x' := Tree.$number_of_vertices
    $Tree.$add_vertex($x', id = id(x)$)
    for $y \in Tree.$successor($x$) do
        $x^* := Breadth(P, y, X)$
        if $x^* \neq -1$ then
            $Tree.$add_edge($x', x^*$)
    return $x'$
```

copy parts of the enumeration tree. The algorithm consists of two functions: *Depth* to discover new vertices, and *Breadth* to copy parts of the enumeration tree.

In the enumeration tree T each node has a label *id* which represents exactly one vertex in the graph G and each path from the root to a another node in the tree represents a connected subgraph in G. Hence, the set of leaves in depth k corresponds to the set of connected subgraphs of order k. Clearly, the root has id v. Consider node x with id u in the enumeration tree. The set P is referred to as the set of ids of the nodes in the enumeration tree which are predecessors of x. The *exclusive neighborhood* $X(u) := N(u) \setminus N[P]$ is the set of neighbors of u in the graph G which are neither in P nor neighbors of the vertices in P and not in the set P.

Clearly, a sibling s of a node t in the enumeration tree has many children similar to children of t. The algorithm uses this fact as follows: Let r be a child of s. If r is not an exclusive neighbor of t, copy the subtree which is rooted at r and call this copy r'. Next, make r' a child of t. This will be achieved by the procedure *Breadth*. Furthermore, procedure *Depth* will be used to add new edges and to discover new vertices.

Now, we describe the procedures *Depth* and *Breadth* in further detail.

Depth has three parameters: The set P, which is the set of all ids of predecessors of p, the vertex u, which is the actual vertex we consider, and a list C which stores all successors of nodes representing the last vertex in P. First, we make a new node x in the enumeration tree with the id u and we create a new list C' for the branches of x. Second, with *Breadth* we copy the branches stored in C as new branches of x. If we copy a branch x' successfully we add an edge from x to x' and append x' to C'. Third, for each vertex $z \in X(u)$ we call *Depth*. If we created a branch x' successfully, we create an edge from x to x' and append x' to C'. In the end we return x.

Breadth has three parameters: The set P is the set of all ids of predecessors of node x, the node x which is a sibling of the node this function will create, and the set X is the exclusive neighborhood of the last node in the set P created from the function *Depth*. The set X is needed to avoid enumerating some subgraphs twice. If the id of x is in the set X, we return. Otherwise, we create a new node x' which has the same id as node x. Then for each successor y of x in T, we call *Breadth* recursively. If this call returns a node x^*, we make an edge from x' to x^*. In the end of this function, we return x'.

Now we bound the running time of *BDDE*.

Proposition 5 *Enumerate with BDDE has a worst-case running time of* $\mathcal{O}((e(\Delta-1))^k \cdot k \cdot \Delta \cdot n)$ *time.*

Proof. It was shown that for two nodes T and S in the enumeration trees of *Enumerate* with *BDDE* the vertex sets P_T and P_S are different [15, Lemma 5]. In other words, each subgraph of size at most k is enumerated exactly once.

We now bound the running time for each call of the procedures *Depth* and *Breadth*. We start with *Depth* with parameters P, u, and C. Since each vertex has degree at most Δ, the size of the exclusive neighborhood X of u is bounded by $\mathcal{O}(\Delta)$. By marking each vertex of $N[P]$ with a color, we can determine the exclusive neighborhood X in $\mathcal{O}(\Delta)$ time. In each recursive call of *Depth*, we enlarge C by at least Δ, the size of X. Hence, there are at most $k\Delta$ recursive calls of *Breadth*. Each recursive call of *Breadth* includes computing the new vertex which can be done in $\mathcal{O}(\Delta)$ time. We charge this running time to the corresponding child in the enumeration tree. Furthermore, there are at most Δ calls of *Depth*. Similar, each recursive call of *Depth* includes computing the new vertex which can be done in $\mathcal{O}(\Delta)$ time. Again, we charge this running time to the corresponding child. Overall, *Depth* needs $\mathcal{O}(\Delta)$ time to construct the next child.

Now, we consider *Breadth* with parameters P, x, and X. Checking existence of the id of node x in the set X can be done in constant time. Since each vertex in the graph has degree at most Δ, there are at most Δ recursive calls for *Breadth*. Again, we charge the running time of each recursive call to the corresponding children. Overall, *Breadth* needs $\mathcal{O}(\Delta)$ time to construct the next child.

Since each solution can be output in $\mathcal{O}(k)$ time, we obtain a running time of $\mathcal{O}(k + \Delta)$ per enumeration tree node. The overall running time follows. □

Since the basic idea of *BDDE* is to start with a vertex v and to enumerate all connected subgraphs of order at most k such that each connected subgraph S which contains v is enumerated before S' if $S \subseteq S'$, *BDDE* does *not* yield a polynomial delay for E-CISE. For example, consider the case $k = n$ and G is complete. Then *BDDE* enumerates each connected induced subgraph of order less than k, before enumerating the graph G. Clearly, these are exponentially many.

9.4. RSSP

The last algorithm in our comparison is *RSSP* [1]. It was introduced to enumerate all connected induced subgraphs and to mine all maximal cohesive subgraphs. The pseudocode is shown in Algorithm 11.

Each enumeration tree node T_i consists of a subgraph set P and a set X of vertices which can be added to P. Fix an ordering on the vertices of V. For $x, y \in P$, we let $\text{dist}(x, y)$ denote the length of a shortest path from x to y in $G[P]$. Let $z \in P$ be the vertex with smallest label with respect to the fixed ordering, and let $W \subseteq P$ be the set of vertices of P which have the longest shortest path in $G[P]$ to vertex v. Let $u \in W$ be the vertex with maximal index. The subgraph set of the parent T_{i-1} of node T_i is $P \setminus \{u\}$. Let $w \in N(P)$. If $\text{dist}(z, w) > \text{dist}(u, w)$ or $\text{dist}(z, w) = \text{dist}(u, w)$ and the index of w is larger than the index of u, then node T_i has a child with subgraph set $P \cup \{w\}$. To obtain the children T_{i+1} efficiently, all distances to vertex v are stored. Due to the strict child definition, a vertex $x \in X_i$ cannot be a valid candidate to expand P_{i+1}. To this end, after adding vertex p_i to P_i to obtain P_{i+1}, all vertices in X_i are checked if they are still a valid candidate to expand P_i. Because of this step, a similar pruning rule as introduced for *Simple* and *Pivot* is not possible since the ordering of X changes during the exploration of the enumeration tree.

Algorithm 11 The *RSSP* algorithm; the initial call is $RSSP(\{v\}, N(v))$.

```
1: procedure RSSP($P, X$)
2:     $s$ := first vertex of $P$
3:     $z$ := last vertex of $P$
4:     if $|P| = k$ then
5:         output $P$
6:         return
7:     while $X \neq \emptyset$ do
8:         $u$ := choose first vertex from $X$
9:         delete $u$ from $X$
10:        remove invalid candidates from X
11:        for $w \in N(u)$ do
12:            if $\mathrm{dist}(s, w) > \mathrm{dist}(s, z)$ or ($\mathrm{dist}(s, w) = \mathrm{dist}(s, z)$ and $w > z$) then
13:                $X := X \cup \{w\}$
14:        $RSSP(P \cup \{u\}, X)$
15:    return
```