This advance publication article will be replaced by the finalized version after proofreading.
Space-efficient FPT Algorithms for Degeneracy

Naohito MATSUMOTO[^a], Nonmember, Kazuhiro KURITA[^b], Member, and Masashi KIYOMI[^c], Nonmember

SUMMARY The degeneracy of a graph $G$ is defined as the smallest value $k$ such that every subgraph of $G$ has a vertex with a degree of at most $k$. Given a graph $G$, its degeneracy can be easily calculated provided sufficient memory is available. In this paper, we focus on scenarios where only $o(n)$ bits of additional read-write memory are available, apart from the input stored in read-only memory. Within this context, we introduce two FPT algorithms for degeneracy, parameterized by neighborhood diversity and the cluster vertex deletion number.

key words: space-efficient algorithm, degeneracy, neighborhood diversity, cluster vertex deletion

1. Introduction

Graphs are mathematical models for representing various data, such as road networks, chemical compound structures, and social networks. These real-world graphs are often sparse, meaning that they typically have relatively few edges compared to the square of the number of vertices. Among the various parameters for sparsity, degeneracy is one of the typical parameters. A graph $G$ is $k$-degenerate if every induced subgraph of $G$ has vertices with degrees at most $k$ [1]. Furthermore, the minimum integer that satisfies the aforementioned condition is defined as the degeneracy of $G$. The degeneracy is also an interesting parameter from a practical point of view since real-world graphs often have small degeneracy [2], [3].

A classical task in network analysis is the extraction of dense subgraphs. There are various definitions of dense subgraphs, such as pseudo-clique, $k$-plex, and $k$-club [4]. One of the most fundamental problems in dense subgraph extraction is the enumeration of maximal cliques. There are theoretically, and practically efficient algorithms based on various approaches [2], [5]–[7]. Among these algorithms, a maximal clique enumeration algorithm using degeneracy has a good theoretical and practical performance [2], [3], [5]. Besides maximal clique enumeration, computing degeneracy is essential since computing degeneracy is closely related to computing the core decomposition. This decomposition is used for network community discovery, dense subgraph discovery, etc. [4], [8], [9].

It is known that the core decomposition and computation of the degeneracy can be done in $O(n + m)$ time [10], see also Algorithm 1. In some practical fields, the computational cost is not sufficient even for linear time algorithms for the number of edges since many real-world graphs are huge. From this motivation, there are approximation algorithms that run in sublinear time [11],[12]. The computation of degeneracy on huge graphs is of interest not only to improve computation time but also to improve working space. Since many real-world graphs are too large to store in memory, algorithms have been proposed to compute degeneracy in a small working space [13]–[15]. From a theoretical point of view, a lower bound of the space complexity is known for computing the degeneracy in the streaming setting [16].

While a linear-time algorithm for degeneracy is known, efficient algorithms in various settings have been studied. In this paper, we focus on the space complexity for computing degeneracy. Before describing our results, we mention the results of Elberfeld et al [17]. Elberfeld et al. show a logarithmic space version of Courcelle’s theorem. Whether the degeneracy of a graph is at most $k$ can be described by an MSO formula if $k$ is a constant. Moreover, the degeneracy of $G$ is at most the treewidth of $G$. Therefore, for graphs with bounded treewidth, the degeneracy can be computed in logarithmic space using a meta-theorem in [17].

In this paper, we give FPT-time and sublinear-space algorithms parameterized by the cluster vertex deletion number and the neighborhood diversity. These parameters are not comparable to treewidth and can be small even for dense graphs such as complete graphs. See Fig. 1 for the relationship to treewidth.

Our proposed algorithms are based on the peeling algorithm in [10]. The bottleneck is memorizing which vertices are deleted. Therefore, it is not easy to implement this peeling algorithm for general graphs with less than $n$ bits. To overcome this difficulty, we use the decomposition of graphs based on the cluster vertex deletion and the neighborhood diversity. The neighborhood diversity $nd(G)$ is defined by the number of neighborhood classes in a graph. Intuitively, when running the peeling algorithm, if one vertex of a class is deleted, then all vertices of that class can be deleted. If we only memorize which class is deleted, we use just $nd(G)$ bits. Based on this idea, we obtain an $O(nd(G) \cdot n^4)$-time algorithm for degeneracy with $O(\max\{nd(G), \log n\})$ bits.

A graph with the cluster vertex deletion number $cd(G)$ becomes a cluster graph by deleting $cd(G)$ vertices. Let $G$ be a graph and $U$ be a set of vertices such that $G[V \setminus U]$ is a cluster graph. Our proposed algorithm memorizes only

[^a]: The author is with the Seikei University, Tokyo, Japan
[^b]: The author is with the Nagoya University, Nagoya, Japan
[a) E-mail: dm2362111@cc.seikei.ac.jp
[b) E-mail: kurita@i.nagoya-u.ac.jp
[c) E-mail: kiyomi@st.seikei.ac.jp

Copyright © 200x The Institute of Electronics, Information and Communication Engineers
the deleted vertices in $U$. A key observation is that it is possible to determine which vertices in $V \setminus U$ are deleted if we know the deleted vertices in $U$. Therefore, we only need to memorize which vertices in $U$ are deleted to compute the next vertex to be deleted. This observation gives us an FPT algorithm for degeneracy with $O(cd(G) \cdot \log n)$ bits.

3. Degeneracy by Neighborhood Diversity

In this section, we introduce a space-efficient FPT algorithm.

Algorithm 1 An algorithm for the degeneracy of $G$ with $O(n)$ space

procedure: Degeneracy($G$)
Output: the degeneracy of $G$
1: $k \leftarrow 0$
2: while $G$ is not $P_k$ do
3: $v_{\text{min}} \leftarrow \arg \min \{d_G(v) \mid v \in V(G)\}$
4: $k \leftarrow \max\{d_G(v_{\text{min}})\}, k\}$
5: Delete $v_{\text{min}}$ from current graph
6: return $k$

We explain two parameters used in this paper, the neighborhood diversity and the cluster vertex deletion number. The neighborhood decomposition is classifying vertices based on their adjacency. The neighborhood diversity is the minimum number of classes and any tow vertices in a class can be considered the similar vertex.

Definition 1. The neighborhood decomposition of a graph $G$ is a partition $C = \{C_1, C_2, \ldots, C_m\}$ of the vertex set $V(G)$ such that all the vertices in $C_i$ are twins. Each $C_i$ is referred as a neighborhood class, and $w$ denotes the size of the decomposition. The neighborhood diversity $\text{nd}(G)$ is defined as the size of minimum neighborhood decomposition (i.e. the minimum value of $w$).

By Lemma 1, since all the vertices in $C_i$ are twins, each $C_i$ forms either a clique or a set of independent vertices. When there is no risk of confusion, we represent $\text{nd}$ instead of $\text{nd}(G)$.

Next, we explain a cluster vertex deletion set and the cluster vertex deletion number.

Definition 2. For a vertex set $X \subseteq V(G)$, $X$ is a cluster vertex deletion set of the graph $G$ if a graph $G[V(G) \setminus X]$ is a cluster graph. $\text{cd}(G)$ denotes the minimum cluster vertex deletion set of $G$, and the cluster vertex deletion number of $G$ is $|\text{cd}(G)|$, denoted by $\text{cd}(G)$. When there is no risk of confusion, we represent $\text{cd}$ instead of $\text{cd}(G)$.

Moreover, we can show the following fact, which is useful for searching a cluster vertex deletion set of $G$.

Observation 1. Let $X$ be a cluster vertex deletion set of $G$, and $\{v_1, v_2, v_3\}$ induces $P_3$ on $G$. At least one of the vertices $v_1, v_2, v_3$ is included in $X$.

Proof. If none of the vertices inducing $P_3$ on $G$ belong to a cluster deletion $X$, it contradicts the fact that they form a clique.

3. Degeneracy by Neighborhood Diversity

In this section, we introduce a space-efficient FPT algorithm
for degeneracy parameterized by the neighborhood diversity \( \text{nd} \).

3.1 Neighborhood representative

Firstly, we introduce a notion called neighborhood representative.

**Definition 3.** The neighborhood representative, denoted as \( r_i \), in a neighborhood class \( C_i \) is defined as the vertex with the smallest index within \( C_i \).

For example, if \( C_i = \{v_3, v_5, v_7, v_8\} \), the neighborhood representative \( r_i \) of \( C_i \) is \( v_3 \).

**Lemma 2.** A vertex \( v_p \in V \) is a neighborhood representative if and only if none of the vertices \( v_1, v_2, \ldots, v_{p-1} \) is a twin of \( v_p \).

**Proof.** We need to establish the following two statements for Lemma 2.

Let \( v_p \in V \) belong to the neighborhood class \( C \). If none of the vertices \( v_1, v_2, \ldots, v_{p-1} \) is a twin of \( v_p \) and, each of those vertices does not belong to the neighborhood class \( C \). Therefore, the index \( p \) is the smallest index within the neighborhood class \( C \).

When a vertex \( v_p \in V \) is the neighborhood representative in the neighborhood class \( C \), the index \( p \) is the smallest index in \( C \). This means that none of \( v_1, v_2, \ldots, v_{p-1} \) are included in the neighborhood class \( C \). Therefore, none of these vertices is a twin of \( v_p \).

We present Algorithm 2, which checks whether a given vertex \( v_p \in V \) is the neighborhood representative. The correctness of the algorithm directly follows from Lemma 2. To determine whether a vertex \( v \in V \) is one of the neighborhood representatives of \( G \), we check if any two vertices, \( u \) and \( v \), are twins when \( u \) has an index smaller than \( v \). For each \( u \), this check can be done by checking if every neighbor of \( u \) is adjacent to \( v \) and each neighbor of \( v \) is adjacent to \( v \) in \( O(n^2) \) time and \( O(\log n) \) space. Therefore, the time complexity of all checks together is \( O(n^3) \), and the space complexity is \( O(\log n) \).

**Algorithm 2** An algorithm for checking whether vertex \( v_p \) is neighborhood representative.

**procedure:** NeighborhoodRep\((G, v_p)\)

**Output:** “Yes” if \( v_p \) is the representative, “No” otherwise

1: \( j \leftarrow 1 \)
2: while \( j < p \) do
3: \( \text{if } N_G(v_p) \setminus \{v_j\} = N_G(v_j) \setminus \{v_p\} \text{ then} \)
4: \( \text{return } \text{No} \)
5: \( j \leftarrow j + 1 \)
6: \( \text{return Yes} \)

We can easily calculate the neighborhood diversity of a graph \( G \) using Algorithm 3. It counts the number of neighborhood representatives in the given graph \( G \) instead of keeping track of the neighborhood class for each vertex. This is correct, as each neighborhood class has exactly one neighborhood representative and the total count of neighborhood representatives in \( G \) is equal to the neighborhood diversity of \( G \). The time complexity of Algorithm 3 is \( O(n^3) \), and the space complexity is \( O(\log n) \), since Algorithm 3 calls Algorithm 2 \( O(n) \) times. Therefore, we have the following Lemma.

**Algorithm 3** An algorithm for the neighborhood diversity of \( G \)

**procedure:** NeighborhoodDiversity\((G)\)

**Output:** Neighborhood diversity of \( G \)

1: \( \text{nd} \leftarrow 0 \)
2: for \( v \in V(G) \) do
3: \( \text{if NeighborhoodRep}(G, v) = \text{Yes} \) then
4: \( \text{nd} \leftarrow \text{nd} + 1 \)
5: return \( \text{nd} \)

**Lemma 3.** There is an algorithm for the neighborhood diversity whose space complexity is \( O(\log n) \) and the time complexity is \( O(n^3) \).

3.2 Calculate Degeneracy by Neighborhood Representative

We present an FPT algorithm, Algorithm 4, for calculating the degeneracy of a given graph parameterized by \( \text{nd} \). This algorithm keeps track of the deleted neighborhood classes rather than individual deleted vertices, as the vertices within the same neighborhood class have the same neighbors except for themselves, and therefore, if one vertex can be deleted, the other vertices in the same class can also be deleted. The space complexity is evidently \( O(\max(\text{nd}, \log n)) \) bits.

We only memorize which neighborhood classes are deleted in \( \text{nd} \) bits. So, we need to determine to which neighborhood class \( v \in V(G) \) belongs in order to know if \( v \) is deleted. This can be achieved by the following:

- Find the representative of the neighborhood class to which \( v \) belongs by comparing neighbors with the vertices whose indexes are smaller than that of \( v \). We denote the index of the representative by \( l \). This process takes \( O(n^3) \) time due to Algorithm 2. (Instead of returning "No", we can return the neighborhood representative of the class to which \( v_p \) belongs.)
- For each \( i = 1, \ldots, l - 1 \), check if \( v_i \) is a representative with Algorithm 2, and count the number \( i \) of neighborhood classes whose representative’s indexes are smaller than \( l \). Then, \( v \) belongs to the \( (i + 1) \)th neighborhood class. This calculation takes \( O(n^3) \) time.

Thus, we can determine to which neighborhood class \( v \) belongs, and we can also determine if a vertex \( v \) is deleted, in \( O(n^3) \) time and \( O(\max(\text{nd}, \log n)) \) space.

We consider the time complexity of Algorithm 4. First, we run Algorithm 3 in \( O(n^3) \) time. Then, the while statement...
is executed \(nd\) times. In each loop, we calculate \(r_{\text{min}}\). This can be achieved by the following:

- For \(i = 1, \ldots, n\), check if \(v_i\) is a representative by Algorithm 2 in \(O(n^3)\) time.
- For each \(i\) such that \(v_i\) is a representative, calculate \(d(v_i)\) in \(O(n^3)\) time.

\(d(v_i)\) can be calculated by checking if each neighbor of \(v_i\) is deleted. This check for each adjacent vertex \(u\) can be done by calculating the class to which \(u\) belongs in \(O(n^3)\) time with the way described above. Thus, the whole time complexity of Algorithm 4 is \(O(n^4 + nd(n^3 + n \cdot n^3))\) = \(O(nd \cdot n^3)\). Thus, we have the following theorem.

**Theorem 1.** There is an algorithm for the degeneracy whose space complexity is \(O(\max\{nd, \log n\})\) and the time complexity is \(O(nd \cdot n^2)\), where \(nd\) is the neighborhood diversity of the input graph.

**Algorithm 4** An algorithm for the degeneracy of \(G\)

\[
\text{procedure: DegeneracyByNeighborhoodRep}(G) \\
\text{Output: the degeneracy of } G \\
1: c \leftarrow \text{NeighborhoodDiversity}(G) \\
2: k \leftarrow 0 \\
3: \text{while } c > 0 \text{ do} \\
4: \quad r_{\text{min}} \leftarrow \arg \min \{d(r) \mid r \text{ is non-deleted neighborhood representative}\} \\
5: \quad \text{if } d(r_{\text{min}}) > k \text{ then} \\
6: \quad \quad k \leftarrow d(r_{\text{min}}) \\
7: \quad \text{Delete the neighborhood class that has the vertex } r_{\text{min}} \quad \Rightarrow \text{Only memorize which classes are deleted} \\
8: \quad c \leftarrow c - 1 \\
9: \text{return } k.
\]

We note that there is also an algorithm for degeneracy whose space complexity is \(O(nd \log n)\) and the time complexity is \(O(nd^2 \cdot n^2)\), since storing all neighborhood representatives using \(O(nd \log n)\) bits of space, we can compute in \(O(nd \cdot n^2)\) time to which neighborhood class some vertex belongs.

### 4. Degeneracy by Cluster Vertex Deletion Set

In this section, we present a space-efficient FPT algorithm for degeneracy parameterized by the cluster vertex deletion number \(cd\).

#### 4.1 Calculate the Cluster Vertex Deletion Set

Algorithm 5 presents an FPT algorithm for finding a cluster vertex deletion set of size at most \(w\) if it exists in \(G\). We can easily obtain cluster vertex deletion number \(cd\) by executing the algorithm for \(w = 1, 2, \ldots\) to find the minimum \(w\) such that there is a cluster vertex deletion set of size \(w\) (Algorithm 6).

**Algorithm 5** An algorithm for the cluster vertex deletion set of size at most \(w\) on a graph \(G\) with \(O(w \log n)\) bits

\[
\text{procedure: ClusterVertexDeletion}(G, w, C \leftarrow \emptyset) \\
\text{Output: a cluster vertex deletion of size at most } w \text{ if it exists, “No” otherwise} \\
1: \text{if } w = 0 \text{ then} \\
2: \quad \text{if all the vertices except for the vertices in } C \text{ form cliques then} \\
3: \quad \quad \text{return } C \\
4: \quad \text{else} \\
5: \quad \quad \text{return “No”} \\
6: \text{Let } a, b, c \in V(G) \setminus C \text{ be vertices which induce } P_3 \text{ in } G. \\
7: \text{if there are no such vertices then} \\
8: \quad \text{return } C \\
9: \text{for } v \in \{a, b, c\} \text{ do} \\
10: \quad C \leftarrow \text{ClusterVertexDeletion}(G, w - 1, C \cup \{v\}) \\
11: \quad \text{if } C \neq “No” \text{ then} \\
12: \quad \quad \text{return } C \cup \{v\} \\
13: \text{return “No”}
\]

By Observation 1, at least one of the three vertices that induce a \(P_3\) must belong to a cluster vertex deletion, and vertices not belonging to it form cliques. Therefore, if \(C\) is not yet a cluster vertex deletion at the execution of Algorithm 5, there must be three vertices that induce a \(P_3\), and at least one of them must be added to \(C\). This guarantees the correctness of Algorithm 5.

We now consider the space complexity of Algorithm 5. We have to memorize \(C\) in \(O(w \log n)\) bits, since \(|C| \leq w\). To implement the recursion, we have to be able to restore the state before the recursive call when returned from recursions. Therefore, we need to memorize what we add to \(C\) in \(O(\log n)\) bits. Since the depth of the recursion is suppressed by \(w\), we can achieve the recursion with additional \(O(w \log n)\) bits. Note that we can determine the return address of the code when returning from the recursion by the information which of three vertices are added to \(C\). Thus, the total space complexity of Algorithm 5 is \(O(w \log n)\).

Next we consider the time complexity of Algorithm 5. We can find three vertices that induce a \(P_3\) by checking for every triple of vertices whether they induce a \(P_3\). The check for each triple can be done in \(O(n)\) time by checking each pair in the triple are adjacent, but we also need to check if the vertices are in \(C\). Checking if a vertex \(v\) is in \(C\) can be done by linear search in \(C\). Since the size of \(C\) is \(O(n)\), this takes \(O(n) = O(n)\) time. Together with the fact that the number of triples of vertices is \(O(n^3)\), whole time complexity to find
three vertices not in C that induce a $P_3$ is O($n^3$). The vertices except for C form cliques if and only if there are no three vertices that induce $P_3$ since Observation 1. Therefore, the work in every node of the recursion tree takes O($n^3$) time. The number of the recursions is O($3^w$), since the depth of the recursion is O($w$). Thus, the total time complexity of the Algorithm 5 is O($3^w n^3$).

From the above, the following lemma is obtained.

**Lemma 4.** There is an algorithm for cluster vertex deletion whose space complexity is $O(c'd \cdot \log n)$ and the time complexity is $O(3^w c'd \cdot n^3)$, where $c'd$ is the cluster vertex deletion number of the input graph.

### 4.2 Calculate Degeneracy by Cluster Vertex Deletion

Algorithm 7 presents a space-efficient FPT algorithm for degeneracy parameterized by cluster vertex deletion number $c'd$. This algorithm is similar to Matula and Beck's algorithm [10].

**Algorithm 7** An algorithm for the degeneracy of $G$ parameterized by cluster vertex deletion number $c'd$  

**procedure:** DegeneracyByClusterDeletion($G$)  
**Output:** the degeneracy of $G$  
1: $w$ ← ClusterVertexDeletionNumber($G$)  
2: $C$ ← ClusterVertexDeletion($G, w$)  
3: $l$ ← the number of undeleted vertices in $C$  
4: $k$ ← 0  
5: while $l > 0$ do  
6: $r$ ← arg min $\{d(v) \mid v \in V(G)\}$  
7: if $d(r) > k$ then  
8: $k$ ← $d(r)$  
9: if $r \in C$ then  
10: $l$ ← $l - 1$  
11: Delete $r$ form current graph.\textendash Only memorise which vertices in $C$ are deleted  
12: return max $\{k, max \{d(v) \mid v \text{ is remaining vertex of } G\}\}$

In Algorithm 7, we only memorize the cluster vertex deletion $C$ in O($c'd \log n$) bits, and which vertices in $C$ are deleted in O($c'd$) bits. First we show that we can calculate the degree of every vertex.

**Lemma 5.** In Algorithm 7, we can calculate $d(v)$ for every vertex $v$ provided we have every vertex of $C$ in memory and we know whether each vertex in $C$ is deleted. The time complexity is $O(c'd \cdot n^3)$.

**Proof.** To calculate the degree of $v$, all we have to do is subtracting the number of vertices which are adjacent to $v$ in the input graph and have been deleted, from $d_G(v)$. Therefore, we contemplate calculating the number of vertices which are adjacent to $v$ and have been deleted. First, we focus on a vertex $v \in V(G)$ that does not belong to $C$. All vertices in $N_G(v)$ are categorized into those that belong to $C$ and those that do not. Since we have a list of vertices in $C$ and we memorise if each vertex in $C$ is deleted, we can check whether a vertex $u$ is in $C$ and whether it is deleted if it is in $C$, in O($c'd$) time. Thus, the number of $v$'s neighbors in $C$ that have been deleted can be calculated in O($c'd \cdot n$) time. We denote this number as $NDC(v)$.

Let $u_1, u_2, \ldots, u_t \in N_G(v)$ be $v$'s neighbors in the input graph $G$ that do not belong to $C$, arranged in ascending order of $d_G(u_1) - NDC(u_1)$, i.e. sorted by the current degrees. Note that, according to Definition 2, $\{u_1, \ldots, u_t\} \cup \{v\}$ forms a clique. Determining if any of $v$'s neighbors is in $\{u_1, \ldots, u_t\}$ takes O($c'd$) time since we only need to check whether it does not belong to $C$.

If there are vertices in $\{u_{t+1}, \ldots, u_t\}$ that are deleted, $u_t$ is also deleted, since $u_1, \ldots, u_t$ are sorted by the current degrees. We denote the maximum $i$ such that $\forall j \leq i \{d_G(u_j) - NDC(u_j) = (j - 1) \leq k\}$ as $i_{\text{max}}$. Then $i_{\text{max}}$ vertices in $\{u_1, \ldots, u_m\}$ have been deleted. Therefore, if we can calculate $i_{\text{max}}$, we can calculate the current degree of $v$. To calculate $i_{\text{max}}$, we use the algorithm below.

**Algorithm 8** An algorithm for calculating $i_{\text{max}}$

**procedure:** CalculateMax($G, v, k, C$)  
**Output:** $i_{\text{max}}$

1: $num_{del} \leftarrow 0$  
2: while True do  
3: $num_{del} \leftarrow 0$  
4: for $v \in N_G(v) \setminus C$ do  
5: if $d_G(u) - NDC(u) - num_{del} \leq k$ then  
6: $num_{del} \leftarrow num_{del} + 1$  
7: if $num_{del} = num_{del} + 1$ then  
8: return $num_{del}$

Assume that we know $f$ vertices in $N_G(v) \setminus C$ have been deleted. Then, vertex $u$ in $N_G(v) \setminus C$ satisfying  

$$d_G(u) - NDC(u) - f \leq k$$

is either has already deleted, or can be deleted. Therefore, in Algorithm 8, we can calculate the number of vertices in $N_G(v) \setminus C$ that can be deleted, that is, we set the initial number of deleted vertices (i.e. $num_{del}$) to 0, we compute $num_{del}$ iteratively. $\forall i \leq i_{\text{max}}$ \{d_G(u_j) - NDC(u_j) = (j - 1) \leq k\} is not true by the definition of $i_{\text{max}}$. That means the number of vertices which can be deleted does not increase from $i_{\text{max}}$ to $i_{\text{max}} + 1$. Thus, $i_{\text{max}}$ is equal to $num_{del}$ when Algorithm 8 terminates.

We analyze the time complexity of Algorithm 8. From the observation above, calculating $NDC(u)$ requires O($c'd \cdot n$) time. When the algorithm has the largest repeat loop, for every step, there is $num_{del} = num_{del} + 1$ and all vertices can be deleted. Thus, calculating $NDC(u)$ is executed O($n^2$) time. The total time complexity of calculate $d(v)$ is thus O($c'd \cdot n^3$) time.

Now we consider the time complexity of calculating $d(v)$. Since $d(v) = d_G(v) - NDC(v) - i_{\text{max}}$, we can calculate $d(v)$ in O($c'd \cdot n^3$) time.

Next we consider the case that $v \in C$. In this case, $d(v) = d_1(v) + d_2(v)$ holds, where $d_1(v)$ is the degree in C,
and $d_2(v)$ is the number of undeleted neighbors of $v$ not in $C$. We can easily calculate $d_1(v)$ in $O(cd \cdot n)$ time, since we know which vertices are deleted. To calculate $d_2(v)$, we calculate $d(u)$ in $O(cd \cdot n^3)$ time with the above method, for each neighbor of $v$ in the input graph, and check if it is already deleted. This takes $O(cd \cdot n^4)$ time. Therefore, we can calculate $d_1(v) + d_2(v)$ in $O(cd \cdot n + cd \cdot n^3) = O(cd \cdot n^3)$ time.

Finally, we consider the time complexity of Algorithm 7. It takes $O(3^{cd} \cdot n^4)$ time for calculating a cluster vertex deletion set of size $cd$. In the “while loop” we need to calculate the degree of each vertex. This takes $O(n \cdot cd \cdot n^3 + cd \cdot cd^4 \cdot \log cd)$ time. The “while loop” is executed $O(cd)$ times. Thus, the whole time complexity of Algorithm 7 is $O(3^{cd} \cdot n^4 + cd^3 \cdot n^4)$. Thus, we have the theorem below.

**Theorem 2.** There is an algorithm for the degeneracy whose space complexity is $O(cd \cdot \log n)$ and the time complexity is $O(3^{cd} \cdot n^4)$, where $cd$ is the cluster vertex deletion number of the input graph.

5. Concluding Remarks

We showed two space-efficient FPT-algorithms, one of which is parameterized by neighborhood diversity, and the other is parameterized by cluster vertex deletion number. Whether there is a polynomial time algorithm for degeneracy whose space complexity is $o(n)$ bits is an interesting open problem.

Acknowledgments

This work was partially supported by JSPS Kakenhi Grant Numbers JP21K17812, JP22H03549, JST ACT-X Grant Number JPMJAX2105.

References