External-Memory Network Analysis Algorithms for Naturally Sparse Graphs

Michael T. Goodrich and Paweł Pszona

Dept. of Computer Science
University of California, Irvine

Abstract. In this paper, we present a number of network-analysis algorithms in the external-memory model. We focus on methods for large naturally sparse graphs, that is, n-vertex graphs that have $O(n)$ edges and are structured so that this sparsity property holds for any subgraph of such a graph. We give efficient external-memory algorithms for the following problems for such graphs:

1. Finding an approximate d-degeneracy ordering.
2. Finding a cycle of length exactly c.
3. Enumerating all maximal cliques.

Such problems are of interest, for example, in the analysis of social networks, where they are used to study network cohesion.

1 Introduction

Network analysis studies the structure of relationships between various entities, with those entities represented as vertices in a graph and their relationships represented as edges in that graph (e.g., see [11]). For example, such structural analyses include link-analysis for Web graphs, centrality and cohesion measures in social networks, and network motifs in biological networks. In this paper, we are particularly interested in network analysis algorithms for finding various kinds of small subgraphs and graph partitions in large graphs that are likely to occur in practice. Of course, this begs the question of what kinds of graphs are likely to occur in practice.

1.1 Naturally Sparse Graphs

A network property addressing the concept of a "real world" graph that is gaining in prominence is the *k-core number* [25], which is equivalent to a graph's *width* [16], *linkage* [19], *k-inductivity* [18], and *k-degeneracy* [2,21], and is one less than its Erdős-Hajnal coloring number [14]. A k-core, G', in a graph, G, is a maximal connected subgraph of G such that each vertex in G' has degree at least k. The k-core number of a graph G is the maximum k such that G has a non-empty k-core. We say that a graph G is *naturally sparse* if its k-core number is $O(1)$. This terminology is motivated by the fact that almost every n-vertex graph with $O(n)$ edges has a bounded k-core number, since Pittel *et al.* [23] show that a random graph with n vertices and cn edges (in the Erdős-Rényi model) has k-core number at most $2c + o(c)$, with high probability. Riordan [24] and

Fernholz and Ramachandran [15] have also studied k-cores in random graphs.

In addition, we also have the following:

- Every s-vertex subgraph of a naturally sparse graph is naturally sparse, hence, has $O(s)$ edges.
- Any planar graph has k-core number at most 5, hence, is naturally sparse.
- Any graph with bounded arboricity is naturally sparse (e.g., see [10]).
- Eppstein and Strash [13] verify experimentally that real-world graphs in four different data repositories all have small k-core numbers relative to their sizes; hence, these real-world graphs give an empirical motivation for naturally sparse graphs.
- Any network generated by the Barabási-Albert [4] preferential attachment process, with $m \in O(1)$, or as in Kleinberg's small-world model [20], is naturally sparse.

Of course, one can artificially define an n-vertex graph, G', with $O(n)$ edges that is not naturally sparse just by creating a clique of $O(n^{1/2})$ vertices in an n-vertex graph, G, having $O(n)$ edges. We would argue, however, that such a graph G' would not arise "naturally." We are interested in algorithms for large, naturally sparse graphs.

1.2 External-Memory Algorithms

One well-recognized way of designing algorithms for processing large data sets is to formulate such algorithms in the *external memory model* (e.g., see the excellent survey by Vitter [27]). In this model, we have a single CPU with main memory capable of storing M items and that computer is connected to D external disks that are capable of storing a much larger amount of data. Initially, we assume the parallel disks are storing an input of size N. A single I/O between one of the external disks and main memory is defined as either reading a block of B consecutively stored items into memory or writing a block of the same size to a disk. Moreover, we assume that this can be done on all D disks in parallel if need be.

Two fundamental primitives of the model are *scanning* and *sorting*. Scanning is the operation of streaming N items stored on D disks through main memory, with I/O complexity

$$scan(N) = \Theta\left(\frac{N}{DB}\right),$$

and sorting N items has I/O complexity

$$sort(N) = \Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right),$$

e.g., see Vitter [27].

Since this paper concerns graphs, we assume a problem instance is a graph $G = (V, E)$, with $n = |V|$, $m = |E|$ and $N = |G| = m + n$. If G is d-degenerate, that is, has k-core number, d, then $m \le dn$ and $N = O(dn) = O(n)$ for $d = O(1)$. We use d to denote the k-core number of an input graph, G, and we use the term "d-degenerate" as a shorthand for "k-core number equal to d."

1.3 Previous Related Work

Several researchers have studied algorithms for graphs with bounded k-core numbers (e.g., see [1,3,12,17,18]). These methods are often based on the fact that the vertices in a graph with k-core number, d, can be ordered by repeatedly removing a vertex of degree at most d, which gives rise to a numbering of the vertices, called a *d-degeneracy ordering* or *Erdős-Hajnal sequence*, such that each vertex has at most d edges to higher-numbered vertices. In the RAM model, this greedy algorithm takes $O(n)$ time (e.g., see [5]). Bauer *et al.* [6] describe methods for generating such graphs and their d-degeneracy orderings at random.

In the internal-memory RAM model, Eppstein *et al.* [12] show how to find all maximal cliques in a d-degenerate graph in $O(d3^{d/3}n)$ time. Alon *et al.* [3] show that one can find a cycle of length exactly c, or show that one does not exist, in a d-degenerate graph in time $O(d^{1-1/k}m^{2-1/k})$, if $c = 4k-2$, time $O(dm^{2-1/k})$, if $c = 4k-1$ or $4k$, and time $O(d^{1+1/k}m^{2-1/k})$, if $c = 4k+1$.

A closely related concept to a d-degeneracy ordering is a *k-core decomposition* of a graph, which is a labeling of each vertex v with the largest k such that v belongs to a k-core. Such a labeling can also be produced by the simple linear-time greedy algorithm that removes a vertex of minimum degree with each iteration. Cheng *et al.* [9] describe recently an external-memory method for constructing a k-core decomposition, but their method is unfortunately fatally flawed[1]. The challenge in producing a k-core decomposition or d-degeneracy ordering in external memory is that the standard greedy method, which works so well in internal memory, can cause a large number of I/Os when implemented in external memory. Thus, new approaches are needed.

1.4 Our Results

In this paper, we present efficient external-memory network analysis algorithms for naturally sparse graphs (i.e., degenerate graphs with small degeneracy). First, we give a simple algorithm for computing a $(2+\epsilon)d$-degeneracy ordering of a d-degenerate graph $G = (V, E)$, without the need to know the value of d in advance. The I/O complexity of our algorithm is $O(sort(dn))$.

Second, we give an algorithm for determining whether a d-degenerate graph $G = (V, E)$ contains a simple cycle of a fixed length c. This algorithm uses $O\Big(d^{1\pm\epsilon} \cdot \big(k \cdot sort(m^{2-\frac{1}{k}}) + (4k)! \cdot scan(m^{2-\frac{1}{k}})\big)\Big)$ I/O complexity, where ϵ is a constant depending on $c \in \{4k-2, \ldots, 4k+1\}$.

Finally, we present an algorithm for listing all maximal cliques of an undirected d-degenerate graph $G = (V, E)$, with $O(3^{\delta/3} sort(dn))$ I/O complexity, where $\delta = (2+\epsilon)d$.

One of the key insights to our second and third results is to show that, for the sake of designing efficient external-memory algorithms, using a $(2+\epsilon)d$-degeneracy ordering is almost as good as a d-degeneracy ordering. In addition to this insight, there are a number of technical details that lead to our results, which we outline in the remainder of this manuscript.

[1] We contacted the authors and they confirmed that their method is indeed incorrect.

2 Approximating a d-Degeneracy Ordering

Our method for constructing a $(2+\epsilon)d$-degeneracy ordering for a d-degenerate graph, $G = (V, E)$, is quite simple and is given below as Algorithm 1. Note that our algorithm does not take into account the value of d, but it assumes we are given a constant $\epsilon > 0$ as part of the input. Also, note that this algorithm destroys G in the process. If one desires to maintain G for other purposes, then one should first create a backup copy of G.

```
1: $L \leftarrow \emptyset$
2: **while** $G$ is nonempty **do**
3:     $S \leftarrow n\epsilon/(2+\epsilon)$ vertices of smallest degree in $G$
4:     $L \leftarrow L|S$                    // append $S$ to the end of $L$
5:     remove $S$ from $G$
6: **end while**
7: **return** $L$
```

Algorithm 1: Approximate degeneracy ordering of vertices

Lemma 1. *If G is a d-degenerate graph, then Algorithm 1 computes a $(2+\epsilon)d$-degeneracy ordering of G.*

Proof. Observe that any d-degenerate graph with n vertices has at most $2n/c$ vertices of degree at least cd. Thus, G has at most $2n/(2+\epsilon)$ vertices of degree at least $(2+\epsilon)d$. This means that the $n\epsilon/(2+\epsilon)$ vertices of smallest degree in G each have degree at most $(2+\epsilon)d$. Therefore, every element of set S created in line 3 has at most $(2+\epsilon)d$ neighbors in (the remaining graph) G. When we add S to L in line 4, we keep the property that every element of L has at most $(2+\epsilon)d$ neighbors in G that are placed behind it in L. Furthermore, note that, after we remove vertices in S (and their incident edges) from G in line 5, G is still at most d-degenerate (every subgraph of a d-degenerate graph is at most d-degenerate); hence, an inductive argument applies to the remainder of the algorithm. □

Note that, after $\lceil \log_{(2+\epsilon)/2}(dn) \rceil = O(\lg n)$ iterations, we must have processed all of G and placed all its vertices on L, which is a $(2+\epsilon)d$-degeneracy ordering for G and that this property holds even though the algorithm does not take the value of d into account.

The following lemma is proved in the Appendix.

Lemma 2. *An iteration of the `while` loop (lines 3-5) of Algorithm 1 can be implemented in $O(sort(dn))$ I/O's in the external-memory model, where n is the number of vertices in G at the beginning of the iteration.*

Thus, we have the following.

Theorem 1. *We can compute a $(2+\epsilon)d$-degeneracy ordering of a d-degenerate graph, G, in $O(sort(dn))$ I/O's in the external-memory model, without knowing the value of d in advance.*

Proof. Since the number of vertices of G decreases by a factor of $2/(2+\epsilon)$ in each iteration, and each iteration uses $O(sort(dn))$ I/O's, where n is the number of vertices in G at the beginning of the iteration (by Lemma 2), the total number of I/O's, $I(G)$, is bounded by

$$\begin{aligned} I(G) &= O\left(sort(dn) + sort\Big(\big(2/(2+\epsilon)\big)dn\Big) + sort\Big(\big(2/(2+\epsilon)\big)^2 dn\Big) + \cdots\right) \\ &= O\left(sort(dn)\left(1 + \frac{2}{2+\epsilon} + \Big(\frac{2}{2+\epsilon}\Big)^2 + \cdots\right)\right) \\ &= O(sort(dn)). \end{aligned}$$

□

This theorem hints at the possibility of effectively using a $(2+\epsilon)d$-degeneracy ordering in place of a d-degeneracy ordering in external-memory algorithms for naturally sparse graphs. As we show in the remainder of this paper, achieving this goal is indeed possible, albeit with some additional alterations from previous internal-memory algorithms.

3 Short Paths and Cycles

In this section, we present external-memory algorithms for finding short cycles in directed or undirected graphs. Our approach is an external-memory adaptation of internal-memory algorithms by Alon *et al.* [3]. We begin with the definition and an example of a *representative* due to Monien [22]. A p-set is a set of size p.

Definition 1 (representative). *Let $\mathcal{F}$ be a collection of p-sets. A sub-collection $\widehat{\mathcal{F}} \subseteq \mathcal{F}$ is a q-*representative *for $\mathcal{F}$, if for every q-set B, there exists a set $A \in \mathcal{F}$ such that $A \cap B = \emptyset$ if and only if there exists a set $\widehat{A} \in \widehat{\mathcal{F}}$ with this property.*

Every collection of p-sets $\mathcal{F}$ has a q-representative $\widehat{\mathcal{F}}$ of size at most $\binom{p+q}{p}$ (from Bollobás [7]). An optimal representative, however, seems difficult to find. Monien [22] gives a construction of representatives of size at most $O(\sum_{i=1}^{q} p^i)$. It uses a p-ary tree of height $\leq q$ with the following properties.

- Each node is labeled with either a set $A \in \mathcal{F}$ or a special symbol λ.
- If a node is labeled with a set A and its depth is less than q, it has exactly p children, edges to which are labeled with elements from A (one element per edge, every element of A is used to label exactly one edge).
- If a node is labeled with λ or has depth q, it has no children.
- Let $E(v)$ denote the set of all edge labels on the way from the vertex v to the root of the tree. Then, for every v:
  - if v is labeled with A, then $A \cap E(v) = \emptyset$
  - if v is labeled with λ, then there are no $A \in \mathcal{F}$ s.t. $A \cap E(v) = \emptyset$.

Monien shows that if a tree T fulfills the above conditions, defining $\widehat{\mathcal{F}}$ to be the set of all labels of the tree's nodes yields a q-representative for $\mathcal{F}$. As an example, consider a collection of 2-sets, $\mathcal{F} = \{\{2,4\},\{1,5\},\{1,6\},\{1,7\},\{3,6\},\{3,8\},\{4,7\},\{4,8\}\}$. Fig. 1 presents $\widehat{\mathcal{F}}$, a 3-representative of $\mathcal{F}$ in the tree form.

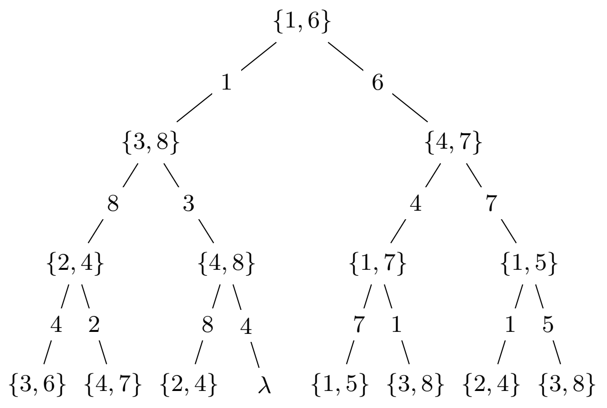

Fig. 1. Tree representation of $\widehat{\mathcal{F}}$

The main benefit of using representatives in the tree form stems from the fact that their sizes are bounded by a function of only p and q (i.e., maximum size of a representative does not depend on $|\mathcal{F}|$). It gives a way of storing paths of given length between two vertices of a graph in a space-efficient way (see Appendix for details).

The algorithm for finding a cycle of given length has two stages. In the first stage, vertices of *high degree* are processed to determine if any of them belongs to a cycle. This is realized using algorithm `cycleThrough` from Lemma 5. Since there are not many vertices of *high degree*, this can be realized efficiently.

In the second stage, we remove vertices of *high degree* from the graph. Then, we group all simple paths that are half the cycle length long by their endpoints and compute representatives for every such set (see Lemma 3). For each pair of vertices (u, v), we determine (using `findDisjoint` from Lemma 4) if there are two paths: p from u to v and p' from v to u, such that p and p' do not share any internal vertices. If this is the case, $C = p \cup p'$ is a cycle of required length.

The following representatives-related lemmas are proved in the Appendix.

Lemma 3. *We can compute a q-representative $\widehat{\mathcal{F}}$ for a collection of p-sets $\mathcal{F}$, of size $|\widehat{\mathcal{F}}| \leq \sum_{i=1}^{q} p^i$, in $O\Big(\big(\sum_{i=1}^{q+1} p^i\big) \cdot scan\big(|\mathcal{F}|\big)\Big)$ I/O's.*

Lemma 4. *For a collection of p-sets, $\mathcal{F}$, and a collection of q-sets, $\mathcal{G}$, there is an external-memory method,* `findDisjoint`$(\mathcal{F}, \mathcal{G})$*, that returns a pair of sets (A, B) ($A \in \mathcal{F}$, $B \in \mathcal{G}$) s.t. $A \cap B = \emptyset$ or returns ϵ if there are no such pairs of sets.* `findDisjoint` *uses $O\Big(\big(\sum_{i=1}^{q+3} p^i + \sum_{i=1}^{p+3} q^i\big) \cdot scan\big(|\mathcal{F}| + |\mathcal{G}|\big)\Big)$ I/O's.*

Lemma 5. *Let $G = (V, E)$. A cycle of length exactly k that passes through arbitrary $v \in V$, if it exists, can be found by an external-memory algorithm* `cycleThrough`(G, k, v) *in $O\big((k-1)! \cdot scan(m)\big)$ I/O's, where $m = |E|$, via the use of representatives.*

Before we present our result for naturally sparse graphs, we first give an external-memory method for general graphs.

Theorem 2. *Let $G = (V, E)$ be a directed or an undirected graph. There is an external-memory algorithm that decides if G contains a cycle of length exactly $c \in \{2k-1, 2k\}$, and finds such cycle if it exists, that takes $O\big(k \cdot sort(m^{2-\frac{1}{k}}) + (2k-1)! \cdot scan(m^{2-\frac{1}{k}})\big)$ I/O's.*

Proof. Algorithm 2 handles the case of general graphs (which are not necessarily *naturally sparse*), and cycles of length $c = 2k$ (the case of $c = 2k-1$ is analogous).

```
 1: $\Delta \leftarrow m^{\frac{1}{k}}$
 2: for all $v$ – vertex of degree $\geq \Delta$ do
 3:     $C \leftarrow$ cycleThrough$(G, 2k, v)$
 4:     if $C \neq \epsilon$ then
 5:         return $C$
 6:     end if
 7: end for
 8: remove vertices of degree $\geq \Delta$ from $G$
 9: generate all directed paths of length $k$ in $G$
10: sort the paths lexicographically, according to their endpoints
11: group all paths $u \rightsquigarrow v$ into collection of $(k-1)$-sets $\mathcal{F}_{uv}$
12: for all pairs $(\mathcal{F}_{uv}, \mathcal{F}_{vu})$ do
13:     $P \leftarrow$ findDisjoint$(\mathcal{F}_{uv}, \mathcal{F}_{vu})$
14:     if $P = (A, B)$ then
15:         return $C = A \cup B$
16:     end if
17: end for
18: return $\epsilon$
```

Algorithm 2: Short cycles in general graphs

Since there are at most $m/\Delta = m^{1-\frac{1}{k}}$ vertices of degree at least Δ, and each call to `cycleThrough` requires $O\big((2k-1)! \cdot scan(m)\big)$ I/O's (by Lemma 5), the first `for` loop (lines 2-7) takes $O\big(m^{1-\frac{1}{k}} \cdot (2k-1)! \cdot scan(m)\big) = O\big((2k-1)! \cdot scan(m^{2-\frac{1}{k}})\big)$ I/O's.

Removing vertices of high degree in line 8 is realized just like line 5 of Algorithm 1, in $O\big(sort(m)\big)$ I/O's. There are at most $m\Delta^{k-1} = m^{2-\frac{1}{k}}$ paths to be generated in line 9. It can be done in $O\big(k \cdot sort(m^{2-\frac{1}{k}})\big)$ I/O's (see Appendix). Sorting the paths (line 10) takes $O\big(sort(m^{2-\frac{1}{k}})\big)$ I/O's. After that, creating $\mathcal{F}_{uv}$'s (line 11) requires $O\big(scan(m^{2-\frac{1}{k}})\big)$ I/O's.

The `groupF` procedure groups $\mathcal{F}_{uv}$ and $\mathcal{F}_{vu}$ together. Assume we store $\mathcal{F}_{uv}$'s as tuples (u, v, S), for $S \in \mathcal{F}_{uv}$, in a list F. By $u \prec v$ we denote that u precedes v in an arbitrary ordering of V. For $u \prec v$, tuples $(u, v, 1, S)$ from line 3 mean that $S \in \mathcal{F}_{uv}$, while tuples $(u, v, 2, S)$ from line 5 mean that $S \in \mathcal{F}_{vu}$. The `for` loop (lines 1-7) clearly takes $O\big(scan(m^{2-\frac{1}{k}})\big)$ I/O's. After sorting F (line 8) in $O\big(sort(m^{2-\frac{1}{k}})\big)$ I/O's, tuples for sets from $\mathcal{F}_{uv}$ directly precede those for sets from $\mathcal{F}_{vu}$, allowing us to execute line 9 in $O\big(scan(m^{2-\frac{1}{k}})\big)$ I/O's.

```
proc groupF
1: for all $(u, v, S)$ in $F$ do
2:    if $u \prec v$ then
3:       write $(u, v, 1, S)$ back to $F$
4:    else
5:       write $(v, u, 2, S)$ back to $F$
6:    end if
7: end for
8: sort $F$ lexicographically
9: scan $F$ to determine pairs $(\mathcal{F}_{uv}, \mathcal{F}_{uv})$
```

Based on Lemma 4, the total number of I/O's in calls to `findDisjoint` in Algorithm 2, line 13 is

$$\begin{aligned} &O\Big(\textstyle\sum_{u,v} \big(\sum_{i=1}^{k+2} (k-1)^i \cdot scan(|\mathcal{F}_{uv}| + |\mathcal{F}_{vu}|)\big)\Big) \\ &= O\Big(\big(\textstyle\sum_{i=1}^{k+2} (k-1)^i \big) \cdot \sum_{u,v} scan\big(|\mathcal{F}_{uv}| + |\mathcal{F}_{vu}|\big)\Big) \\ &= O\big((2k-1)! \cdot scan(m^{2-\frac{1}{k}})\big) \end{aligned}$$

as we set $p = q = k - 1$ and $\sum_{i=1}^{k+2}(k-1)^i = O\big((k-1)^{k+3}\big) = O\big((2k-1)!\big)$.

Putting it all together, we get that Algorithm 2 runs in $O\big(sort(m^{2-\frac{1}{k}}) + (2k-1)! \cdot scan(m^{2-\frac{1}{k}})\big)$ total I/O's. □

Theorem 3. *Let $G = (V, E)$ be a directed or an undirected graph. There is an external-memory algorithm that, given L – a δ-degeneracy ordering of G (for $\delta = (2+\epsilon)d$), finds a cycle of length exactly c, or concludes that it does not exist:*

(i) in $O\Big(\delta^{1-\frac{1}{k}} \cdot \big(k \cdot sort(m^{2-\frac{1}{k}}) + (4k)! \cdot scan(m^{2-\frac{1}{k}})\big)\Big)$ I/O's if $c = 4k - 2$
(ii) in $O\Big(\delta \cdot \big(k \cdot sort(m^{2-\frac{1}{k}}) + (4k)! \cdot scan(m^{2-\frac{1}{k}})\big)\Big)$ I/O's if $c = 4k - 1$ or $c = 4k$
(iii) in $O\Big(\delta^{1+\frac{1}{k}} \cdot \big(k \cdot sort(m^{2-\frac{1}{k}}) + (4k)! \cdot scan(m^{2-\frac{1}{k}})\big)\Big)$ I/O's if $c = 4k + 1$

Proof. We describe the algorithm for the case of directed G, with $c = 4k + 1$, as other cases are similar (and a little easier). We assume that $\delta < m^{\frac{1}{2k+1}}$, which is obviously the case for *naturally sparse* graphs. Otherwise, running Algorithm 2 on G achieves the advertised complexity.

Algorithm 3 is remarkably similar to Algorithm 2 and so is its analysis. Differences lie in the value of $\varDelta$ and in line 9, when only *some* paths of length $2k$ and $2k + 1$ are generated. As explained in [3], it suffices to only consider

```
1: $\Delta \leftarrow m^{\frac{1}{k}}/\delta^{1+\frac{1}{k}}$
2: for all $v$ – vertex of degree $\geq \Delta$ do
3:     $C \leftarrow$ cycleThrough$(G, 4k+1, v)$
4:     if $C \neq \epsilon$ then
5:         return $C$
6:     end if
7: end for
8: remove vertices of degree $\geq \Delta$ from $G$
9: generate directed paths of length $2k$ and $2k+1$ in $G$
10: sort the paths lexicographically, according to their endpoints
11: group all paths $u \rightsquigarrow v$ of length $2k$ into collection of $(2k-1)$-sets $\mathcal{F}_{uv}$
12: group all paths $u \rightsquigarrow v$ of length $2k+1$ into collection of $(2k)$-sets $\mathcal{G}_{uv}$
13: for all pairs $(\mathcal{F}_{uv}, \mathcal{G}_{uv})$ do
14:     $P \leftarrow$ findDisjoint$(\mathcal{F}_{uv}, \mathcal{G}_{vu})$
15:     if $P = (A, B)$ then
16:         return $C = A \cup B$
17:     end if
18: end for
19: return $\epsilon$
```

Algorithm 3: Short cycles in degenerate graphs

all $(2k+1)$-paths that start with two backward-oriented (in L) edges and all $2k$-paths that start with a backward-oriented (in L) edge. The number of these paths is $O(m^{2-\frac{1}{k}}\delta^{1+\frac{1}{k}})$. Since we can generate them in $O\left(k\delta^{1+\frac{1}{k}} \cdot sort(m^{2-\frac{1}{k}})\right)$ I/O's (see Appendix), and there are at most $O(m^{1-\frac{1}{k}}\delta^{1+\frac{1}{k}})$ vertices in G of degree $\geq \Delta$, the theorem follows. □

4 All Maximal Cliques

The Bron-Kerbosch algorithm [8] is often the choice when one needs to list all maximal cliques of an undirected graph $G = (V, E)$. It was initially improved by Tomita *et al.* [26]. We present this improvement as the BronKerboschPivot procedure ($\Gamma(v)$ denotes the set of neighbors of vertex v).

```
proc BronKerboschPivot($P$, $R$, $X$)
1: if $P \cup X = \emptyset$ then
2:     output $R$        //maximal clique
3: end if
4: $u \leftarrow$ vertex from $P \cup X$ that maximizes $|P \cap \Gamma(u)|$
5: for all $v \in P \setminus \Gamma(v)$ do
6:     BronKerboschPivot($P \cap \Gamma(v)$, $R \cup \{v\}$, $X \cap \Gamma(v)$)
7:     $P \leftarrow P \setminus \{v\}$
8:     $X \leftarrow X \cup \{v\}$
9: end for
```

The meaning of the arguments to `BronKerboschPivot`: R is a (possibly non-maximal) clique, P and X are a division of the set of vertices that are neighbors of all vertices in R, s.t. vertices in P are to be considered for adding to R while vertices in X are restricted from the inclusion.

Whereas Tomita *et al.* run the algorithm as `BronKerboschPivot(`V`,`$\emptyset$`,`$\emptyset$`)`, Eppstein *et al.* [12] improved it even further for the case of a d-degenerate G by utilizing its d-degeneracy ordering $L = \{v_1, v_2, \ldots, v_n\}$ and by performing n independent calls to `BronKerboschPivot`. Algorithm 4 presents their version. It runs in time $O(dn3^{d/3})$ in the RAM model.

```
1: for $i \leftarrow 1 \ldots n$ do
2:     $P \leftarrow \Gamma(v_i) \cap \{v_j : j > i\}$
3:     $X \leftarrow \Gamma(v_i) \cap \{v_j : j < i\}$
4:     BronKerboschPivot($P$,$\{v_i\}$,$X$)
5: end for
```

Algorithm 4: Maximal cliques in degenerate graph

The idea behind Algorithm 4 is to limit the depth of recursive calls to $|P| \leq d$ and then apply the analysis of Tomita *et al.* [26].

We show how to efficiently implement Algorithm 4 in the external memory model using a $(2+\epsilon)d$-degeneracy ordering of G. Following [12], we define subgraphs $H_{P,X}$ of G.

Definition 2 (Graphs $H_{P,X}$). *Subgraph $H_{P,X} = (V_{P,X}, E_{P,X})$ of $G = (V, E)$ is defined as follows:*

$$\begin{aligned} V_{P,X} &= P \cup X \\ E_{P,X} &= \{(u, v) \colon (u, v) \in E \wedge (u \in P \vee v \in P)\} \end{aligned}$$

That is, $H_{P,X}$ contains all edges in G whose endpoints are from $P \cup X$, and at least one of them lies in P. To ensure efficiency, $H_{P,X}$ is passed as an additional argument to every call to `BronKerboschPivot` with P and X. It is used in determining u at line 4 of `BronKerboschPivot` (we simply choose a vertex of highest degree in $H_{P,X}$).

The following two lemmas regarding construction of $H_{P,X}$'s are proved in the Appendix.

Lemma 6. *Given a δ-degeneracy ordering L of an undirected d-degenerate graph G ($\delta = (2+\epsilon)d$), all initial sets P, X, and graphs $H_{P,X}$ that are passed to* `BronKerboschPivot` *in line 4 of Algorithm 4 can be generated in $O(sort(\delta^2 n))$ I/O's.*

Lemma 7. *Given a δ-degeneracy ordering L of an undirected d-degenerate graph G ($\delta = (2+\epsilon)d$), in a call to* `BronKerboschPivot` *that was given $H_{P,X}$, with $|P| = p$ and $|X| = x$, all graphs $H_{P\cap\Gamma(v), X\cap\Gamma(v)}$ that have to be passed to recursive calls in line 6, can be formed in $O(sort(\delta p^2(p + x)))$ I/O's.*

Theorem 4. *Given a δ-degeneracy ordering L of an undirected d-degenerate graph G ($\delta = (2+\epsilon)d$), we can list all its maximal cliques in $O(3^{\delta/3} sort(\delta n))$ I/O's.*

Proof. Consider a call to `BronKerboschPivot(`P_v`, `$\{v\}$`, `X_v`)`, with $|P_v| = p$ and $|X_v| = x$. Define $\widehat{D}(p,x)$ to be the maximum number of I/O's in this call. Based on Lemma 7, $\widehat{D}(p,x)$ satisfies the following recurrence relation:

$$\widehat{D}(p,x) \le \begin{cases} \max_k\{k\widehat{D}(p-k,x)\} + O\big(sort(\delta p^2(p+x))\big) & \text{if } p > 0 \\ e & \text{if } p = 0 \end{cases}$$

for constant e greater than zero, which can be rewritten as

$$\widehat{D}(p,x) \le \begin{cases} \max_k\{k\widehat{D}(p-k,x)\} + c \cdot \frac{\delta p^2(p+x)}{DB} \log_{M/B}(\delta p^2(p+x)) & \text{if } p > 0 \\ e & \text{if } p = 0 \end{cases}$$

for a constant $c > 0$. Since $p \le \delta$ and $p+x \le n$, we have $\log_{M/B}(\delta p^2(p+x)) \le \log_{M/B}(\delta^3 n) = O(\log_{M/B} n)$ for $\delta = O(1)$. Thus, the relation for $\widehat{D}(p,x)$:

$$\widehat{D}(p,x) \le \begin{cases} \max_k\{k\widehat{D}(p-k,x)\} + \delta p^2(p+x) \cdot \frac{c' \log_{M/B} n}{DB} & \text{if } p > 0 \\ e & \text{if } p = 0 \end{cases}$$

where c' and e are constants greater than zero. Note that this is the relation for $D(p,x)$ of Eppstein *et. al* [12] (we set $d = \delta$, $c_1 = \frac{c' \log_{M/B} n}{DB}$ and $c_2 = e$). Since the solution for $D(p,x)$ was $D(p,x) = O((d+x)3^{p/3})$, the solution for $\widehat{D}(p,x)$ is

$$\widehat{D}(p,x) = O\Big((\delta+x)3^{p/3} \cdot \frac{c' \log_{M/B} n}{DB}\Big) = O\Big(\frac{\delta+x}{DB} 3^{p/3} \log_{M/B} n\Big)$$

The total size of all sets X_v passed to initial calls to `BronKerboschPivot` is $O(\delta n)$, and every set P has at most δ vertices. It follows that the total number of I/O's in recursive calls is

$$\sum_v O\Big(\frac{\delta+|X_v|}{DB} 3^{\delta/3} \log_{M/B} n\Big) = O\Big(3^{\delta/3} \frac{\delta n}{DB} \log_{M/B} n\Big) = O\big(3^{\delta/3} sort(\delta n)\big)$$

Combining this with Lemma 6, we get that our external memory version of Algorithm 4 takes $O\big(sort(\delta^2 n) + 3^{\delta/3} sort(\delta n)\big) = O\big(3^{\delta/3} sort(\delta n)\big)$ I/O's. □

References

1. N. Alon and S. Gutner. Linear time algorithms for finding a dominating set of fixed size in degenerated graphs. *Algorithmica*, 54(4):544–556, 2009.
2. N. Alon, J. Kahn, and P. D. Seymour. Large induced degenerate subgraphs. *Graphs and Combinatorics*, 3:203–211, 1987.
3. N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

4. A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
5. V. Batagelj and M. Zaveršnik. An $O(m)$ algorithm for cores decomposition of networks, 2003. http://arxiv.org/abs/cs.DS/0310049.
6. R. Bauer, M. Krug, and D. Wagner. Enumerating and generating labeled k-degenerate graphs. In *7th Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 90–98. SIAM, 2010.
7. B. Bollobás. On generalized graphs. *Acta Mathematica Hungarica*, 16:447–452, 1965. 10.1007/BF01904851.
8. C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.
9. J. Cheng, Y. Ke, S. Chu, and T. Ozsu. Efficient core decomposition in massive networks. In *IEEE Int. Conf. on Data Engineering (ICDE)*, 2011.
10. M. Chrobak and D. Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theor. Comput. Sci.*, 86(2):243–266, 1991.
11. P. Doreian and K. L. Woodard. Defining and locating cores and boundaries of social networks. *Social Networks*, 16(4):267–293, 1994.
12. D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In O. Cheong, K.-Y. Chwa, and K. Park, editors, *ISAAC 2010*, volume 6506 of *LNCS*, pages 403–414. Springer-Verlag, 2010.
13. D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *arXiv eprint*, 1103.0318, 2011.
14. P. Erdős and A. Hajnal. On chromatic number of graphs and set-systems. *Acta Mathematica Hungarica*, 17(1–2):61–99, 1966.
15. D. Fernholz and V. Ramachandran. The giant k-core of a random graph with a specified degree sequence. *manuscript*, 2003.
16. E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29:24–32, January 1982.
17. P. A. Golovach and Y. Villanger. Parameterized complexity for domination problems on degenerate graphs. *Proc. 34th Int. Worksh. Graph-Theoretic Concepts in Computer Science (WG 2008)*, 5344:195–205, 2008.
18. S. Irani. Coloring inductive graphs on-line. *Algorithmica*, 11:53–72, 1994.
19. L. M. Kirousis and D. M. Thilikos. The linkage of a graph. *SIAM Journal on Computing*, 25(3):626–647, 1996.
20. J. Kleinberg. The small-world phenomenon: an algorithm perspective. In *32nd ACM Symp. on Theory of Computing (STOC)*, pages 163–170, 2000.
21. D. R. Lick and A. T. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22:1082–1096, 1970.
22. B. Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25:239–254, 1985.
23. B. Pittel, J. Spencer, and N. Wormald. Sudden emergence of a giant k-core in a random graph. *Journal of Combinatorial Theory, Series B*, 67(1):111–151, 1996.
24. O. Riordan. The k-core and branching processes. *Probability And Computing*, 17:111, 2008.
25. S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.
26. E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.
27. J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33:209–271, June 2001.

A Appendix

A.1 Proof of Lemma 2

The input is a d-degenerate graph $G = (V, E)$. Let $V = (1, \ldots, n)$. We store E as set of edges (u, v). We assume an order on V that we utilize during sorting.

Finding vertices of smallest degree (Algorithm 1, line 3) is realized by the `smallVertices` procedure.

```
proc smallVertices
1: for all $v$ – vertex of $G$ do
2:     $d(v) \leftarrow$ degree of $v$
3:     store pair $(d(v), v)$ in $F$
4: end for
5: sort $F$ lexicographically
6: $S \leftarrow$ first $n\epsilon/(2+\epsilon)$ vertices in $F$        // of smallest degree
```

Computing degrees of vertices in the `for` loop (lines 1-4) is easily realized in $O(sort(dn))$ I/O's. First, E is sorted lexicographically in $O(sort(dn))$ I/O's. After that, edges of E form blocks ordered by their starting vertex, so a simple scan taking $O(scan(dn))$ I/O's is enough to determine degrees of the vertices.

Sorting F (line 5) is done in $O(sort(n))$ I/O's. After that, S is just $n\epsilon/(2+\epsilon)$ first items of F and its construction (line 6) takes $O(scan(n\epsilon/(2+\epsilon)))$ I/O's. Likewise, appending S to L (Algorithm 1, line 4) takes $O(scan(n\epsilon/(2+\epsilon)))$ I/O's.

Finally, removing edges adjacent to S from G (Algorithm 1, line 5) is realized as follows.

```
1: sort $E$ lexicographically
2: for all $(u, v)$ – edge in $E$ do
3:     if $u \in S$ then
4:         add tuples $(u, v, "-")$ and $(v, u, "-")$ to $E$
5:     end if
6: end for
7: sort $E$ lexicographically
8: for all $p$, $q$ – consecutive tuples in $E$ do
9:     if $p = (u, v)$ and $q = (u, v, "-")$ then
10:        do not write $p$ back to $E$
11:    else if $p = (u, v)$ then
12:        write $p$ back to $E$
13:    else        $//p = (u, v, "-")$
14:        do not write $p$ back to $E$
15:    end if
16: end for
```

Sorting E in line 1 takes $O(sort(dn))$ I/O's. The first `for` loop (lines 2-6) takes $O(scan(dn))$ I/O's (vertices in S are stored according to the order on V, in the same relative order as the origins of edges in E, so it is realized by a single synchronized scan going through E and S at the same time). Each edge in E causes at most 2 tuples to be added to E in line 4, so clearly the size of E is $O(dn)$ after line 6. Sorting E (line 7) obviously takes time $O(sort(dn))$.

The last `for` loop (lines 8-16) is easily realized by a single scan of E, in $O(scan(dn))$ I/O's. Correctness follows from two facts. First, for each edge (u, v), if $u \in S$, the tuple $(u, v, \texttt{"-"})$ (meaning that (u, v) does not belong to G after this iteration) is added to E in line 4. Second, if $(u, v, \texttt{"-"})$ is in E, its direct predecessor is (u, v) (or another copy of $(u, v, \texttt{"-"})$) after E was sorted lexicographically in line 7. Therefore, the edges that are no longer in G are rejected in line 10. Also, no tuples $(u, v, \texttt{"-"})$ are further stored in E (line 14).

Altogether, lines 3-5 of Algorithm 1 are implemented in $O(sort(dn))$ I/O's.

$\square$

A.2 Representatives

Monien [22] gave a simple algorithm (which we call `repQuery`) that operates on representatives in the tree form described in Sec. 3. Given $\widehat{\mathcal{F}}$ – a representative for $\mathcal{F}$, `repQuery(`$\widehat{\mathcal{F}}$`, `B`)` decides for a q-set B whether there exists a set $A \in \mathcal{F}$ s.t. $A \cap B = \emptyset$, and returns such A if it exists. The running time of `repQuery` is $O(pq)$ in the RAM model. We use `repQuery` in our algorithms "as is", i.e., we allow it to take $O(pq)$ I/O's.

Proof of Lemma 3 The size of the resulting tree is is bounded by $\sum_{i=1}^{q} p^i$ (number of nodes in a p-ary tree of height q). We can afford to build the tree one node at a time, spending $O\big(scan(p|\mathcal{F}|)\big)$ I/O's on each node. Procedure `repLabel` labels vertex v in the representative tree for $\mathcal{F}$ in $O\big(scan(p|\mathcal{F}|)\big)$ I/O's.

```
proc repLabel($\mathcal{F}$, $v$)
 1: $\widehat{A} \leftarrow$ set $A$ in $\mathcal{F}$ s.t. $A \cap E(v) = \emptyset$
 2: if $\widehat{A} \neq \epsilon$ then
 3:     label $v$ with $\widehat{A}$
 4:     create children of $v$
 5:     label edges from $v$ to its children with elements from $\widehat{A}$
 6: else
 7:     label $v$ with $\lambda$
 8: end if
```

Set $\widehat{A}$ in line 1 can simply be found by scanning $\mathcal{F}$ in $O\big(scan(p|\mathcal{F}|)\big)$ I/O's ($p|\mathcal{F}|$ is the total size of all p-sets in $\mathcal{F}$). Creating children of v (line 4) and labeling their edges (line 5) takes $O(p)$ I/O's.

Therefore, we compute a q-representative for $\mathcal{F}$ in $O\Big(\big(\sum_{i=1}^{q} p^i\big)\cdot scan\big(p|\mathcal{F}|\big)\Big) = O\Big(\big(\sum_{i=1}^{q} p^{i+1}\big)\cdot scan\big(|\mathcal{F}|\big)\Big) = O\Big(\big(\sum_{i=1}^{q+1} p^{i}\big)\cdot scan\big(|\mathcal{F}|\big)\Big)$ I/O's. □

Proof of Lemma 4 The proof is essentially the same as that of Lemma 3.2 in [3]: first we compute a q-representative $\widehat{\mathcal{F}}$ of $\mathcal{F}$, in $O\Big(\big(\sum_{i=1}^{q+1}\big)\cdot scan\big(|\mathcal{F}|\big)\Big)$ I/O's, and a p-representative $\widehat{\mathcal{G}}$ of $\mathcal{G}$, in $O\Big(\big(\sum_{i=1}^{p+1}\big)\cdot scan\big(|\mathcal{G}|\big)\Big)$ I/O's.

The sizes of $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{G}}$ are bounded by $\sum_{i=1}^{q} p^i$ and $\sum_{i=1}^{p} q^i$, respectively. Assuming $p \geq q$ (w.l.o.g.), determining whether $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{G}}$ contain two disjoint sets can be easily done in $O\big(\sum_{i=1}^{q+1} p^i \cdot pq\big) = O\big(\sum_{i=1}^{q+1} p^{i+2}\big) = O\big(\sum_{i=1}^{q+3} p^i\big)$ I/O's, by querying $\widehat{\mathcal{G}}$ (via `repQuery`) with all sets from $\widehat{\mathcal{F}}$. □

Proof of Lemma 5 The (very) big picture of the `cycleThrough(`G`, `k`, `v`)` algorithm is as follows:

```
proc cycleThrough($G$, $k$, $v$)
1: for all $u$ s.t. $(v,u) \in E$ do
2:     if there exists simple path $p$: $u \rightsquigarrow v$ of length exactly $k-1$ then
3:         return $p \cup (v,u)$
4:     end if
5: end for
6: return $\epsilon$
```

Obviously, main difficulty lies in checking the condition in line 2. To show how we answer that query, let us first explain how [22] handles the paths.

Let $\mathcal{P}_{uv}^p$ denote the set of all simple paths from u to v of length exactly $(p+1)$ (so that these paths have p inner vertices). First, all paths from u to v of length $(p+1)$ that have the exact same set of inner vertices are represented as a single set containing these vertices (the vertices are stored as one of the paths; it provides a representative of the set). Performing this compression on $\mathcal{P}_{uv}^p$ yields $\mathcal{F}_{uv}^p$ – a family of p-sets:

$$\mathcal{F}_{uv}^p = \{S\colon S \text{ is a set of inner vertices on some path from } u \text{ to } v \text{ of length } p+1\}$$

The condition from line 2 of `cycleThrough` is therefore equivalent to $\mathcal{F}_{uv}^{k-2}$ being nonempty. We will now focus on how to test if this is the case.

The clou of [22] was that having q-representatives for $\mathcal{F}_{uv}^p$ (for all $u \in V$) enables efficient computation of $(q-1)$-representatives for $\mathcal{F}_{uv}^{p+1}$ (for all $u \in V$). The labels for a $(q-1)$-representative tree for $\mathcal{F}_{uv}^{p+1}$ are computed node by node. The algorithm is based on the following observation (γ is the node whose label we compute, $E(\gamma)$ is the set of edge labels on the way from γ to root):

$$\begin{gathered}\exists U \in \mathcal{F}_{uv}^{p+1} \text{ s.t. } U \cap E(\gamma) = \emptyset \\ \Longleftrightarrow \\ \exists w \in V \setminus \{u,v\} \text{ s.t. } (u,w) \in E \ \wedge \ w \notin E(\gamma) \ \wedge \\ \exists \widehat{U} \in \mathcal{F}_{wv}^{p} \text{ s.t. } \widehat{U} \cap (E(\gamma) \cup \{u\}) = \emptyset\end{gathered}$$

Having a q-representative for $\mathcal{F}_{wv}^{p}$ allows us to find $\widehat{U}$ (or determine that it does not exist) via the `repQuery` algorithm. Determining the label for γ is therefore realized as follows:

```
1: for all $w$ s.t. $(u,w) \in E$ and $w \notin E(\gamma)$ do
2:     $\widehat{U} \leftarrow$ repQuery($\mathcal{F}_{wv}^{p}$, $E(\gamma) \cup \{u\}$)
3:     if $\widehat{U} \neq \epsilon$ then
4:         label $\gamma$ with $\widehat{U} \cup \{w\}$
5:         return
6:     end if
7: end for
8: label $\gamma$ with $\lambda$
```

`repQuery` (querying a representative tree) takes $O(pq)$ I/O's, so our implementation of labeling γ takes $O\big(pq \cdot scan\big(\mathit{\Gamma}(u)\big)\big)$ I/O's (where $\mathit{\Gamma}(u)$ denotes the number of neighbors of u in G). It simply scans neighbors of u and calls `repQuery` accordingly. Because the $(q-1)$-representative tree for $\mathcal{F}_{uv}^{p+1}$ has size bounded by $\sum_{i=1}^{q-1}(p+1)^i \leq q(p+1)^{q+1}$, labeling all its nodes requires $O\Big(q(p+1)^{q-1} \cdot \big(pq \cdot scan\big(\mathit{\Gamma}(u)\big)\big)\Big) = O\Big(q^2(p+1)^q \cdot scan\big(\mathit{\Gamma}(u)\big)\Big)$ I/O's. We are computing $(q-1)$-representatives for $\mathcal{F}_{uv}^{p+1}$'s for all u's, so it takes $O\Big(\sum_u \big(q^2(p+1)^q \cdot scan\big(|\mathit{\Gamma}(u)|\big)\big)\Big) = O\Big(q^2(p+1)^q \cdot \sum_u scan\big(|\mathit{\Gamma}(u)|\big)\Big) = O\big(q^2(p+1)^q \cdot scan(m)\big)$ I/O's in total.

Our goal is to compute 0-representatives $\widehat{\mathcal{F}}_{uv}^{k-2}$ for $\mathcal{F}_{uv}^{k-2}$ (for all $u \in V$). Then, by calling `repQuery(`$\widehat{\mathcal{F}}_{uv}^{k-2}$`, `$\emptyset$`)`, we determine whether $\mathcal{F}_{uv}^{k-2}$ is nonempty, as it either returns ϵ (if $\mathcal{F}_{uv}^{k-2}$ is empty), or a set $A \in \mathcal{F}_{uv}^{k-2}$, representing a path from u to v of length $k-1$.

We start with $(k-2)$-representatives for $\mathcal{F}_{uv}^{0}$'s. They are built as trees having only the root vertex, labeled with either $\emptyset$ (if $(u,v) \in E$), or λ (otherwise). They can be obviously constructed in $O(scan(m))$ I/O's, via scanning E. Based on the discussion above, computing $\widehat{\mathcal{F}}_{uv}^{k-2}$'s (for all $u \in V$) takes

$$\begin{aligned} &O\big(\textstyle\sum_{p=0}^{k-3}(k-2-p)(p+1)^{k-2-p} \cdot scan(m)\big) \\ &= O\big((k-2) \cdot scan(m) \cdot \textstyle\sum_{p=0}^{k-3}(p+1)^{k-2-p}\big) \\ &= O\big((k-2) \cdot scan(m) \cdot (k-2)!\big) \\ &= O\big((k-1)! \cdot scan(m)\big)\end{aligned}$$

I/O's, as it can be easily shown by induction that $\sum_{p=0}^{k-3}(p+1)^{k-2-p} \leq (k-2)!$ for $k \geq 6$.

A.3 Path Generation

General Graphs (Algorithm 2) Recall that for Algorithm 2 we need to generate all directed paths of length k in a graph $G = (V, E)$, where maximum degree of each vertex is bounded by Δ. As shown in [3], there are at most $O(m\Delta^{k-1}) = O(m^{2-\frac{1}{k}})$ such paths. They are generated by the `pathGen` procedure.

```
proc pathGen
 1: generate all sequences of length $k-1$, with elements from $\{1, \ldots, \Delta\}$
 2: for all $s$ – sequence $\in \{1, \ldots, \Delta\}^{k-1}$ do
 3:     generate sequences $e|s$, for all $e \in E$
 4: end for
 5: for all $s$ – sequence $\in E \times \{1, \ldots, \Delta\}^{k-1}$ do
 6:     decode $s$ into $s' \in E^k$
 7:     if $s' \neq \epsilon$ then
 8:         if $s'$ is a simple path then
 9:             output $s'$
10:         end if
11:     end if
12: end for
```

Line 1 is simply realized in $O\big(scan(\Delta^{k-1})\big) = O\big(scan(m^{1-\frac{1}{k}})\big)$ I/O's. Adding an edge at the beginning of each sequence in the first `for` loop (lines 2-4) takes $O\big(m \cdot scan(\Delta^{k-1})\big) = O\big(scan(m^{2-\frac{1}{k}})\big)$ I/O's.

Decoding a sequence $s \in E \times \{1, \ldots, \Delta\}^{k-1}$ into a path $s' \in E^k$ (`pathGen`, line 6) is conceptually straightforward. e is the first edge in the path. Then, each consecutive number i determines next vertex on the path – ith neighbor of the previously decoded one (if it has less than i neighbors, the path is dropped as invalid). The `decodePaths` procedure handles decoding of S – the set of sequences. s`[i]` is the ith element of tuple s, s`[i].from` is the origin, and s`[i].to` is the destination vertex of edge at s`[i]`.

The first `for` loop (lines 1-3) decodes the starting edge of the sequence s into two vertices, and then cyclically shifts the resulting tuple by one position. It takes $O\big(scan(m^{2-\frac{1}{k}})\big)$ I/O's. Each iteration of the second `for` loop (lines 4-12) decodes the next vertex of s and again cyclically shifts s by one position. Sorting S in line 5 takes $O\big(sort(m^{2-\frac{1}{k}})\big)$ I/O's. After that, the inner loop (lines 6-11) requires only $O\big(scan(m^{2-\frac{1}{k}})\big)$ I/O's (it takes one synchronized scan of S and E). Invalid paths that do not meet the condition at line 8 are dropped. Since the outer `for` loop runs for $k-1$ iterations, `decodePaths` uses $O\big(k \cdot sort(m^{2-\frac{1}{k}})\big)$ I/O's.

```
proc decodePaths($S$)
1: for all $s$ – sequence $\in E \times \{1, \ldots, \Delta\}^{k-1}$ do
2:   write tuple ($s$[1].to, $s$[2], $s$[3], $\ldots$, $s$[k], $s$[1].from) to $S$
3: end for
4: for all $i \leftarrow 1, \ldots, k-1$ do      // $S$ contains tuples $V \times \{1, \ldots, \Delta\}^{k-i} \times V^i$
5:   sort $S$ lexicographically
6:   for all $s$ – tuple in $S$ do
7:     $u \leftarrow s$[2]th neighbor of $s[1]$ in $V$
8:     if $u \neq \epsilon$ then
9:       write tuple ($u$, $s$[3], $s$[4], $\ldots$, $s$[k+1], $s$[1]) back to $S$
10:    end if
11:  end for
12: end for
```

Verifying that a path is simple (pathGen, line 8) is done by checking that it does not contain repeated vertices. Thus, pathGen takes $O\big(sort(m^{2-\frac{1}{k}})\big)$ I/O's.

Degenerate Graphs (Algorithm 3) For Algorithm 3, we need to generate all paths of length $2k+1$ that start with two backward-oriented (in L) edges. As shown in [3], there are at most $O\big(m \sum_{i=0}^{k} \binom{2k}{i} \Delta^i \delta^{2k-i}\big) = O(2^{2k} m \Delta^k \delta^k)$ paths of length $2k+1$ in G. It follows from the fact that for each path p, either p or p^{R} (the reverse of p) has at most k edges with opposite directions than in L.

The procedure pathGenForward generates paths p that have at most k edges with opposite directions than in L.

```
proc pathGenForward
1: generate sequences $s = u|v$, for all $(u, v) \in E$
2: for $i \leftarrow 1 \ldots 2k$ do
3:   for all $s$ – sequence $\in V^2 \times (\{"L", "E"\} \times \{1, \ldots, \Delta\})^{i-1}$ do
4:     generate all sequences $s|("L", j)$, for $j \in \{1, \ldots, \delta\}$
5:     if $s$ has $< k$ pairs of the type $("E", j)$ then
6:       generate all sequences $s|("E", j)$, for $j \in \{1, \ldots, \Delta\}$
7:     end if
8:   end for
9: end for
10: for all $s$ – sequence $\in V^2 \times \big(\{"L", "E"\} \times \{1, \ldots, \max\{\delta, \Delta\}\}\big)^{2k}$ do
11:   decode $s$ into $s' \in E^{2k+1}$
12:   if $s' \neq \epsilon$ then
13:     if $s'$ is a simple path then
14:       output $s'$
15:     end if
16:   end if
17: end for
```

The encoding of the sequences works as follows: it starts with two vertices, u and v, that represent the starting edge of the path. v is followed by $2k$ pairs of the format $(\texttt{"L"}, i)$ (with $i \in \{1, \ldots, \delta\}$) or $(\texttt{"E"}, i)$ (with $i \in \{1, \ldots, \Delta\}$). $(\texttt{"L"}, i)$ means that the next vertex is the ith neighbor of the current vertex in degeneracy ordering L, and $(\texttt{"E"}, i)$ means that the next vertex is the ith neighbor of the current vertex in E. Any sequence s has at most k pairs of the type $(\texttt{"E"}, j)$, and only edges represented by them may have the opposite direction than in L.

The number of sequences generated by the first `for` loop (lines 2-9) is clearly $O\big(m \sum_{i=0}^{k} \binom{2k}{i} \Delta^i \delta^{2k-i}\big) = O(2^{2k} m \Delta^k \delta^k)$, and the whole generation process takes $O\big(scan(2^{2k} m \Delta^k \delta^k)\big)$ I/O's. The sequences are decoded into paths in the second `for` loop (lines 10-17) in a manner similar to `decodePaths`, using $O\big((2k+1) \cdot sort(2^{2k} m \Delta^k \delta^k\big)$ I/O's.

Paths of length $2k+1$, that have at most k edges in the opposite direction than in L when they are read backwards, can be generated by an analogous procedure `pathGenBackward` (using E^{R} – reversed edges instead of E), with the same I/O complexity.

Procedure `pathGen2` generates all paths of length $2k+1$ that start with two backward-oriented (in L) edges.

```
proc pathGen2
 1: generate paths of length $2k-1$, via pathGenForward
 2: generate paths of length $2k-1$, via pathGenBackward
 3: for all $s$ – generated path of length $2k-1$ do
 4:     generate all sequences $i|j|s$, for $(i,j) \in \{1,\ldots,\delta\}^2$
 5: end for
 6: for all $s$ – sequence $\in \{1,\ldots,d\}^2 \times E^{2k-1}$ do
 7:     decode $s$ into $s' \in E^{2k+1}$
 8:     if $s' \neq \epsilon$ then
 9:         if $s'$ is a simple path then
10:             output $s'$
11:         end if
12:     end if
13: end for
```

Paths generated in lines 1-2 are the *tails* of the resulting paths. The two numbers in sequences added to these paths in line 4 denote the edges in L that are to be taken to determine first two vertices on the final path, starting at the tail's first vertex. This assures that these edges are backward-oriented in L.

The number of paths generated by `pathGen2` is $O(\delta^2 \cdot 2^{2k-2} m \Delta^{k-1} \delta^{k-1}) = O(2^{2k-2} m \Delta^{k-1} \delta^{k+1}) = O(2^{2k-2} m^{2-\frac{1}{k}} \delta^{1+\frac{1}{k}})$, and its I/O complexity is $O\big(k \cdot sort(2^{2k-2} m^{2-\frac{1}{k}} \delta^{1+\frac{1}{k}})\big)$.

Since generating all paths of length $2k$ that begin with a backward-oriented (in L) edge is essentially the same as `pathGen2`, path generation in this case requires $O\big(k \cdot sort(2^{2k-2} m^{2-\frac{1}{k}} \delta^{1+\frac{1}{k}})\big)$ I/O's.

A.4 Graph Reordering

Assume we are given a graph $G = (V, E)$, with $V = (1, 2, \ldots, n)$, and d-degeneracy ordering $L = (v_1, v_2, \ldots, v_n)$ of V. Our goal is to *reorder* G according to L, i.e., substitute each edge $(v_i, v_j) \in E$ with an edge (i, j).

```
proc reorderG($G$, $L$)
 1: for all $k \leftarrow 1, 2$ do
 2:     for all $v_i$ – $i$th vertex in $L$ do
 3:         append tuple ($v_i$, "$i$") to $E$
 4:     end for
 5:     sort $E$ lexicographically
 6:     for all $p$ – tuple in $E$ do
 7:         if $p = (u, v)$ then
 8:             $q \leftarrow$ tuple ($u$, "$i$")        // precedes $p$
 9:             write $(v, i)$ back to $E$
10:         else
11:             do not write $(v, i)$ back to $E$
12:         end if
13:     end for
14: end for
```

A single iteration of the outer `for` loop (lines 1-14) first renames origins of edges in E and then reverts them (thus, after 2 iterations edges have their original directions). First, it adds vertices along with their positions in L to E in lines 2-4. This takes $O(scan(n))$ I/O's. Then it sorts E (line 5) in $O(sort(dn))$ I/O's. The next `for` loop (lines 6-13) scans E, renames origins of edges to their positions in L and outputs their opposite versions. The tuples q obtained in line 8 can clearly be found in the same single scan (tuple (u, "i") directly precedes all edges (u, v) after E was sorted), so this part is done in $O(scan(dn))$ I/O's.

Therefore, `reorderG` runs in $O(sort(dn))$ I/O's altogether.

A.5 Proof of Lemma 6

First, observe that the total size of all sets P and X passed to initial calls to `BronKerboschPivot` is $O(m) = O(\delta n)$. To see this, note that every edge (v_i, v_j) (with v_i preceding v_j in $L = (v_1, v_2, \ldots, v_n)$) puts v_i into initial X for $R = \{v_j\}$ and v_j into initial P for $R = \{v_i\}$. Since the size of each P is at most δ, the total number of edges in all $H_{P,X}$'s is $O(\delta^2 n)$.

Our approach is to generate and store all P's, X's and $H_{P,X}$'s at the very beginning of the algorithm, and then just pass appropriate P, X, and $H_{P,X}$ to each initial call to `BronKerboschPivot`.

Procedure `genPX` generates all initial P's and X's in $O(sort(\delta n))$ I/O's. We assume that for each vertex $v \in V$ we know its position in L (i.e., we know i for $v = v_i$). Also, for each edge $(u, v) \in E$, we know positions in L of its endpoints (i.e., we know both i and j for $(u, v) = (v_i, v_j)$). We can easily achieve this in $O(sort(\delta n))$ I/O's (see Sec. A.4). By $u \overset{L}{\prec} v$, we denote that u precedes v in L. The `for` loop (lines 1-5) clearly takes $O(scan(\delta n))$ I/O's.

```
proc genPX(G, L)
 1: for all $(v_i, v_j)$ – edge in $E$ do
 2:     if $v_i \overset{L}{\prec} v_j$ then
 3:         output tuples $(v_i, "P", v_j)$, and $(v_j, "X", v_i)$ to set $F$
 4:     end if
 5: end for
 6: sort $F$ lexicographically
 7: scan $F$ to create sets $P$ and $X$
```

The meaning of $(v_i, "P", v_j)$ is "add v_j to P_{v_i}", and of $(v_j, "X", v_i)$: "add v_i to X_{v_j}". The above discussion explains why this information allows to generate all P's and X's. Therefore, after set F is sorted in $O(sort(\delta n))$ I/O's in line 6, generation of P's and X's in line 7 takes $O(scan(\delta n))$ I/O's.

Procedure `genH` generates all initial $H_{P,X}$'s in $O(sort(\delta^2 n))$ I/O's. Each vertex has at most δ neighbors that succeed it in L, so there are $O(\delta^2 n)$ tuples added to E in the first `for` loop (lines 1-5). They can be generated in $O(scan(\delta^2 n))$ I/O's, as the edges in E are sorted lexicographically.

```
proc genH(G, L)
 1: for all $u$ – vertex in $V$ do
 2:     for all $v, w$ – neighbors of $u$ s.t. $u \overset{L}{\prec} v \overset{L}{\prec} w$ do
 3:         append tuple $(v, w, ?u?)$ to $E$
 4:     end for
 5: end for
 6: sort $E$ lexicographically
 7: for all $p$ – tuple in $E$ do
 8:     if $p = (v, w, ?u?)$ then
 9:         $q \leftarrow$ tuple of form $(v_i, v_j)$ immediately preceding $p$ in $E$
10:         if $q = (v, w)$ then
11:             output tuples $(H_v, u, w)$ and $(H_v, w, u)$ to set $H$
12:         end if
13:         do not write $p$ back to $E$
14:     else    $//p = (v, w)$
15:         write $p$ back to $E$
16:     end if
17: end for
18: sort $H$ lexicographically
19: scan $H$ to create sets $H_v$
20: scan edges of $H_v$'s and mark the endpoints that belong to $P_v$'s
```

To understand the meaning of tuples from line 3, refer to Fig. 2.

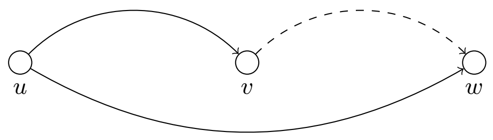

Fig. 2. v and w are neighbors of u, and $u \overset{L}{\prec} v \overset{L}{\prec} w$, so the tuple $(v, w, ?u?)$ is output in `genH`, line 3. If $(v, w) \in E$, the edge (u, w) is in H_v.

After E is sorted in line 6 in $O(sort(\delta^2 n))$ I/O's, the next `for` loop (lines 7-17) identifies edges that belong to sets H_v. In E, each edge (v, w) is followed by zero or more tuples of the form $(v, w, ?u?)$. This makes it easy to determine q in line 9, and therefore, the loop takes $O(scan(\delta^2 n))$ I/O's.

As explained in Fig. 2, (v, w) followed by $(v, w, ?u?)$ means that the edge (u, w) has to be added to H_v. It is denoted by the tuples (H_v, u, w) and (H_v, w, u) output in line 11. Sorting H in line 18 takes $O(sort(\delta^2 n))$ I/O's, and after that, $H_{P,X}$'s are generated in line 19 in $O(scan(\delta^2 n))$ I/O's. The marking of endpoints (line 20) also takes $O(scan(\delta^2 n))$ I/O's.

Therefore, total complexity of generating initial P's, X's, and $H_{P,X}$'s is $O(sort(\delta^2 n))$ I/O's. □

A.6 Proof of Lemma 7

```
proc updateH($v$)
 1: for all $H_{P,X}$ − candidate do
 2:     for all $e$ − edge in $H_{P,X}$ do
 3:         if $e = (u, v)$ or $e = (v, w)$ then
 4:             unmark $v$ in $e$
 5:         end if
 6:     end for
 7:     for all $e$ − edge in $H_{P,X}$ do
 8:         if at least one vertex of $e$ is marked then
 9:             write $e$ back to $H_{P,X}$
10:         else
11:             do not write $e$ back to $H_{P,X}$
12:         end if
13:     end for
14: end for
```

Recall that $|P| = p$ and $|X| = x$. Our idea in computing $H_{P,X}$'s is to first generate *candidates* for $H_{P,X}$'s. *Candidates* are defined as $H_{P,X}$'s as they would be if there were no lines 7 and 8 in `BronKerboschPivot` (i.e., as if P and X did not change). We then update the *candidates* according to lines 7 and 8 of `BronKerboschPivot`.

Generation of the *candidates* for $H_{P,X}$'s is almost the same as in `genH`, only v and w in line 2 are now taken from P, so it uses $O(sort(p^2(p+x)))$ I/O's. Updating $H_{P,X}$'s (moving v from P to X) is realized by the procedure `updateH`.

Both inner `for` loops (lines 2-6 and 7-13) clearly take $O(scan(|H_{P,X}|))$ I/O's. Since the total size of all *candidates* is $O(p^2(p+x))$, a single call to `updateH` takes $O(scan(p^2(p+x)))$ I/O's. There are at most p such calls, so generating $H_{P,X}$'s that are passed to recursive calls in line 6 of `BronKerboschPivot` takes $O(sort(p^2(p+x)) + p \cdot scan(p^2(p+x))) = O(sort(\delta p^2(p+x)))$ I/O's. $\square$