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* [26], which is equivalent to a graph's *width* [16], *linkage* [20], *k-inductivity* [19], and *k-degeneracy* [2,22], 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.* [24] 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 [25] 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 [21], 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 [28]). 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 [28].

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 $m \leq dn$ 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 [1,3,12,17,19]). 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 $O\left(d^{1\pm\epsilon} \cdot \left(k \cdot sort(m^{2-\frac{1}{k}}) + (4k)! \cdot scan(m^{2-\frac{1}{k}})\right)\right)$ I/O complexity, where ϵ is a constant depending on $c \in \{4k - 2, \dots, 4k + 1\}$.

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 ← ∅
2: while G is nonempty do
3:     S ← nε/(2 + ε) vertices of smallest degree in G
4:     L ← 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 full version of this paper [18] contains proof of the following lemma.

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

$$I(G) = O\left(sort(dn) + sort\left((2/(2+\epsilon))dn\right) + sort\left((2/(2+\epsilon))^2 dn\right) + \cdots\right)$$
$$= O\left(sort(dn)\left(1 + \frac{2}{2+\epsilon} + \left(\frac{2}{2+\epsilon}\right)^2 + \cdots\right)\right)$$
$$= O(sort(dn)). \qquad \qquad \square$$

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 [23]. 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 [23] 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 labeled with A, then $A \cap E(v) = \emptyset$
  - if v 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,
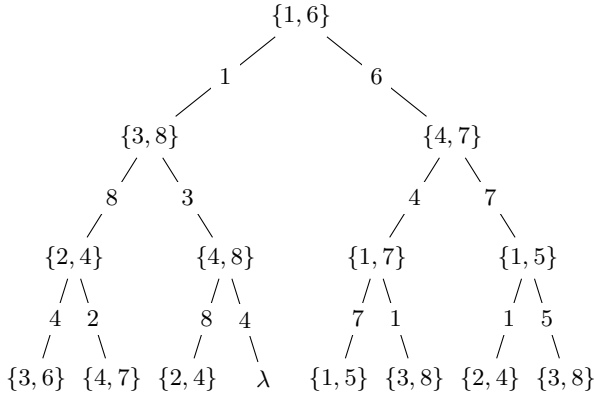
$$\{1,6\}$$

$$1 \qquad\qquad 6$$

$$\{3,8\} \qquad\qquad \{4,7\}$$

$$8 \quad 3 \qquad\qquad 4 \quad 7$$

$$\{2,4\} \qquad \{4,8\} \qquad \{1,7\} \qquad \{1,5\}$$

$$4 \quad 2 \qquad 8 \quad 4 \qquad 7 \quad 1 \qquad 1 \quad 5$$

$$\{3,6\} \ \{4,7\} \ \{2,4\} \qquad \lambda \qquad \{1,5\} \ \{3,8\} \ \{2,4\} \ \{3,8\}$$

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

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.

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 full version of this paper [18] 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 [18].

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 $O\left(\left(\sum_{i=1}^{q+1} p^i\right) \cdot scan\left(|\mathcal{F}|\right)\right)$ 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 $O\left(\left(\sum_{i=1}^{q+3} p^i + \sum_{i=1}^{p+3} q^i\right) \cdot scan\left(|\mathcal{F}| + |\mathcal{G}|\right)\right)$ 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** ϵ

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 [18]). 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 ≺ 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 (F_uv, F_uv)
```

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

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

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. $\square$

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.

1: $\Delta \leftarrow m^{\frac{1}{k}}/\delta^{1+\frac{1}{k}}$
2: **for all** v – vertex of degree $\geq \Delta$ **do**
3:     $C \leftarrow \texttt{cycleThrough}(G, 4k + 1, v)$
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:     $P \leftarrow \texttt{findDisjoint}(\mathcal{F}_{uv}, \mathcal{G}_{vu})$
15:     **if** $P = (A, B)$ **then**
16:         **return** $C = A \cup B$
17:     **end if**
18: **end for**
19: **return** ϵ

Algorithm 3. Short cycles in degenerate graphs

Algorithm 3 is remarkably similar to Algorithm 2 and so is its analysis. Differences lie in the value of Δ 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 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 [18]), 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.* [27]. 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 ∪ X = ∅ then
 2:    output R      //maximal clique
 3: end if
 4: u ← vertex from P ∪ X that maximizes |P ∩ Γ(u)|
 5: for all v ∈ P \ Γ(v) do
 6:    BronKerboschPivot(P ∩ Γ(v), R ∪ {v}, X ∩ Γ(v))
 7:    P ← P \ {v}
 8:    X ← X ∪ {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 ← 1 . . . n do
 2:    P ← Γ(vi) ∩ {vj : j > i}
 3:    X ← Γ(vi) ∩ {vj : j < i}
 4:    BronKerboschPivot(P, {vi}, 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.* [27].

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:*

$$V_{P,X} = P \cup X$$
$$E_{P,X} = \{(u,v) \colon (u,v) \in E \land (u \in P \lor v \in P)\}$$

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 $H_{P,X}$).

Lemmas 6 and 7 are proved in the full version of this paper [18]. In the following, $\delta = (2 + \epsilon)d$.

Lemma 6. *Given a δ-degeneracy ordering L of an undirected d-degenerate graph G, 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, 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, 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:

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

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

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

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

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

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 $|P| \leq \delta$. Thus, 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. $\square$

References

1. Alon, N., Gutner, S.: Linear time algorithms for finding a dominating set of fixed size in degenerated graphs. Algorithmica 54(4), 544–556 (2009)
2. Alon, N., Kahn, J., Seymour, P.D.: Large induced degenerate subgraphs. Graphs and Combinatorics 3, 203–211 (1987)
3. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica 17(3), 209–223 (1997)
4. Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. Science 286(5439), 509–512 (1999)
5. Batagelj, V., Zaveršnik, M.: An O(m) algorithm for cores decomposition of networks (2003), http://arxiv.org/abs/cs.DS/0310049
6. Bauer, R., Krug, M., Wagner, D.: Enumerating and generating labeled k-degenerate graphs. In: 7th Workshop on Analytic Algorithmics and Combinatorics (ANALCO), pp. 90–98. SIAM, Philadelphia (2010)
7. Bollobás, B.: On generalized graphs. Acta Mathematica Hungarica 16, 447–452 (1904), doi:10.1007/BF01904851
8. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. Commun. ACM 16(9), 575–577 (1973)
9. Cheng, J., Ke, Y., Chu, S., Ozsu, T.: Efficient core decomposition in massive networks. In: IEEE Int. Conf. on Data Engineering, ICDE (2011)
10. Chrobak, M., Eppstein, D.: Planar orientations with low out-degree and compaction of adjacency matrices. Theor. Comput. Sci. 86(2), 243–266 (1991)
11. Doreian, P., Woodard, K.L.: Defining and locating cores and boundaries of social networks. Social Networks 16(4), 267–293 (1994)
12. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in sparse graphs in near-optimal time. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010. LNCS, vol. 6506, pp. 403–414. Springer, Heidelberg (2010)
13. Eppstein, D., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. arXiv eprint, 1103.0318 (2011)
14. Erdős, P., Hajnal, A.: On chromatic number of graphs and set-systems. Acta Mathematica Hungarica 17(1-2), 61–99 (1966)
15. Fernholz, D., Ramachandran, V.: The giant k-core of a random graph with a specified degree sequence (2003) (manuscript)
16. Freuder, E.C.: A sufficient condition for backtrack-free search. J. ACM 29, 24–32 (1982)
17. Golovach, P.A., Villanger, Y.: Parameterized complexity for domination problems on degenerate graphs. In: Broersma, H., Erlebach, T., Friedetzky, T., Paulusma, D. (eds.) WG 2008. LNCS, vol. 5344, pp. 195–205. Springer, Heidelberg (2008)
18. Goodrich, M.T., Pszona, P.: External-memory network analysis algorithms for naturally sparse graphs. arXiv eprint, 1106.6336 (2011)
19. Irani, S.: Coloring inductive graphs on-line. Algorithmica 11, 53–72 (1994)
20. Kirousis, L.M., Thilikos, D.M.: The linkage of a graph. SIAM Journal on Computing 25(3), 626–647 (1996)
21. Kleinberg, J.: The small-world phenomenon: an algorithm perspective. In: 32nd ACM Symp. on Theory of Computing (STOC), pp. 163–170 (2000)
22. Lick, D.R., White, A.T.: k-degenerate graphs. Canadian Journal of Mathematics 22, 1082–1096 (1970)
23. Monien, B.: How to find long paths efficiently. Annals of Discrete Mathematics 25, 239–254 (1985)

24. Pittel, B., Spencer, J., Wormald, N.: Sudden emergence of a giant k-core in a random graph. Journal of Combinatorial Theory, Series B 67(1), 111–151 (1996)
25. Riordan, O.: The k-core and branching processes. Probability And Computing 17, 111 (2008)
26. Seidman, S.B.: Network structure and minimum degree. Social Networks 5(3), 269–287 (1983)
27. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. Theor. Comput. Sci. 363(1), 28–42 (2006)
28. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. 33, 209–271 (2001)