



## Certifying Induced Subgraphs in Large Graphs

Ulrich Meyer<sup>1,2</sup>  Hung Tran<sup>1,2</sup> Konstantinos Tsakalidis<sup>3,4</sup> 

<sup>1</sup>Goethe University Frankfurt, Germany

<sup>2</sup>Frankfurt Institute for Advanced Studies, Germany

<sup>3</sup>University of Liverpool, United Kingdom

<sup>4</sup>Archimedes Research Unit, Athena Research Center, Greece

Submitted: June 2023

Accepted: May 2024

Published: September  
2024

Article type: Regular paper

Communicated by: C.-C. Lin, B. M.-T.  
Lin, G. Liotta

**Abstract.** We introduce I/O-efficient certifying algorithms for the recognition of bipartite, split, threshold, bipartite chain, and trivially perfect graphs. When the input graph is a member of the respective class, the certifying algorithm returns a certificate that characterizes this class. Otherwise, it returns a forbidden induced subgraph as a certificate for non-membership. On a graph with  $n$  vertices and  $m$  edges, our algorithms take  $\mathcal{O}(\text{sort}(n + m))$  I/Os in the worst case for split, threshold and trivially perfect graphs. In the same complexity bipartite and bipartite chain graphs can be certified with high probability. We provide implementations and an experimental evaluation for split and threshold graphs.

## 1 Introduction

*Certifying algorithms* [28] ensure the correctness of an algorithm's output without having to trust the algorithm itself. The user of a certifying algorithm inputs  $x$  and receives the output  $y$  with a *certificate* or *witness*  $w$  that proves that  $y$  is a correct output for input  $x$ . In a subsequent step, the certificate can be inspected using an authentication algorithm that considers the input, output and certificate and returns whether the output is indeed correct. Certifying the bipartiteness of a graph is a textbook example where the returned witness  $w$  is a bipartition of the vertices (YES-certificate) or an *odd-length cycle* subgraph, i.e. a cycle of vertices with an odd number of edges (NO-certificate). In this case, the authentication algorithm either verifies that all edges have endpoints in both bipartition classes, or verifies that the returned cycle is indeed an odd-length cycle, see [Figure 1](#) for an illustration.

---

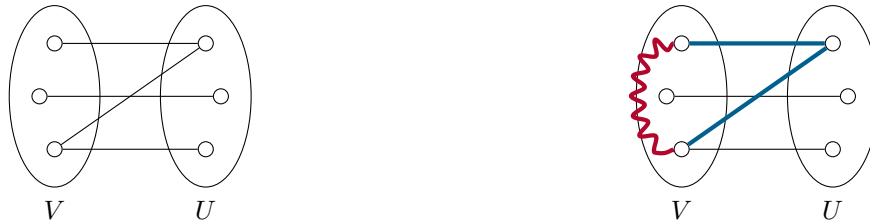
Konstantinos Tsakalidis was supported by the International Exchanges Grant IES\R3\203041 of the Royal Society. Ulrich Meyer and Hung Tran were supported by the Deutsche Forschungsgemeinschaft (DFG) under grant ME 2088/5-1 (FOR 2975 — Algorithms, Dynamics, and Information Flow in Networks) and the LOEWE programme of the state Hesse (CMMS). A preliminary version of this paper appeared in WALCOM 2023 [33].

---

*E-mail addresses:* [umeyer@ae.cs.uni-frankfurt.de](mailto:umeyer@ae.cs.uni-frankfurt.de) (Ulrich Meyer) [htran@ae.cs.uni-frankfurt.de](mailto:htran@ae.cs.uni-frankfurt.de) (Hung Tran) [K.Tsakalidis@liverpool.ac.uk](mailto:K.Tsakalidis@liverpool.ac.uk) (Konstantinos Tsakalidis)



This work is licensed under the terms of the [CC-BY](#) license.



(a) In the positive case, the bipartition classes  $(V, U)$  serve as a **YES**-certificate.

(b) In the negative case, an odd length cycle serves as a **NO**-certificate.

Figure 1: Illustration of the respective certificates when recognizing bipartite graphs.

Emerging big data applications, spanning domains such as databases, geographical information systems, bioinformatics, network analysis and beyond, require efficient processing of massive datasets. Further applications consider computing regimes with limited internal memory, e.g. mobile devices and (swarm) robotics with a central server. Standard models of computation in internal memory (RAM, pointer machine) do not capture the algorithmic complexity of processing data with sizes that exceed the main memory. The *I/O-model* by Aggarwal and Vitter [1] is suitable for studying large data stored in an external memory hierarchy, e.g. comprised of cache, RAM and hard disk memories. The input data elements are stored in *external memory* (EM) packed in *blocks* of at most  $B$  elements and computation is free in *main memory* for at most  $M$  elements. The *I/O-complexity* is measured in *I/O-operations* (*I/Os*) that transfer a block from external to main memory and vice versa. Common tasks of many algorithms include reading or writing  $n$  contiguous items (which is referred to as scanning) requiring  $\text{scan}(n) := \Theta(n/B)$  I/Os and sorting  $n$  consecutive elements<sup>1</sup> requiring  $\text{sort}(n) := \Theta((n/B) \log_{M/B}(n/B))$  I/Os.

## 1.1 Previous Work

There has been extensive work on certifying algorithms with different applications, including property testing [31, 27], graph problems [12, 13, 6], graph recognition, e.g. most famously planarity tests [22] and many more [23, 15, 21, 36, 10, 34, 20] (see also the survey of McConnell et al. [28]). Most of the aforementioned references are algorithms for the standard model of computation, and to the best of our knowledge none exist for the external memory model.

Most internal memory graph recognition algorithms use techniques that are in a direct translation inapplicable to the external memory model. This can be exemplified by the simple problem of bipartiteness testing where a graph traversal, e.g. depth-first or breadth-first search, can produce the correct output with a certificate in linear time. In external memory, however, breadth-first search [29, 2, 32] and depth-first search [7, 4] algorithms take sub-optimal  $\omega(\text{sort}(n+m))$  I/Os for an input graph with  $n$  vertices and  $m$  edges.

Nevertheless, some of the aforementioned algorithms follow a commonly used paradigm that may be efficiently exploited in external memory, see [30, 20, 24] for a few examples where the paradigm is used. It consists of a few essential parts<sup>2</sup>: incrementally consider the vertices in a judicious order  $(v_1, \dots, v_n)$ , if the induced subgraph  $G_{i+1}$  of the first  $(i+1)$  vertices does not have a desired property then search for a forbidden subgraph in  $G_{i+1}$  based on the addition of  $v_{i+1}$ .

<sup>1</sup>This is typically implemented by a  $k$ -way Mergesort with a suitable  $k$  leading to the  $\text{sort}(n)$  complexity.

<sup>2</sup>These requirements essentially describe a pseudo-incremental semi-certifying recognition algorithm.

This requires a constructive model depending on  $G_i$ .

Heggernes and Kratsch [20] present optimal internal memory algorithms for certifying whether a graph belongs to the classes of split, threshold, bipartite chain, and trivially perfect graphs following this paradigm. They return in linear time a YES-certificate characterizing the corresponding class or a forbidden induced subgraph of the class (NO-certificate). The YES- and NO-certificates are authenticated in linear and constant time, respectively. A straightforward application to the I/O-model leads to suboptimal certifying algorithms since graph traversal algorithms in external memory are much more involved and no worst-case efficient algorithms are known.

## 1.2 Our Results

We present I/O-efficient certifying algorithms for

- split,
- threshold,
- bipartite,
- bipartite chain, and
- trivially perfect graphs.

All algorithms return in the membership case, a YES-certificate  $w$  characterizing the graph class, or a  $\mathcal{O}(1)$ -size NO-certificate in the non-membership case. The YES- and NO-certificates can be authenticated using  $\mathcal{O}(\text{sort}(n+m))$  and  $\mathcal{O}(1)$  I/Os, respectively. As a subroutine for the certification of bipartite chain graphs we develop a certifying algorithm to recognize bipartite graphs using  $\mathcal{O}(\text{sort}(n+m))$  I/Os with high probability. Additionally, we perform experiments for split and threshold graphs showing scaling well beyond the size of main memory.

## 2 Preliminaries and Notation

For a graph  $G = (V, E)$ , let  $n = |V|$  and  $m = |E|$  denote the number of vertices in  $V$  and edges in  $E$ , respectively. We assume that the vertices  $V = \{v_1, \dots, v_n\}$  are ordered by their indices. For a vertex  $v \in V$  we denote by  $N(v)$  the *neighborhood* of  $v$  and by  $N[v] = N(v) \cup \{v\}$  the *closed neighborhood* of  $v$ . The *degree*  $\text{deg}(v)$  of a vertex  $v$  is given by  $\text{deg}(v) = |N(v)|$ . A vertex  $v$  is called *simplicial* if  $N(v)$  is a clique and *universal* if  $N[v] = V$ .

**Subgraphs and Orderings** The subgraph of  $G$  that is induced by a subset  $A \subseteq V$  of vertices is denoted by  $G[A]$ . The *substructure* (subgraph) of a cycle on  $k$  vertices is denoted by  $C_k$  and of a path on  $k$  vertices is denoted by  $P_k$ . The  $2K_2$  is a graph that is isomorphic to the following constant size graph:  $(\{a, b, c, d\}, \{ab, cd\})$ .

All algorithms reorganize the initial input by a suitable ordering of the vertices. For any given vertex ordering  $\sigma$ , we partition the set of neighbors  $N(v)$  into  $L(v) = \{x \in N(v) : v \text{ is ranked higher than } x \text{ in } \sigma\}$  and  $H(v) = \{x \in N(v) : v \text{ is ranked lower than } x \text{ in } \sigma\}$  where  $L(v)$  and  $H(v)$  denote the lower and higher ranked neighbors, respectively.

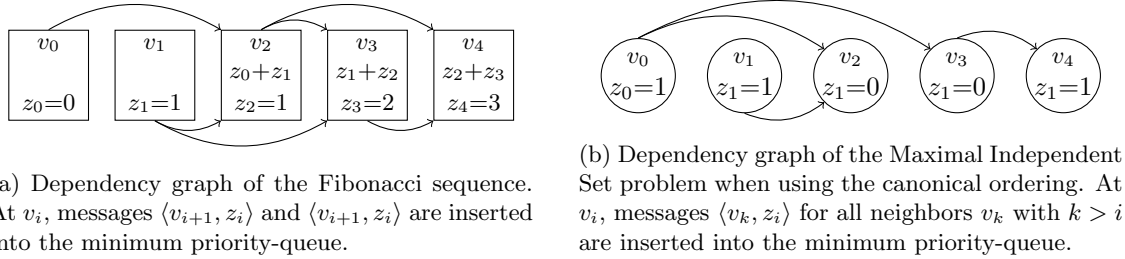


Figure 2: Message forwarding scheme using time-forward processing.

**Graph Relabeling** A *relabeling* of a graph  $G = (V, E)$  is defined by a bijection  $f : V \rightarrow V$  where each edge  $\{u, v\} \in E$  is reflected by an edge  $\{f(u), f(v)\}$  of relabeled endpoints. For an ordering  $\alpha = (u_1, \dots, u_n)$ , a relabeling of  $G$  by  $\alpha$  corresponds to the mapping where each  $v_i$  is mapped to its rank in  $\alpha$ , e.g.  $f(v_i) = v_r$  where  $r$  is the rank of  $v_i$  in  $\alpha$ .

Employing this subroutine can lead to a more suitable representation of the graph in memory and often allows for more efficient data processing. The relabeling can be done I/O-efficiently in a constant number of scanning and sorting steps incurring  $\mathcal{O}(\text{sort}(n+m))$  I/Os [5]. As all our algorithms perform an initial relabeling according to some ordering, we use the vertex labels obtained by this initial relabeling.

**Graph Representation** We assume an *adjacency array representation* [35] where the graph  $G = (V, E)$  is represented by two arrays  $P = [P_i]_{i=1}^n$  and  $E = [u_i]_{i=1}^m$ , where  $P$  *points* to a location of the *edge-list* of  $G$ . The neighbors of a vertex  $v_i$  are then given in sorted order by the vertices at position  $P_i$  to  $P_{i+1}-1$  in  $E$ . This representation allows for efficient straight-forward processing of  $G$ : (i) scanning  $k$  consecutive adjacency lists consisting of  $m'$  edges requires  $\mathcal{O}(\text{scan}(m'))$  I/Os and (ii) computing and scanning the degrees of  $k$  consecutive vertices requires  $\mathcal{O}(\text{scan}(k))$  I/Os.

### 3 Certifying Graph Classes in External Memory

Our algorithms follow the same base structure with further details depending on the considered graph class. In order to allow for more data locality, the vertices of the input graph are first relabeled according to some vertex ordering that characterizes the graph class. Then, on the resulting graph representation, computation is carried out in the order of the used vertex ordering leading to significantly less I/Os. Typically, in order to certify the required properties of the graph class, when iterating over all vertices, additional information beyond its own neighbors is required and has to be provided I/O-efficiently at the time of processing. For this, the main algorithmic technique used is called time-forward processing and will be later illustrated using two examples. *Time-forward processing (TFP)* is a generic technique to manage data dependencies of external memory algorithms [26]. These dependencies are typically modeled by a directed acyclic graph  $G = (V, E)$  where every vertex  $v_i \in V$  models the computation of a corresponding value  $z_i$  and an edge  $(v_i, v_j) \in E$  indicates that  $z_i$  is required for the computation of  $z_j$ .

Computing a solution then requires the algorithm to traverse  $G$  according to some topological order  $\prec_T$  of the vertices  $V$ . The TFP technique achieves this in the following way: after  $z_i$  has been

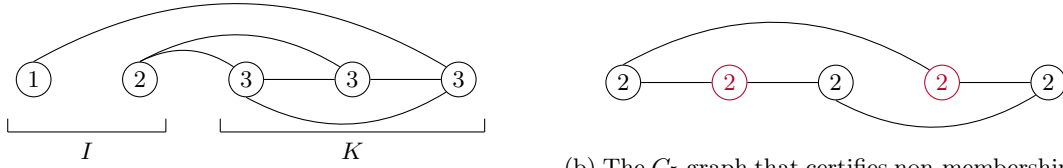
calculated, the algorithm inserts a message  $\langle v_j, z_i \rangle$  into a minimum priority-queue data structure for every successor  $(v_i, v_j) \in E$  where the items are sorted by the recipients according to  $\prec_T$ . By construction,  $v_j$  receives all required values  $z_i$  of its predecessors  $v_i \prec_T v_j$  as messages in the data structure. Since these predecessors already removed their messages from the priority-queue, items addressed to  $v_j$  are currently the smallest elements in the data structure and thus can be dequeued with a delete-minimum operation. By using suitable external memory priority-queues [3], TFP incurs  $\mathcal{O}(\text{sort}(k))$  I/Os, where  $k$  is the number of messages sent.

We provide two simple examples. First, consider the computation of the Fibonacci sequence  $z_0 = 0, z_1 = 1$  and  $z_i = z_{i-1} + z_{i-2}$  for all  $i \geq 2$  where each node  $v_i$  with  $i \geq 2$  depends on only its two predecessors, see Figure 2a. While a linear scan over increasing  $i$  suffices to solve the dependencies, it serves as an instructive example showing how messages at  $v_i$  are used to compute the value of  $z_i$  and how to forward it to later vertices. Second, a more complex problem is to find a Maximal Independent Set. This can be realized by a simple greedy algorithm that iterates over any ordering of the vertices and extending a currently computed independent set  $I$ , if possible. In order to I/O-efficiently verify whether candidate vertices are eligible, TFP can be employed to forward the status of a vertex to its neighbors, see Figure 2b. For this, let  $z_i$  represent whether  $v_i$  is in the independent set  $I$ , i.e.  $z_i = 1$  if  $v_i \in I$  and  $z_i = 0$  otherwise. When processing  $v_i$ , messages of the form  $\langle v_i, z_j \rangle$  are accumulated and evaluated where  $v_j$  is a neighbor of  $v_i$ . If any of the received values  $z_j$  is equal to 1 then  $v_i$  is ineligible as a neighboring vertex is already present in  $I$ . In this case, the value  $z_i = 0$  is forwarded to future neighbors and  $z_i = 1$  otherwise. Contrary to the first example, the use of the priority-queue alone is insufficient, as this algorithm has to perform a concurrent scan over the edges while processing the vertices  $v_i$  to forward messages to the neighbors. Note that, messages of the form  $\langle v_i, z_j = 0 \rangle$  can be omitted entirely and only serve an instructive purpose in this example.

### 3.1 Split Graphs

A split graph is a graph that can be partitioned into two sets of vertices  $(K, I)$  where  $K$  and  $I$  induce a clique and an independent set, respectively. The partition  $(K, I)$  is called the *split partition*. They are additionally characterized by the forbidden induced subgraphs  $2K_2, C_4$  and  $C_5$ , meaning that any vertex subset of a split graph cannot induce these substructures [18]. Remarkably, any split graph  $G$  can be solely recognized by its degree sequence  $d = (d_1, d_2, \dots, d_n)$  [19] where a group of the highest degree vertices forms the clique  $K$ , however, without providing a certificate. In fact, any non-decreasing degree ordering of  $G$  is a perfect elimination ordering [16, 20], providing enough structure to enable time-forward processing as a means to find a certificate I/O-efficiently. An ordering  $(u_1, \dots, u_n)$  is a *perfect elimination ordering (peo)* if  $u_i$  is simplicial in  $G[\{u_i, u_{i+1}, \dots, u_n\}]$  for all  $i \in \{1, \dots, n\}$ . Intuitively, a non-decreasing degree ordering that is a perfect elimination ordering reflects some of the desired properties directly. Given a vertex  $v \in I$ , any neighbor  $u$  of  $v$  cannot be in  $I$  as  $I$  is an induced independent set. As such  $u$  has to be part of the clique  $K$ , in particular  $N(u) \setminus I$  has to be a clique itself, therefore  $u$  has to be simplicial in its corresponding subgraph given by the perfect elimination ordering. Similarly, for any vertex  $v \in K$  all higher ranked vertices in the degree ordering have to be a part of the clique  $K$  as only the highest degree vertices actually form the clique, see Figure 3.

Our algorithm adapts the internal memory certifying algorithm of Heggenes and Kratsch [20] to external memory by adopting time-forward processing in the critical subroutines as described below. As output it either returns the split partition  $(K, I)$  as a YES-certificate or one of the forbidden substructures  $C_4, C_5$  or  $2K_2$  as a NO-certificate. We present the certifying algorithm and



(a) Example of a split graph where the vertices are ordered non-decreasingly by degree from left to right. The ordering is a perfect elimination ordering.

(b) The  $C_5$  graph that certifies non-membership for split graphs. Here, the leftmost vertex invalidates the perfect elimination ordering. Its neighbors are not a clique since they are not connected, therefore the first vertex is not simplicial.

Figure 3: Example visualization of a non-decreasing degree ordering for a split graph in (a) and one of its forbidden induced subgraphs in (b).

its corresponding authentication algorithm and provide details in [Proposition 1](#) and [Proposition 2](#) and conclude with [Theorem 1](#) at the end of this subsection.

**Algorithm Description** First, we compute a non-decreasing degree ordering  $\alpha = (v_1, \dots, v_n)$  and relabel the graph according to  $\alpha$ . Thereafter we check whether  $\alpha$  is a perfect elimination ordering in  $\mathcal{O}(\text{sort}(n+m))$  I/Os by [Proposition 1](#). In the case that  $\alpha$  is not a perfect elimination ordering, the algorithm returns three vertices  $v_j, v_k, v_i$  where  $\{v_i, v_j\}, \{v_i, v_k\} \in E$  but  $\{v_j, v_k\} \notin E$  and  $i < j < k$ , violating that  $v_i$  is simplicial in  $G[\{v_i, \dots, v_n\}]$ , similar to the counterexample provided in [Figure 3b](#). In order to return a forbidden substructure we find additional vertices that complete the induced subgraphs. Note that  $(v_k, v_i, v_j)$  already forms a  $P_3$  and may extend to a  $C_4$  if  $N(v_k) \cap N(v_j)$  contains a vertex  $z \neq v_i$  that is not adjacent to  $v_i$ , see [Figure 4](#) for an illustration.

Computing  $(N(v_k) \cap N(v_j)) \setminus N(v_i)$  requires scanning the adjacencies of three vertices totaling to  $\mathcal{O}(\text{scan}(n))$  I/Os. If  $(N(v_k) \cap N(v_j)) \setminus N(v_i)$  is empty we try to extend the  $P_3$  to a  $C_5$  or output a  $2K_2$  otherwise. To do so, we find vertices  $x \neq v_i$  and  $y \neq v_i$  for which  $\{x, v_j\}, \{y, v_k\} \in E$  but  $\{x, v_k\}, \{y, v_j\} \notin E$  that are also not adjacent to  $v_i$ , i.e.  $\{x, v_i\}, \{y, v_i\} \notin E$ . Both  $x$  and  $y$  exist due to the ordering  $\alpha$  [20] and are found using  $\mathcal{O}(1)$  scanning steps requiring  $\mathcal{O}(\text{scan}(n))$  I/Os. If  $\{x, y\} \in E$  then  $(v_j, v_i, v_k, y, x)$  is a  $C_5$ , otherwise  $G[\{v_j, x, v_k, y\}]$  constitutes a  $2K_2$ . Determining whether  $\{x, y\} \in E$  requires scanning  $N(x)$  and  $N(y)$  using  $\mathcal{O}(\text{scan}(n))$  I/Os, see [Figure 4](#).

In the membership case,  $\alpha$  is a perfect elimination ordering and the algorithm proceeds to verify first the clique  $K$  and then the independent set  $I$  of the split partition  $(K, I)$ . Note that for a split graph the maximum clique of size  $k$  must consist of the  $k$ -highest ranked vertices in  $\alpha$  [20] where  $k$  can be computed using  $\mathcal{O}(\text{sort}(n+m))$  I/Os by [Proposition 2](#). Therefore, it suffices to verify for each of the  $k$  candidates  $v_i$  whether it is connected to  $\{v_{i+1}, \dots, v_n\}$  since the graph is undirected. For a sorted sequence of edges relabeled by  $\alpha$ , we check this property using  $\mathcal{O}(\text{scan}(m))$  I/Os. If we find a vertex  $v_i \in \{v_{n-k+1}, \dots, v_n\}$  where  $\{v_i, v_j\} \notin E$  with  $i < j$  then  $G[\{v_i, \dots, v_n\}]$  already does not constitute a clique and we have to return a NO-certificate. Since the maximum clique has size  $k$ , there are  $k$  vertices with degree at least  $k-1$ . By these degree constraints there must exist an edge  $\{v_i, x\} \in E$  where  $x \in \{v_1, \dots, v_{i-1}\}$  [20]. Additionally, it holds that  $\{x, v_j\} \notin E$  and there exists an edge  $\{z, v_j\} \in E$  where  $z \in \{v_1, \dots, v_{i-1}\}$  that cannot be connected to  $x$ , i.e.  $\{x, z\} \notin E$  [20]. Thus, we first scan the adjacency lists of  $v_i$  and  $v_j$  to find  $x$  and  $z$  in  $\mathcal{O}(\text{scan}(n))$  I/Os and return  $G[\{v_i, v_j, x, z\}]$  as the  $2K_2$  NO-certificate. Otherwise let  $K = \{v_{n-k+1}, \dots, v_n\}$  be the confirmed clique.

Lastly, the algorithm verifies whether the remaining vertices form an independent set. We verify

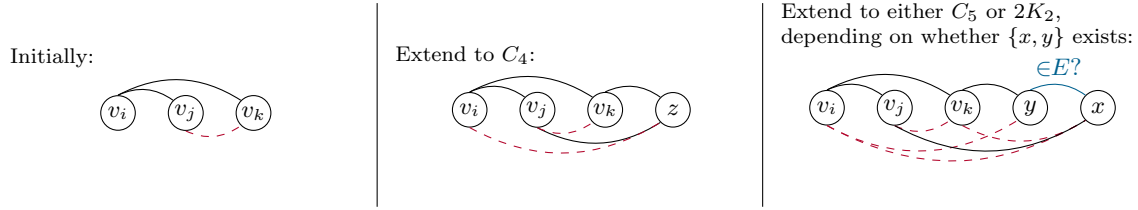


Figure 4: Sequence of operations in case  $\alpha$  is not a perfect elimination ordering. We highlight non-edges  $\{u, v\} \notin E$  by a red dashed line as the certificates are *induced* subgraphs. In the last case, if  $\{x, y\} \in E$  then the whole subgraph is a  $C_5$  and otherwise  $G[\{v_j, x, v_k, y\}]$  is a  $2K_2$ .

that each candidate  $v_i$  is not connected to  $\{v_{i+1}, \dots, v_{n-k}\}$ , since the graph is undirected. For this, it suffices to scan over  $n - k$  consecutive adjacency lists in  $\mathcal{O}(\text{scan}(m))$  I/Os. More precisely, we scan the adjacency lists from  $v_{n-k}$  to  $v_1$  and in case an edge  $\{v_i, v_j\}$  where  $i < j \leq n - k$  is found we find two more vertices to again complete a  $2K_2$ . For the first occurrence of such a vertex  $v_i$ , we remark that  $\{v_{i+1}, \dots, v_{n-k}\}$  and  $\{v_{n-k+1}, \dots, v_n\}$  form an independent set and a clique, respectively. Therefore there exists a vertex  $y \in K$  that is adjacent to  $x$  but not to  $v_i$  [20]. We find  $y$  by scanning  $N(x)$  and  $N(v_i)$  in  $\mathcal{O}(\text{scan}(n))$  I/Os. To complete the  $2K_2$  we similarly find  $z \in N(y) \setminus (N(x) \cup N(v_i))$  in  $\mathcal{O}(\text{scan}(n))$  I/Os which is guaranteed to exist [20].

**Authentication** Given  $G$  and a split partition  $(K, I)$  we can verify in  $\mathcal{O}(\text{sort}(n + m))$  I/Os that  $G$  is indeed a split partition. After relabeling  $G$  by a non-decreasing degree ordering  $\alpha$ , we verify that the relabeled vertices of  $K$  correspond to the  $k$ -highest ranked vertices in  $\alpha$ . By a subsequent scan over the relabeled edges we check whether any edge runs between vertices of  $I$  and that the last  $k$  vertices form a clique.

For a graph  $G$  and any of the forbidden substructures  $2K_2, C_4$  or  $C_5$  we not only return the corresponding vertex subsets but also the edge positions in the adjacency array representation for both edges and non-edges. To do so, we revert the relabeling in  $\mathcal{O}(\text{sort}(n + m))$  I/Os and access all  $\mathcal{O}(1)$  corresponding adjacency lists in  $\mathcal{O}(\text{scan}(n))$  I/Os and return appropriate pointers to the adjacency array representation. For edges that are present in the substructure we directly point to the corresponding entry. Conversely, as non-edges are not present, we instead return pointers to the position the edge would have occupied if it existed using the fact that the individual adjacency lists are sorted. Since all NO-certificates are of constant size, authentication therefore only requires  $\mathcal{O}(1)$  I/Os by direct accesses to memory.

**Proposition 1** *Verifying that a non-decreasing degree ordering  $\alpha = (v_1, \dots, v_n)$  of a graph  $G$  is a perfect elimination ordering takes  $\mathcal{O}(\text{sort}(n + m))$  I/Os.*

**Proof:** We follow the approach of [17, Theorem 4.5] and adapt it to the external memory using time-forward processing, see Algorithm 1.

After relabeling and sorting the edges by  $\alpha$ , we iterate over the vertices in the order given by  $\alpha$ . For a vertex  $v_i$  the set of neighbors  $N(v_i)$  needs to be a clique in order for  $v_i$  to be simplicial. In order to verify this for all vertices, we iterate over  $\alpha$  and at vertex  $v_i$  retrieve  $H(v_i)$  by a continuous scan over  $E$ . Then, let  $u \in H(v_i)$  be the smallest higher ranked neighbor. As  $u \in H(v_i) \subseteq N(v_i)$  is adjacent to  $v_i$ , it has to be verified that it also is adjacent to the remaining neighbors. We verify this property partially for higher ranked neighbors in time-forward fashion. To do so, we insert a message  $\langle u, w \rangle$  into a priority-queue where  $w \in H(v_i) \setminus \{u\}$  to inform  $u$  of every vertex it should be

**Algorithm 1:** Recognizing Perfect Elimination in External Memory

---

**Data:** edges  $E$  of graph  $G$ , non-decreasing degree ordering  $\alpha = (v_1, \dots, v_n)$   
**Output:** bool whether  $\alpha$  is a peo, three invalidating vertices  $\{v_i, v_j, v_k\}$  if not a peo

```

1 Relabel  $G$  according to  $\alpha$ 
2 for  $i = 1, \dots, n$  do
3   Retrieve  $H(v_i)$  from  $E$ 
4   if  $H(v_i) \neq \emptyset$  then
5     Let  $u$  be the smallest successor of  $v_i$  in  $H(v_i)$ 
6     for  $x \in H(v_i) \setminus \{u\}$  do
7       PQ.push( $\langle u, x, v_i \rangle$ )           // inform  $u$  of  $x$  coming from  $v_i$ 
8   while  $\langle v, v_k, v_j \rangle \leftarrow$  PQ.top() where  $v = v_i$  do           // for each message to  $v_i$ 
9     if  $v_k \notin H(v_i)$  then           //  $v_i$  does not fulfill peo property
10      return FALSE,  $\{v_i, v_j, v_k\}$ 
11    PQ.pop()
12 return TRUE

```

---

adjacent to. For any given  $v_i$  it is therefore verified that  $N(v_i)$  is a clique after the processing of all neighbors has finished. Conversely, after sending all required adjacency information, we retrieve for  $v_i$  all messages  $\langle v_i, - \rangle$  directed to  $v_i$  and check that all received vertices are indeed neighbors of  $v_i$  by comparison to the existing adjacencies as seen by the scan over  $E$ .

Relabeling and sorting the edges takes  $\mathcal{O}(\text{sort}(m))$  I/Os. Every vertex  $v_i$  inserts at most all its higher ranked neighbors into the priority-queue totaling up to  $\mathcal{O}(m)$  messages which takes  $\mathcal{O}(\text{sort}(m))$  I/Os in total. Checking that all received vertices are indeed neighbors only requires a concurrent scan over all edges since vertices are handled in ascending order by  $\alpha$ .  $\square$

**Proposition 2** *Computing the size of a maximum clique in a split graph takes  $\mathcal{O}(\text{sort}(n + m))$  I/Os.*

**Proof:** Note that split graphs are both chordal and co-chordal [18]. For chordal graphs, computing the size of a maximum clique in internal memory takes linear time [17, Theorem 4.17] and can be adapted straight-forwardly to an external memory algorithm using  $\mathcal{O}(\text{sort}(m))$  I/Os.

To do so, we simulate the data accesses of the internal memory variant using priority-queues to employ time-forward processing, see Algorithm 2. The algorithm proceeds similar to Algorithm 1 but relays different information forward in time. For a vertex  $v_i$  we instead inform the smallest successor  $u \in H(v_i)$  of the fact that it is in a clique of size  $|H(v_i)|$ , namely the higher ranked neighbors of  $v_i$ . Conversely, at each vertex  $v_i$ , we collect all sent messages and compute the size of the maximum clique that  $v_i$  is a part of and update the global maximum accordingly.  $\square$

By the above description and Proposition 1 and Proposition 2 it follows that split graphs can be recognized certifiably using  $\mathcal{O}(\text{sort}(n + m))$  I/Os which we summarize in Theorem 1.

**Theorem 1** *A graph  $G$  can be recognized whether it is a split graph or not in  $\mathcal{O}(\text{sort}(n + m))$  I/Os. In the membership case the algorithm returns the split partition  $(K, I)$  as the YES-certificate, and otherwise it returns an  $\mathcal{O}(1)$ -size NO-certificate.*



---

**Algorithm 2:** Maximum Clique Size for Chordal Graphs in External Memory

---

**Data:** edges  $E$  of input graph  $G$ , perfect elimination ordering  $\alpha = (v_1, \dots, v_n)$   
**Output:** maximum clique size  $\chi$

```

1 Relabel  $G$  according to  $\alpha$ 
2  $\chi \leftarrow 0$ 
3 for  $i = 1, \dots, n$  do
4   Retrieve  $H(v_i)$  from  $E$  // scan  $E$ 
5   if  $H(v_i) \neq \emptyset$  then
6     Let  $u$  be the smallest successor of  $v_i$  in  $H(v_i)$ 
7     PQ.push( $(u, |H(v_i)| - 1)$ ) //  $v_i$  simplicial  $\Rightarrow G[N(v_i)]$  is clique
8      $S(v_i) \leftarrow -\infty$ 
9     while  $\langle v, S \rangle \leftarrow$  PQ.top() where  $v = v_i$  do
10     $S(v_i) \leftarrow \max\{S(v_i), S\}$  // compute maximum over all
11    PQ.pop()
12   $\chi \leftarrow \max\{\chi, 1 + S(v_i)\}$ 
13 return  $\chi$ 

```

---

### 3.2 Threshold Graphs

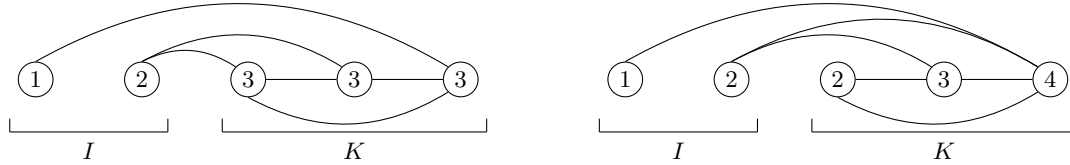
Threshold graphs [11, 17, 25] are split graphs with the additional property that the independent set  $I$  of the split partition  $(K, I)$  has a nested neighborhood ordering. In this context, a subset  $X = \{u_1, \dots, u_k\} \subseteq V$  has a *nested neighborhood ordering* (*nno*) if there exists an ordering  $(u_1, \dots, u_k)$  such that  $(N(u_1) \setminus X) \subseteq (N(u_2) \setminus X) \subseteq \dots \subseteq (N(u_k) \setminus X)$ . Its corresponding forbidden substructures are  $2K_2, P_4$  and  $C_4$ . Alternatively, threshold graphs can be characterized by a graph generation process: repeatedly add universal or isolated vertices to an initially empty graph. Conversely, by repeatedly removing universal and isolated vertices from a threshold graph the resulting graph must be the empty graph.

Intuitively, the relation between the non-decreasing degree ordering  $\alpha$  with the nested neighborhood ordering on the independent set is precisely explained by the graph generation process. When tracking the sorted degree sequence  $d$  of a threshold graph along the graph generation process, it is clear that adding isolated vertices just adds a zero entry at the front of  $d$ . Furthermore, this temporarily isolated vertex will retain its relative order in comparison to the other existing vertices as the degrees of them all will increase simultaneously when adding a universal vertex at the back of  $d$ . The nested neighborhood ordering therefore emerges naturally by the order the isolated vertices are added, see Figure 5b for an illustration of this and Figure 5a for a counterexample.

Our algorithm adapts the internal memory certifying algorithm of Heggernes and Kratsch [20] to external memory by adding an I/O-efficient preprocessing subroutine that differentiates the certification to that of split graphs. As output it either returns a nested neighborhood ordering  $\beta$  of  $I$  as a YES-certificate<sup>3</sup> or one of the forbidden induced subgraphs  $C_4, P_4$  or  $2K_2$  as a NO-certificate. We again present the certifying algorithm and its corresponding authentication algorithm and provide details in Proposition 3 and conclude with Theorem 2 at the end of the subsection.

---

<sup>3</sup>Note that, the ordering  $\beta$  of  $I$  suffices as output since  $K = V \setminus I$ .



(a) Example of a split graph that is not a threshold graph. The vertices in  $I$  do not emit a nested neighborhood ordering. The first three vertices together with the last vertex form a  $P_4$ , a forbidden induced subgraph for threshold graphs.

(b) Example of a threshold graph generated by adding isolated and universal vertices in the following order:  $(\mathcal{I}, \mathcal{I}, \mathcal{U}, \mathcal{I}, \mathcal{U})$  where  $\mathcal{I}$  and  $\mathcal{U}$  indicate the addition of an isolated and universal vertex, respectively.

Figure 5: Illustrations of graphs ordered non-decreasingly by degree from left to right.

**Algorithm Description** First, the algorithm certifies whether the input is a split graph. In the non-membership case, if the returned NO-certificate is a  $C_5$  we extract a  $P_4$  otherwise we return the substructure immediately. For the membership case, we recognize whether the input is a threshold graph by repeatedly removing universal and isolated vertices using the previously computed perfect elimination ordering  $\alpha$  in  $\mathcal{O}(\text{sort}(n + m))$  I/Os by Proposition 3 (see below). If the remaining graph is empty, we return the independent set  $I$  with its non-decreasing degree ordering. Note that after removing a universal vertex  $v_i$ , adjacent vertices with degree one become isolated. Hence, the removal of a high-degree universal vertex that is located at the back of  $\alpha$  has an effect on low-degree vertices that are at the front of  $\alpha$ . These changes cannot be reflected on-the-fly by an I/O-efficient algorithm as these incur unstructured I/Os. Therefore preprocessing is required.

For every vertex  $v_i$  we compute the number of vertices  $S(v_i)$  that become isolated after the removal of  $\{v_i, \dots, v_n\}$ . To do so, we iterate over  $\alpha$  in ascending order and consider vertices  $v_i$  where  $L(v_i) = \emptyset$ . Since  $v_i$  has no lower ranked neighbors, it would become isolated after removing all vertices in  $H(v_i)$ , in particular this happens when the last successor with smallest index  $v_j \in H(v_i)$  is removed. To capture this information, we save  $v_j$  in a vector  $\mathcal{S}$  and sort  $\mathcal{S}$  in non-ascending order incurring  $\mathcal{O}(\text{sort}(m))$  I/Os. The number of consecutive occurrences of any vertex  $v_j$  in  $\mathcal{S}$  correspond to the number of isolated vertices that are created by the removal of the vertices  $\{v_j, \dots, v_n\}$ . Thus, the aforementioned values  $S(v_n), \dots, S(v_1)$  are now accessible by a scan over  $\mathcal{S}$  after counting the occurrences of each  $v_j$  in  $\mathcal{O}(\text{scan}(m))$  I/Os.

The algorithm now proceeds to check whether removing universal and isolated vertices leads to an empty graph. By iterating in reverse order of  $\alpha$ , vertices are considered in non-increasing degree order and verified to be universal using the values that are computed in the preprocessing stage without the need to actually remove them. This incurs a total of  $\mathcal{O}(\text{scan}(n))$  I/Os. In the membership case, the resulting graph would be empty and we return a non-decreasing degree ordering  $\beta$  on the vertices of the independent set  $I$ . In the non-membership case, there must exist a  $P_4$  since the input is a split graph and can therefore not contain a  $C_4$  or a  $2K_2$ .

To find a  $P_4$ , we can disregard further vertices from the remaining graph that cannot be part of a  $P_4$ . For this, let  $K' \subset K$  and  $I' \subset I$  be the remaining vertices when the non-universal vertex is discovered. Any  $v \in K$  where  $N(v) \cap I' = \emptyset$  and any  $v \in I$  where  $N(v) \cap K' = K'$  cannot be part of a  $P_4$  and can therefore be disregarded [20]. We proceed by considering and removing vertices of  $K$  by non-descending degree and vertices of  $I$  by non-ascending degree. After this process, we retrieve the highest-degree vertex  $v$  in  $I$  for which there exists  $\{v, y\} \notin E$  and  $\{y, z\} \in E$  where

**Algorithm 3:** Recognizing Threshold Graphs for Split Graphs in External Memory

---

**Data:** edges  $E$  of split graph  $G$ , perfect elimination ordering  $\alpha = (v_1, \dots, v_n)$   
**Output:** bool whether  $G$  is threshold

```

1 Relabel  $G$  according to  $\alpha$ 
2 Vector  $\mathcal{S}$ 
3 for  $i = 1, \dots, n$  do
4   if  $L(v_i) = \emptyset$  then
5     Let  $v_j$  be the smallest successor of  $v_i$  in  $H(v_i)$ 
6      $\mathcal{S}.push(v_j)$  //  $v_i$  would be isolated after deleting  $\{v_j, \dots, v_n\}$ 
7 Sort  $\mathcal{S}$  in non-ascending order
8  $n_{del} \leftarrow 0$  // number of deleted universal/isolated vertices
9 for  $i = n, \dots, 1$  do
10  if  $L(v_i) \neq \emptyset$  then //  $v_i$  not isolated in  $G[\{v_1, \dots, v_n\}]$ 
11    if  $|L(v_i)| < (n - 1) - n_{del}$  then //  $v_i$  not universal
12      return FALSE
13     $n_{del} \leftarrow n_{del} + 1 + \text{occurrences of } v_i$  //  $v_i$  removed, scan  $\mathcal{S}$ 
14 return TRUE

```

---

$y \in K$  and  $z \in I$  [20]. Additionally, there is a neighbor  $w \in K$  of  $v$  for which  $\{w, z\} \notin E$  [20] and we return the  $P_4$  given by  $G[\{v, w, y, z\}]$ , see Figure 5a for an example. Finding the  $P_4$  at this stage therefore only requires  $\mathcal{O}(\text{scan}(n + m))$  I/Os.

**Authentication** Given  $G$  and a nested neighborhood ordering  $\beta$ , we authenticate that the implicitly given split partition  $(K, I)$  certifies that  $G$  is a split graph using  $\mathcal{O}(\text{sort}(n + m))$  I/Os, as detailed in subsection 3.1. It remains to verify that  $\beta = (v_1, \dots, v_{|I|})$  is indeed a nested neighborhood ordering of  $I$ . To do so, we verify for increasing  $i$  that  $N(v_i) \subseteq N(v_{i+1})$  by a concurrent scan over both neighborhoods requiring a total of  $\mathcal{O}(\text{scan}(m))$  I/Os for all  $i$ .

Since the NO-certificates are again of constant size, authenticating in the non-membership case takes  $\mathcal{O}(1)$  I/Os, as detailed in subsection 3.1.

**Proposition 3** *Verifying that  $G$  emits an empty graph after repeatedly removing universal and isolated vertices requires  $\mathcal{O}(\text{sort}(n + m))$  I/Os.*

**Proof:** The described algorithm can be seen in Algorithm 3. Relabeling of  $G$  by any non-decreasing degree ordering takes  $\mathcal{O}(\text{sort}(n + m))$  I/Os. Generating the values  $S(v_n), \dots, S(v_1)$  requires a scan over all adjacency lists in ascending order and sorting  $\mathcal{S}$  which takes  $\mathcal{O}(\text{scan}(m) + \text{sort}(n))$  I/Os. After preprocessing, the algorithm only requires a reverse scan over the vertices  $v_n, \dots, v_1$ . While iterating over  $\alpha$  in reverse order, we check for each  $v_i$  whether  $L(v_i) = \emptyset$ . If  $v_i$  is not isolated it must be universal. Therefore we compare its current degree  $\deg(v_i)$  with the value  $(n - 1) - n_{del}$  where  $n_{del} = \sum_{j=j+1}^n S(v_j)$  is the number of already removed vertices. All operations take  $\mathcal{O}(\text{scan}(m))$  I/Os in total.  $\square$

By the above description and Proposition 3 it follows that there exists a certifying algorithm for the recognition of threshold graphs using  $\mathcal{O}(\text{sort}(n + m))$  I/Os which is summarized in Theorem 2.

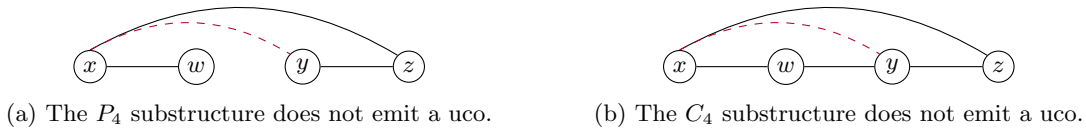


Figure 6: Forbidden induced subgraphs for trivially perfect graphs. We only highlight the critical non-edge that should exist if  $x$  is the earliest vertex in the uco. Note that, (a) only shows one possible layout of a  $P_4$ .

**Theorem 2** *A graph  $G$  can be recognized whether it is a threshold graph or not in  $\mathcal{O}(\text{sort}(n+m))$  I/Os. In the membership case the algorithm returns a nested neighborhood ordering  $\beta$  as the YES-certificate, and otherwise it returns an  $\mathcal{O}(1)$ -size NO-certificate.*

### 3.3 Trivially Perfect Graphs

Trivially perfect graphs have no vertex subset that induces a  $P_4$  or a  $C_4$  [17]. In contrast to split graphs, any non-increasing degree ordering of a trivially perfect graph is a universal-in-a-component ordering [20]. A vertex ordering  $\gamma = (u_1, \dots, u_n)$  is a *universal-in-a-component ordering (uco)* if  $u_i$  is universal in its connected component in  $G[\{u_i, u_{i+1}, \dots, u_n\}]$  for all  $i \in \{1, \dots, n\}$ . In fact, this is a one-to-one correspondence: a non-increasing sorted degree sequence of a graph is a universal-in-a-component ordering if and only if the graph is trivially perfect [20]. To provide further intuition, we reiterate the reason why a  $P_4$  or  $C_4$  cannot exist in a graph that emits a universal-in-a-component ordering  $\gamma$ . Suppose  $\{w, x, y, z\}$  are the vertices of a  $P_4$  or  $C_4$  where  $x$  is wlog. the vertex with the earliest occurrence among the four in  $\gamma$ . Since  $P_4$  and  $C_4$  are connected subgraphs and  $\gamma$  is a uco,  $x$  has to be connected to all remaining vertices, i.e.  $w, y$  and  $z$ . However this contradicts the structure of the  $P_4$  or  $C_4$  as these are *induced* subgraphs, see Figure 6 for an example.

In external memory this can be verified using time-forward processing by adapting the algorithm in [20]. As output it either returns a universal-in-a-component ordering  $\gamma$  as a YES-certificate or one of the two forbidden subgraphs  $C_4, P_4$  as a NO-certificate. We again present the certifying algorithm and its corresponding authentication algorithm and provide details in Proposition 4 and conclude with Theorem 3 at the end of the subsection.

**Algorithm Description** After computing a non-increasing degree ordering  $\gamma$  the algorithm relabels the edges of the graph according to  $\gamma$  and sorts them. Now we iterate over the vertices in ascending order of  $\gamma$ , process for each vertex  $v_i$  its received messages and relay further messages forward in time. Initially all vertices are labeled with 0. Then, at step  $i$  vertex  $v_i$  checks that all adjacent vertices  $N(v_i)$  have the same label as  $v_i$ . After this,  $v_i$  relabels each vertex  $u \in N(v_i)$  with its own index  $i$  and is then removed from the graph.

In the external memory setting we cannot access labels of vertices and relabel them on-the-fly but rather postpone the comparison of the labels to the adjacent vertices instead. To do so,  $v_i$  forwards its own label  $\ell(v_i)$  to  $u \in H(v_i)$  by sending two messages  $\langle u, v_i, \ell(v_i) \rangle$  and  $\langle u, v_i, i \rangle$  to  $u$ , signaling that  $u$  should compare its own label to  $v_i$ 's label  $\ell(v_i)$  and then update it to  $i$ . Since the label of any adjacent vertex is changed after processing a vertex, when arriving at vertex  $v_j$  an odd number of messages will be targeted to  $v_j$ , where the last one corresponds to its actual label at step  $j$ . Then, after collecting all received labels, we compare disjoint consecutive pairs of labels and check whether they match. In the membership case, we do not find any mismatch and return

---

**Algorithm 4:** Recognizing Universal-in-a-Component Orderings in External Memory

---

**Data:** edges  $E$  of graph  $G$ , non-increasing degree ordering  $\gamma = (v_1, \dots, v_n)$   
**Output:** bool whether  $\gamma$  is a uco

```

1 Relabel  $G$  according to  $\gamma$ 
2 for  $i = 1, \dots, n$  do
3   Vector  $\mathcal{L} = [0]$  // initialize with 0
4   while  $\langle v, v_j, \ell \rangle \leftarrow \text{PQ.top}()$  where  $v = v_i$  do //  $v_i$ 's received labels
5      $\mathcal{L}.\text{push}(\ell)$ 
6      $\text{PQ}.\text{pop}()$ 
7   for  $i = 1, \dots, \mathcal{L}.\text{size}/2$  do //  $\mathcal{L}.\text{size}$  is even
8     if  $\mathcal{L}[2i] \neq \mathcal{L}[2i+1]$  and  $\mathcal{L}.\text{size} > 1$  then // mismatch / anomaly
9       return FALSE
10   $\ell(v_i) \leftarrow \mathcal{L}[\mathcal{L}.\text{size}]$  // assign label of  $v_i$ 
11  Retrieve  $H(v_i)$  from  $E$  // scan  $E$ 
12  for  $u \in H(v_i)$  do
13     $\text{PQ}.\text{push}(\langle u, v_i, \ell(v_i) \rangle)$ 
14     $\text{PQ}.\text{push}(\langle u, v_i, i \rangle)$ 
15 return TRUE

```

---

$\gamma$  as the YES-certificate. Otherwise, we have to return a  $P_4$  or  $C_4$ .

In the description of [20] the authors stop at the first anomaly where  $v_i$  detects a mismatch in its own label and one of its neighbors. We simulate the same behavior by writing out every anomaly we find, e.g. that  $v_j$  does not have the expected label of  $v_i$  via an entry  $\langle v_i, v_j, k \rangle$  where  $k$  denotes the label of  $v_j$ . After sorting the entries, we find the earliest anomaly  $\langle v_i, v_j, k \rangle$  with the largest label  $k$  of  $v_i$ 's neighbors in  $\mathcal{O}(\text{sort}(m))$  I/Os. Since  $v_j$  received the label  $k$  from  $v_k$ , but  $v_i$  did not, it is clear that  $v_k$  is not universal in its connected component in  $G[\{v_k, v_{k+1}, \dots, v_n\}]$  and we thus find a  $P_4$  or  $C_4$ . Note that  $(v_k, v_j, v_i)$  already constitutes a  $P_3$  where  $\deg(v_k) \geq \deg(v_j)$ , since  $v_j$  received the label  $k$ . Since  $v_j$  is adjacent to both  $v_k$  and  $v_i$  and  $\deg(v_k) \geq \deg(v_j)$ , there must exist a vertex  $x \in N(v_k)$  where  $\{v_j, x\} \notin E$ . Thus,  $G[\{v_k, v_j, v_i, x\}]$  is a  $P_4$  if  $\{v_i, x\} \notin E$  and a  $C_4$  otherwise. Finding  $x$  and determining whether the forbidden subgraph is a  $P_4$  or a  $C_4$  requires scanning  $\mathcal{O}(1)$  adjacency lists using  $\mathcal{O}(\text{scan}(n))$  I/Os.

**Authentication** Given  $G$  and a universal-in-a-component ordering we run Algorithm 4 using  $\mathcal{O}(\text{sort}(n + m))$  I/Os by Proposition 4. In the case of non-membership, we find the given substructure in  $\mathcal{O}(1)$  I/Os, as detailed in subsection 3.1.

**Proposition 4** *Verifying that a non-increasing degree ordering  $\gamma = (v_1, \dots, v_n)$  of a graph  $G$  with  $n$  vertices and  $m$  edges is a universal-in-a-component ordering requires  $\mathcal{O}(\text{sort}(m))$  I/Os.*

**Proof:** Every vertex  $v_i$  receives exactly two messages per neighbor in  $L(v_i)$  and verifies that all consecutive pairs of labels match. Then, either the label  $i$  is sent to each higher ranked neighbor of  $H(v_i)$  via time-forward processing or it is verified that  $\gamma$  is not a universal-in-a-component ordering. Since at most  $\mathcal{O}(m)$  messages are forwarded, the resulting overall complexity is  $\mathcal{O}(\text{sort}(m))$  I/Os. Correctness follows from [20] since the adapted algorithm performs the same operations but only delays the label comparisons.  $\square$

By the above description and [Proposition 4](#) it follows that there exists a certifying algorithm for the recognition of trivially perfect graphs using  $\mathcal{O}(\text{sort}(n + m))$  I/Os which we summarize in [Theorem 3](#).

**Theorem 3** *A graph  $G$  can be recognized whether it is a trivially perfect graph or not using  $\mathcal{O}(\text{sort}(n + m))$  I/Os. In the membership case the algorithm returns the universal-in-a-component ordering  $\gamma$  as the YES-certificate, and otherwise it returns an  $\mathcal{O}(1)$ -size NO-certificate.*

### 3.4 Bipartite Chain Graphs

Bipartite chain graphs are bipartite graphs where one part of the bipartition has a nested neighborhood ordering [\[37\]](#) similar to threshold graphs. Interestingly, for chain graphs one side of the bipartition exhibits this property if and only if both partitions do [\[37\]](#). Its forbidden induced substructures are  $2K_2$ ,  $C_3$  and  $C_5$ . By definition, bipartite chain graphs are bipartite graphs which therefore requires I/O-efficient bipartiteness testing.

Our algorithm adapts the internal memory certifying algorithm of Heggernes and Kratsch [\[20\]](#) to external memory. As a byproduct, we develop a certifying algorithm to recognize whether an input graph is bipartite or not and use it as a subroutine, see [Lemma 1](#). The algorithm either returns a bipartition  $(U, V \setminus U)$  with two nested neighborhood orderings on  $U$  and  $V \setminus U$  as a YES-certificate or one of the forbidden induced subgraphs  $C_3$ ,  $C_5$  or  $2K_2$  as a NO-certificate. We present the full certifying algorithm first and provide details in [Lemma 1](#), [Corollary 1](#) and conclude with [Theorem 4](#) at the end of the subsection.

**Algorithm Description** We follow the linear time internal memory approach of [\[20\]](#) with slight adjustments to accommodate the external memory setting. First, we check whether the input is indeed a bipartite graph. Instead of using breadth-first search which is very costly in external memory, even for constrained settings [\[7, 29, 2\]](#), we can use a more efficient approach with spanning trees which is presented in [Lemma 1](#). In case the input is not connected, we simply return two edges of two different components as the  $2K_2$ . If the graph is connected, we proceed to verify that the graph is bipartite and return a NO-certificate in the form of a  $C_3$ ,  $C_5$  or  $2K_2$  in case it is not. In order to find a  $C_3$ ,  $C_5$  or  $2K_2$  some modifications to [Lemma 1](#) are necessary. Essentially, the algorithm instead returns a minimum odd cycle that is built from the spanning tree  $T$  and a single non-tree edge. Due to minimality we can then find a  $C_3$ ,  $C_5$  or a  $2K_2$ . The result is summarized in [Corollary 1](#).

Then, it remains to show that each side of the bipartition has a nested neighborhood ordering. Let  $U$  be the larger side of the partition. By [\[25\]](#) it suffices to show that the input is a bipartite chain graph if and only if the graph obtained by adding all possible edges with both endpoints in  $U$  is a threshold graph. Instead of materializing the threshold graph, we implicitly represent the new adjacencies of vertices in  $U$  to retain the same I/O-complexity and apply [Theorem 2](#) using  $\mathcal{O}(\text{sort}(n + m))$  I/Os. Note that, in this threshold graph vertices of  $U$  have higher degrees than vertices in  $V \setminus U$  since  $U$  is the larger side of the bipartition. If the input is bipartite but not bipartite chain, we repeatedly delete vertices that are connected to all other vertices of the other side<sup>4</sup> and the resulting isolated vertices, similar to [subsection 3.2](#) and [\[20\]](#). After this, the vertex  $v$  with highest degree has a non-neighbor  $y$  in the other partition. By similar arguments to [subsection 3.2](#)  $y$  is adjacent to another vertex  $z$  that is adjacent to a vertex  $x$  where  $\{v, x\} \notin E$  [\[20\]](#). As such,  $G[\{v, y, z, x\}]$  is a  $2K_2$  and can be found in  $\mathcal{O}(\text{scan}(n))$  I/Os and returned as the NO-certificate.

<sup>4</sup>These correspond to the universal vertices in the case of threshold graphs but applied to bipartite graphs.

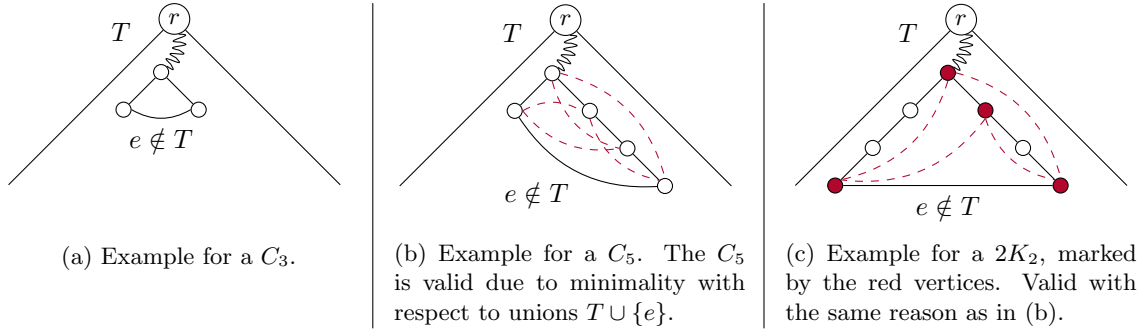


Figure 7: Visualization of the different NO-certificates that are produced in Corollary 1.

**Authentication** Given  $G$  and a bipartition  $(U, V \setminus U)$  with two nested neighborhood orderings on  $U$  and  $V \setminus U$  we first confirm that  $U$  and  $V \setminus U$  are indeed independent sets using  $\mathcal{O}(\text{sort}(n + m))$  I/Os similar to subsection 3.1. After this, we confirm that the provided orderings are nested neighborhood orderings as detailed in subsection 3.2.

As the NO-certificates are of constant size, authentication again only takes  $\mathcal{O}(1)$  I/Os in the non-membership case similar to subsection 3.1.

**Lemma 1** *A graph  $G$  can be recognized whether it is a bipartite graph or not in  $\mathcal{O}(\text{sort}(n + m))$  I/Os, given a spanning forest of the input graph. In the membership case the algorithm returns a bipartition  $(U, V \setminus U)$  as the YES-certificate, and otherwise it returns an odd cycle as the NO-certificate.*

**Proof:** In case there are multiple connected components, we operate on each individually and thus assume that the input is connected. Let  $T$  be the edges of the spanning tree and  $E \setminus T$  the non-tree edges. Any edge  $e \in E \setminus T$  may produce an odd cycle by its addition to  $T$ . In fact, the input is bipartite if and only if  $T \cup \{e\}$  is bipartite for all  $e \in E \setminus T$ <sup>5</sup>. We check whether an edge  $e = \{u, v\}$  closes an odd cycle in  $T$  by computing the distance  $d_T(u, v)$  of its endpoints in  $T$ , see also Figure 7. Since this is required for every non-tree edge  $E \setminus T$ , we resort to batch-processing. Note that  $T$  is a tree and hence after choosing a designated root  $r \in V$  it holds that  $d_T(u, v) = d_T(u, \text{LCA}_T(u, v)) + d_T(v, \text{LCA}_T(u, v))$  where  $\text{LCA}_T(u, v)$  is the lowest common ancestor of  $u$  and  $v$  in  $T$ . Therefore for every edge  $E \setminus T$  we compute its lowest common ancestor in  $T$  using  $\mathcal{O}((1 + m/n) \cdot \text{sort}(n)) = \mathcal{O}(\text{sort}(m))$  I/Os [9].

Additionally, for each vertex  $v \in V$  we compute its depth in  $T$  in  $\mathcal{O}(\text{sort}(m))$  I/Os using Euler Tours [9] and inform each incident edge of this value by a few scanning and sorting steps. Similarly, each edge  $e = \{u, v\}$  is provided of the depth of  $\text{LCA}_T(u, v)$ . Then, after a single scan over  $E \setminus T$  we compute  $d_T(u, v)$  and check if it is even. If any value is even, we return the odd cycle as a NO-certificate or a bipartition in  $T$  as the YES-certificate. Both can be computed using Euler Tours in  $\mathcal{O}(\text{sort}(m))$  I/Os.  $\square$

**Corollary 1** *If a connected graph  $G$  contains a  $C_3, C_5$  or  $2K_2$  then any of these subgraphs can be found in  $\mathcal{O}(\text{sort}(n + m))$  I/Os given a spanning tree of  $G$ .*

<sup>5</sup>Since  $T$  is bipartite, one can think of  $T$  as a representation of a 2-coloring on  $T$ .

**Proof:** We extend the algorithm presented in Lemma 1 to either return the induced cycles  $C_3$  and  $C_5$  or a  $2K_2$ . While iterating over the edges to find an odd cycle we save the smallest one by keeping a copy of the edge  $e \in E \setminus T$  and the length of the minimum odd cycle. In case we find a  $C_3$  or a  $C_5$  we are done and return the NO-certificate immediately, see Figure 7a and Figure 7b. Otherwise for an odd (non-induced) cycle of length  $k$  with  $k = 2\ell + 1 > 5$  we return a  $2K_2$  by finding a matching edge to the non-tree edge  $e \in E \setminus T$  in the cycle.

Let  $C = (u_1, \dots, u_k, u_1)$  be the returned cycle where  $\{u_k, u_1\}$  is the non-tree edge<sup>6</sup>. In this case we return for the  $2K_2$  the graph  $(\{u_\ell, u_{\ell+1}, u_1, u_k\}, \{\{u_1, u_k\}, \{u_\ell, u_{\ell+1}\}\})$ . If  $\ell$  is odd, the non-edges of the  $2K_2$  cannot exist since otherwise any of the following smaller odd cycles  $(u_1, u_2, \dots, u_{\ell+1}, u_k, u_1)$ ,  $(u_1, u_2, \dots, u_\ell, u_1)$ ,  $(u_\ell, u_{\ell+1}, \dots, u_k, u_\ell)$  and  $(u_1, u_{\ell+1}, u_{\ell+2}, \dots, u_k, u_1)$  would be present, contradicting the minimality of  $C$ . For the other case where  $\ell$  is even, a similar argument can be found. The I/O-complexity therefore remains the same, an overall illustration is given in Figure 7.  $\square$

We summarize our findings for bipartite chain graphs in Theorem 4.

**Theorem 4** *A graph  $G$  can be recognized whether it is a bipartite chain graph or not using  $\mathcal{O}(\text{sort}(n+m))$  I/Os with high probability. In the membership case the algorithm returns a bipartition  $(U, V \setminus U)$  and nested neighborhood orderings of both partitions as the YES-certificate, and otherwise it returns a  $\mathcal{O}(1)$ -size NO-certificate.*

**Proof:** Computing a spanning tree  $T$  requires  $\mathcal{O}(\text{sort}(n+m))$  I/Os with high probability by an external memory variant of the Karger, Klein and Tarjan minimum spanning tree algorithm [9, 5]. By Corollary 1 we find a  $C_3, C_5$  or  $2K_2$  if the input is not bipartite or not connected. We proceed by checking the nested neighborhood orderings of both partitions in  $\mathcal{O}(\text{sort}(n+m))$  I/Os using Theorem 2.  $\square$

## 4 Experimental Evaluation

We implemented our external memory certifying algorithms for split and threshold graphs in C++ using the STXXL library [14]. STXXL offers external memory versions of fundamental algorithmic building blocks like scanning, sorting and several data structures. Our benchmarks are built with GNU g++-10.3 and executed on a machine equipped with an AMD EPYC 7302P processor and 64 GB RAM running Ubuntu 20.04 using six 500 GB solid-state disks.

To provide a comparison of our algorithms, we also implemented the internal memory state-of-the-art algorithms by Heggenes and Kratsch [20]. Our internal memory implementations are direct translations of the descriptions provided in [20] making use of standard STL containers especially hash-maps to answer adjacency-queries in constant time. These data structures are infeasible for external-memory algorithms due to the frequently incurred random accesses.

In order to validate the predicted scaling behavior we generate our instances parameterized by  $n$ . For YES-instances of split graphs we generate a split partition  $(K, I)$  with  $|K| = n/10$  and add each possible edge  $\{u, v\}$  with probability  $1/4$  for  $u \in I$  and  $v \in K$ . Analogously, YES-instances of threshold graphs are generated by repeatedly adding either isolated or universal vertices with probability  $9/10$  and  $1/10$ , respectively. We additionally attempt to generate NO-instances by adding  $\mathcal{O}(1)$  many random edges to the YES-instances. In a last step, we randomize the vertex indices to remove any biases emerging from the generation process.

<sup>6</sup>Note that,  $C$  is not an induced cycle unlike  $C_3$  and  $C_5$ .



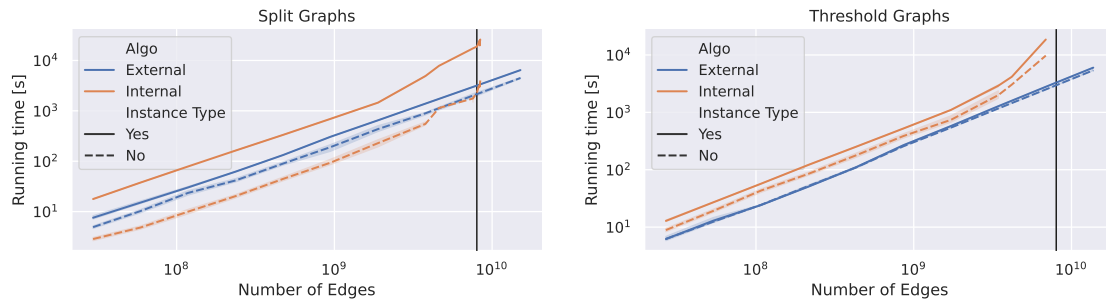


Figure 8: Running times of the certifying algorithms for split (left) and threshold graphs (right) for different random graph instances. The black vertical lines depict the number of elements that can concurrently be held in internal memory.

In Figure 8 we present the running times of all algorithms on multiple YES- and NO-instances. It is clear that the performance of both external memory algorithms is not impacted by the main memory barrier while the running time of their internal memory counterparts already increases when at least half the main memory is used. This effect is amplified immensely after exceeding the size of main memory for split graphs.

Certifying the produced NO-instances of split graphs seems to require less time than their corresponding unmodified YES-instances as the algorithm typically stops early. Furthermore, due to the low data locality of the internal memory variant it is apparent that the external memory algorithm is superior for the YES-instances. The performance on both YES- and NO-instances is very similar in external memory. This is in part due to the fact that the common relabeling step is already relatively costly. For threshold graphs, however, the external memory variant outperforms the internal memory variant due to improved data locality.

## 5 Conclusions

We have presented the first I/O-efficient certifying recognition algorithms for split, threshold, trivially perfect, bipartite and bipartite chain graphs. Our algorithms require  $\mathcal{O}(\text{sort}(n + m))$  I/Os matching common lower bounds for many algorithms in external memory. In our experiments we show that the algorithms perform well even for graphs exceeding the size of main memory.

Further, it would be interesting to extend the scope of certifying recognition algorithms to more graph classes for the external memory regime. In internal memory, a plethora of graph classes are efficiently certifiable which currently have no efficient external memory pendant, e.g. circular-arc graphs [15], HHD-free graphs [34], interval graphs [24], normal helly circular-arc graphs [8], permutation graphs [24], proper interval graphs [21], proper interval bigraphs [21] and many more. Due to limited data locality, straight-forward applications of these algorithms are highly inefficient for use in external memory. In turn, new algorithmic techniques are necessary to bridge the gap to larger processing scales.

## References

- [1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.
- [2] D. Ajwani and U. Meyer. Design and engineering of external memory traversal algorithms for general graphs. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation [DFG priority program 1126]*, volume 5515 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2009. doi:10.1007/978-3-642-02094-0\_1.
- [3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/s00453-003-1021-x.
- [4] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *J. Graph Algorithms Appl.*, 7(2):105–129, 2003. URL: <https://doi.org/10.7155/jgaa.00063>, doi:10.7155/JGAA.00063.
- [5] G. S. Brodal, R. Fagerberg, D. Hammer, U. Meyer, M. Penschuck, and H. Tran. An experimental study of external memory algorithms for connected components. In D. Coudert and E. Natale, editors, *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*, volume 190 of *LIPICs*, pages 23:1–23:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.SEA.2021.23.
- [6] D. Bruce, C. T. Hoàng, and J. Sawada. A certifying algorithm for 3-colorability of  $P_5$ -free graphs. In Y. Dong, D. Du, and O. H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 594–604. Springer, 2009. doi:10.1007/978-3-642-10631-6\_61.
- [7] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In D. B. Shmoys, editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, pages 859–860. ACM/SIAM, 2000.
- [8] Y. Cao. Direct and certifying recognition of normal helly circular-arc graphs in linear time. In J. Chen, J. E. Hopcroft, and J. Wang, editors, *Frontiers in Algorithmics - 8th International Workshop, FAW 2014, Zhangjiajie, China, June 28-30, 2014. Proceedings*, volume 8497 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2014. doi:10.1007/978-3-319-08016-1\_2.
- [9] Y. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In K. L. Clarkson, editor, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*, pages 139–149. ACM/SIAM, 1995. URL: <http://dl.acm.org/citation.cfm?id=313651.313681>.
- [10] F. Y. L. Chin, H. Ting, Y. H. Tsin, and Y. Zhang. A linear-time certifying algorithm for recognizing generalized series-parallel graphs. *Discret. Appl. Math.*, 325:152–171, 2023. URL: <https://doi.org/10.1016/j.dam.2022.10.005>, doi:10.1016/J.DAM.2022.10.005.

- [11] V. Chvátal and P. L. Hammer. Set-packing and threshold graphs. *Research Report, Computer Science Department, Univeristy Waterloo, 1973*, 1973.
- [12] D. G. Corneil, B. Dalton, and M. Habib. Ldfs-based certifying algorithm for the minimum path cover problem on cocomparability graphs. *SIAM J. Comput.*, 42(3):792–807, 2013. doi:10.1137/11083856X.
- [13] D. G. Corneil and M. Habib. Unified view of graph searching and ldfs-based certifying algorithms. In *Encyclopedia of Algorithms*, pages 2291–2297. 2016. doi:10.1007/978-1-4939-2864-4\_685.
- [14] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software - Practice and Experience*, 38(6):589–637, 2008. doi:10.1002/spe.844.
- [15] M. C. Francis, P. Hell, and J. Stacho. Forbidden structure characterization of circular-arc graphs and a certifying recognition algorithm. In P. Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1708–1727. SIAM, 2015. doi:10.1137/1.9781611973730.114.
- [16] D. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965.
- [17] M. C. Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, 2004.
- [18] P. L. Hammer and S. Földes. Split graphs. *Congressus Numerantium*, 19:311–315, 1977.
- [19] P. L. Hammer and B. Simeone. The splittance of a graph. *Combinatorica*, 1:275–284, 1981.
- [20] P. Heggernes and D. Kratsch. Linear-time certifying recognition algorithms and forbidden induced subgraphs. *Nordic Journal of Computing*, 14(1-2):87–108, 2007.
- [21] P. Hell and J. Huang. Certifying lexbfs recognition algorithms for proper interval graphs and proper interval bigraphs. *SIAM Journal on Discrete Mathematics*, 18(3):554–570, 2004. doi:10.1137/S0895480103430259.
- [22] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974. doi:10.1145/321850.321852.
- [23] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 158–167. ACM/SIAM, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644137>.
- [24] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *SIAM Journal on Computing*, 36(2):326–353, 2006. doi:10.1137/S0097539703437855.
- [25] N. V. Mahadev and U. N. Peled. Threshold graphs and related topics. *Annals of Discrete Mathematics*, 56, 1995.

- [26] A. Maheshwari and N. Zeh. A Survey of Techniques for Designing I/O-Efficient Algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*, pages 36–61. Springer, 2002. doi:10.1007/3-540-36574-5\_3.
- [27] R. M. McConnell. A certifying algorithm for the consecutive-ones property. In J. I. Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 768–777. SIAM, 2004. URL: <http://dl.acm.org/citation.cfm?id=982792.982909>.
- [28] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011. doi:10.1016/j.cosrev.2010.09.009.
- [29] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In R. H. Möhring and R. Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer, 2002. doi:10.1007/3-540-45749-6\_63.
- [30] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. URL: <http://www.mpi-sb.mpg.de/%7Emehlhorn/LEDAbook.html>.
- [31] K. Mehlhorn, A. Neumann, and J. M. Schmidt. Certifying 3-edge-connectivity. *Algorithmica*, 77(2):309–335, 2017. URL: <https://doi.org/10.1007/s00453-015-0075-x>, doi:10.1007/S00453-015-0075-X.
- [32] U. Meyer. External memory BFS on undirected graphs with bounded degree. In S. R. Kosaraju, editor, *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*, pages 87–88. ACM/SIAM, 2001. URL: <http://dl.acm.org/citation.cfm?id=365411.365422>.
- [33] U. Meyer, H. Tran, and K. Tsakalidis. Certifying induced subgraphs in large graphs. In C. Lin, B. M. T. Lin, and G. Liotta, editors, *WALCOM: Algorithms and Computation - 17th International Conference and Workshops, WALCOM 2023, Hsinchu, Taiwan, March 22-24, 2023, Proceedings*, volume 13973 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2023. doi:10.1007/978-3-031-27051-2\_20.
- [34] S. D. Nikolopoulos and L. Palios. An  $o(nm)$ -time certifying algorithm for recognizing HDD-free graphs. *Theoretical Computer Science*, 452:117–131, 2012. doi:10.1016/j.tcs.2012.06.006.
- [35] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. doi:10.1007/978-3-030-25209-0.
- [36] Y. H. Tsin. A simple certifying algorithm for 3-edge-connectivity. *Theor. Comput. Sci.*, 951:113760, 2023. URL: <https://doi.org/10.1016/j.tcs.2023.113760>, doi:10.1016/J.TCS.2023.113760.
- [37] M. Yannakakis. Node-deletion problems on bipartite graphs. *SIAM Journal on Computing*, 10(2):310–327, 1981. doi:10.1137/0210022.