Certifying Induced Subgraphs in Large Graphs

Ulrich Meyer[1], Hung Tran[1], and Konstantinos Tsakalidis[2]

[1] Goethe University Frankfurt, Germany
{umeyer,htran}@ae.cs.uni-frankfurt.de
[2] University of Liverpool, United Kingdom
K.Tsakalidis@liverpool.ac.uk

Abstract. We introduce I/O-effiient certifying algorithms for bipartite graphs, as well as for the classes of 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 $O(\textsc{sort}(n+m))$ 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 for split and threshold graphs and provide a preliminary experimental evaluation.

Keywords: certifying algorithm · graph algorithm · external memory

1 Introduction

Certifying algorithms [13] 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 induced *odd-length cycle* subgraph, i.e. a cycle of vertices with an odd number of edges (`NO`-certificate).

Emerging big data applications need to process large graphs efficiently. Standard models of computation in internal memory (RAM, pointer machine) do not capture the algorithmic complexity of processing graphs with size that exceed the main memory. The *I/O model* by Aggarwal and Vitter [1] is suitable for studying large graphs 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. *I/O-optimal* external memory algorithms for sorting n elements take $\textsc{sort}(n) = O((n/B)\log_{M/B}(n/B))$ I/Os and reading or writing n contiguous items (which is referred to as scanning) requires $\textsc{scan}(n) = O(n/B)$ I/Os.

1.1 Previous Work

Certifying bipartiteness in internal memory takes linear time in the number of edges by any traversal of the graph. However, all known external memory breadth-first search [2] and depth-first search [4] traversal algorithms take suboptimal $\omega\left(\text{SORT}\left(n+m\right)\right)$ I/Os for an input graph with n vertices and m edges.

Heggernes and Kratsch [10] present optimal internal memory algorithms for certifying whether a graph belongs to the classes of split, threshold, bipartite chain, and trivially perfect graphs. 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 *bipartite*, *split*, *threshold*, *bipartite chain*, and *trivially perfect* graphs. All algorithms return in the membership case, a `YES`-certificate w characterizing the graph class, or a $O(1)$-size `NO`-certificate in the non-membership case. All `YES`-certificates can be authenticated using $O(\text{SORT}(n+m))$ I/Os as detailed in the full version of the paper [14]. 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 V and edges E, respectively. 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* $\deg(v)$ of a vertex v is given by $\deg(v)=|N(v)|$. A vertex v is called *simplicial* if $N(v)$ is a clique and *universal* if $N[v]=V$.

Graph 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: $(\{a,b,c,d\},\{ab,cd\})$.

Henceforth we refer to different types of orderings of vertices: an ordering $(v_1,\ldots,v_n)$ is a (i) *perfect elimination ordering* (*peo*) if v_i is simplicial in $G[\{v_i,v_{i+1},\ldots,v_n\}]$ for all $i\in\{1,\ldots,n\}$, and a (ii) *universal-in-a-component-ordering* (*uco*) if v_i is universal in its connected component in $G[\{v_i,v_{i+1},\ldots,v_n\}]$ for all $i\in\{1,\ldots,n\}$. For a subset $X=\{v_1,\ldots,v_k\}$, we call $(v_1,\ldots,v_k)$ a *nested neighborhood ordering* (*nno*) if $(N(v_1)\setminus X)\subseteq(N(v_2)\setminus X))\subseteq\ldots\subseteq(N(v_k)\setminus X)$.

Finally for any ordering, we partition $N(v_i)$ into lower and higher ranked neighbors, respectively, $L(v_i) = \{x \in N(v_i) : v_i \text{ is ranked higher than } x\}$ and $H(v_i) = \{x \in N(v_i) : v_i \text{ is ranked lower than } x\}$.

Graph Representation We assume an *adjacency array representation* [15] where the graph $G = (V, E)$ is represented by two arrays $P = [\ P_i\ ]_{i=1}^{n}$ and $E = [\ u_i\ ]_{i=1}^{m}$. The neighbors of a vertex v_i are then given by the vertices at position $P[v_i]$ to $P[v_{i+1}] - 1$ in E. We use the adjacency array representation to straightforwardly allow for efficient scanning of G: (i) scanning k consecutive adjacency lists consisting of m edges requires $O(\text{SCAN}(m))$ I/Os and (ii) computing and scanning the degrees of k consecutive vertices requires $O(\text{SCAN}(k))$ I/Os.

Time-Forward Processing *Time-forward processing* (*TFP*) is a generic technique to manage data dependencies of external memory algorithms [12]. These dependencies are typically modeled by a directed acyclic graph $G = (V, E)$ where every vertex $v_i \in V$ models the computation of 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 data structure, items addressed to v_j are currently the smallest elements in the data structures and thus can be dequeued with a delete-minimum operation. By using suitable external memory priority-queues [3], TFP incurs $O(\text{SORT}(k))$ I/Os, where k is the number of messages sent.

3 Certifying Graph Classes in External Memory

3.1 Certifying Split Graphs in External Memory

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 structures [9]. Since split graphs are a subclass of chordal graphs, there exists a peo of the vertices for every split graph. In fact, any non-decreasing degree ordering of a split graph is a peo [10].

Our algorithm adapts the internal memory certifying algorithm of Heggernes and Kratsch [10] to external memory by adopting TFP. As output it either returns the split partition (K, I) as a `YES`-certificate or one of the forbidden subgraphs C_4, C_5 or $2K_2$ as a `NO`-certificate. We present the algorithm as a whole and refer to details in Proposition 1 and Proposition 2 at the end of the subsection.

First, we compute a non-decreasing degree ordering $\alpha = (v_1, \dots, v_n)$ and relabel[3] the graph according to α. Thereafter we check whether α is a peo in $O(\text{SORT}(n+m))$ I/Os by Proposition 1. In the non-membership case, 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 $G[\{v_i, \dots, v_n\}]$. In order to return a forbidden subgraph 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. Computing $(N(v_k) \cap N(v_j)) \setminus N(v_i)$ requires scanning the adjacencies of $O(1)$ many vertices totaling to $O(\text{SCAN}(n))$ 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 α [10] and are found using $O(1)$ scanning steps requiring $O(\text{SCAN}(n)$ 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 $O(\text{SCAN}(n))$ I/Os.

In the membership case, α is a peo 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 α [10] where k can be computed using $O(\text{SORT}(m))$ 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 α, we check this property using $O(\text{SCAN}(m))$ 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 $x \in \{v_1, \dots, v_{i-1}\}$ [10]. 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. $\{x, z\} \notin E$ [10]. Thus, we first scan the adjacency lists of v_i and v_j to find x and z in $O(\text{SCAN}(n))$ I/Os and return $G[\{v_i, v_j, x, z\}]$ as the $2K_2$ `NO`-certificate. Otherwise let $K = \{v_{n-k+1}, \dots, v_n\}$.

Lastly, the algorithm verifies whether the remaining vertices form an independent set. We verify 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 $O(\text{SCAN}(m))$ 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 ver-

[3] If a vertex v_i has rank k in α it will be relabeled to v_k. The relabeling results in an adjacency array representation of the relabeled graph requiring $O(\text{SORT}(n+m))$ I/Os.

Algorithm 1: Recognizing Perfect Elimination in EM

```
Data: edges $E$ of graph $G$, non-decreasing degree ordering $\alpha = (v_1, \ldots, v_n)$
Output: bool whether $\alpha$ is a peo, three invalidating vertices $\{v_i, v_j, v_k\}$ if not
1  Sort $E$ and relabel according to $\alpha$
2  for $i = 1, \ldots, n$ 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
```

tex $y \in K$ that is adjacent to x but not to v_i [10]. We find y by scanning $N(x)$ and $N(v_i)$ in $O(\text{SCAN}(n))$ I/Os. To complete the $2K_2$ we similarly find $z \in N(y) \setminus (N(x) \cup N(y_i))$ in $O(\text{SCAN}(n))$ I/Os which is guaranteed to exist [10].

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

Proof. We follow the approach of [8, Theorem 4.5] and adapt it to the external memory using TFP, see Algorithm 1.

After relabeling and sorting the edges by α we iterate over the vertices in the order given by α. For a vertex v_i the set of neighbors $N(v_i)$ needs to be a clique. In order to verify this for all vertices, for a vertex v_i we first retrieve $H(v_i)$. Then let $u \in H(v_i)$ be the smallest ranked neighbor according to α. In order for v_i to be simplicial, u needs to be adjacent to all vertices of $H(v_i) \setminus \{u\}$. In TFP-fashion 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 adjacent to. Conversely, after sending all 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.

Relabeling and sorting the edges takes $O(\text{SORT}(m))$ I/Os. Every vertex v_i inserts at most all its neighbors into the priority-queue totaling up to $O(m)$ messages which requires $O(\text{SORT}(m))$ I/Os. Checking that all received vertices are neighbors only requires a scan over all edges since vertices are handled in non-descending order by α. □

Proposition 2. *Computing the size of a maximum clique in a split graph requires $O(\text{SORT}(m))$ I/Os.*

Proof. Note that split graphs are both chordal and co-chordal [9]. For chordal graphs, computing the size of a maximum clique in internal memory takes linear

Algorithm 2: Maximum Clique Size for Chordal Graphs in EM

```
   Data: edges $E$ of input graph $G$, peo $\alpha = (v_1, \ldots, v_n)$
   Output: maximum clique size $\chi$
 1 Sort $E$ and relabel according to $\alpha$
 2 $\chi \leftarrow 0$
 3 for $i = 1, \ldots, n$ 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($\langle u, |H(v_i)| - 1 \rangle$)   // $v_i$ 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     $\chi \leftarrow \max\{\chi, S(v_i)\}$
13 return $\chi$
```

time [8, Theorem 4.17] and is easily convertible to an external memory algorithm using $O(\text{SORT}(m))$ I/Os. To do so, we simulate the data accesses of the internal memory variant using priority-queues to employ TFP, see Algorithm 2. Instead of updating each $S(v_i)$ value immediately, we delay its consecutive computation by sending a message $\langle v_i, S \rangle$ to v_i to inform v_i, that v_i is part of a clique of size S. After collecting all messages, the overall maximum is computed and the global value of the currently maximum clique is updated if necessary. □

By the above description it follows that split graphs can be certified using $O(\text{SORT}(n + m))$ I/Os which we summarize in Theorem 1.

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

3.2 Certifying Threshold Graphs in External Memory

Threshold graphs [6,8,11] are split graphs with the additional property that the independent set I of the split partition (K, I) has an nno. Its corresponding forbidden subgraphs 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. In comparison to certifying split graphs, threshold graphs thus require additional steps.

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 subgraph 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 peo α in $O(\text{SORT}(m))$ 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, vertices with degree one become isolated. Since low-degree vertices are at the front of α, an I/O-efficient algorithm cannot determine them on-the-fly after removing a high-degree vertex. Therefore pre-processing 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 α in non-descending order and check for v_i with $L(v_i) = \emptyset$. Since v_i has no lower ranked neighbors, it would become isolated after removing all vertices in $H(v_i)$, in particular when the successor with smallest index $v_j \in H(v_i)$ is removed. We save v_j in a vector S and sort S in non-ascending order. The values $S(v_n), \dots, S(v_1)$ are now accessible by a scan over S to count the occurrences of each v_j in $O(\text{SCAN}(m))$ I/Os.

In the non-membership case, there must exist a P_4 since the input is split and cannot contain a C_4 or a $2K_2$. We can delete 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 of the split partition. 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 [10] and can therefore be deleted. 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 $y \in K$ and $z \in I$ [10]. Additionally, there is a neighbor $w \in K$ of v for which $\{w, z\} \notin E$ [10] and we return the P_4 given by $G[\{v, w, y, z\}]$. Finding the P_4 therefore only requires $O(\text{SCAN}(n + m))$ I/Os.

Proposition 3. *Verifying that a non-decreasing degree ordering $\alpha = (v_1, \dots, v_n)$ of a graph G emits an empty graph after repeatedly removing universal and isolated vertices requires $O(\text{SORT}(n) + \text{SCAN}(m))$ I/Os.*

Proof. Generating the values $S(v_n), \dots, S(v_1)$ requires a scan over all adjacency lists in non-descending order and sorting S which takes $O(\text{SORT}(n) + \text{SCAN}(m))$ I/Os. Afte pre-processing, the algorithm only requires a reverse scan over the degrees $d_n, \dots, d_1$. We iterate over α in reverse order, where for each v_i we check 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 $(n-1) - n_{\text{del}}$ where $n_{\text{del}} = \sum_{j=j+1}^{n} S(v_j)$. All operations take $O(\text{SCAN}(m))$ I/Os in total. □

We summarize our findings for threshold graphs in Theorem 2.

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

Proof. Certifying that the input graph is a split graph requires $O(\text{SORT}(n+m))$ I/Os by Theorem 1. If it is, we check if the input is a threshold graph directly by checking whether the graph is empty after repeatedly removing universal and isolated vertices in $O(\text{SORT}(m))$ I/Os by Proposition 3. Otherwise we have to find a P_4, since the input is a split but not a threshold graph. Hence, this step requires $O(\text{SCAN}(n+m))$ I/Os and the total I/Os are $O(\text{SORT}(n+m))$. □

3.3 Certifying Trivially Perfect Graphs in External Memory

Trivially perfect graphs have no vertex subset that induces a P_4 or a C_4 [8]. In contrast to split graphs, any non-increasing degree ordering of a trivially perfect graph is a uco [10]. In fact, this is a one-to-one correspondence: a non-increasing sorted degree sequence of a graph is a uco iff the graph is trivially perfect [10].

In external memory this can be verified using TFP by adapting the algorithm in [10]. After computing a non-increasing degree ordering γ the algorithm relabels the edges of the graph according to γ and sorts them. Now we iterate over the vertices in non-descending order of γ, 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 γ as the `YES`-certificate. Otherwise, we have to return a P_4 or C_4.

In the description of [10] 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. 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 will return a P_4 or C_4. Note that (v_k, v_j, v_i) already constitutes a P_3 where $\deg(v_k) \geq \deg(v_j)$, because 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 $O(1)$ adjacency lists in $O(\text{SCAN}(n))$ I/Os.

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 $O(\textsc{sort}(m))$ 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 TFP or it is verified that γ is not a uco. Since at most $O(m)$ messages are inserted, the resulting overall complexity is $O(\textsc{sort}(m))$ I/Os. Correctness follows from [10] since the adapted algorithm performs the same operations but only delays the label comparisons. □

We again summarize our results in Theorem 3.

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

3.4 Certifying Bipartite Chain Graphs in External Memory

Bipartite chain graphs are bipartite graphs where one part of the bipartition has an nno [16] similar to threshold graphs. Its forbidden induced subgraphs are $2K_2, C_3$ and C_5. By definition, bipartite chain graphs are bipartite graphs which therefore requires I/O-efficient bipartiteness testing.

We follow the linear time internal memory approach of [10] 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 [2], we can use a more efficient approach with spanning trees which is presented in the following in Lemma 1. Note that, computing a spanning forest only requires $O(\textsc{sort}(n + m))$ I/Os with high probability [5] and is therefore no real restriction to 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 T and a single non-tree edge. Due to minimality we can then find a $2K_2$. The result is summarized in Corollary 1 and proven in the full version of the paper [14].

Then, it remains to show that each side of the bipartition has an nno. Let U be the larger side of the partition. By [11] it suffices to show that the input is a 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 mentioned threshold graph, we implicitly represent the adjacencies of vertices in U to retain the same I/O-complexity and apply Theorem 2 using $O(\textsc{sort}(n + m))$ I/Os. If the input is bipartite but not chain, we repeatedly delete vertices that are connected to all other vertices of the other side and the resulting isolated vertices, similar to Subsection 3.3 and [10]. After this, the vertex v with highest degree has a non-neighbor y in the other partition. By similar arguments y is adjacent to another vertex z that is adjacent to a vertex x where $\{v, x\} \notin E$ [10]. As such $G[\{v, y, z, x\}]$ is a $2K_2$ and can be found in $O(\text{SCAN}(n))$ I/Os and returned as the `NO`-certificate.

Lemma 1. *A graph G can be certified whether it is a bipartite graph or not in $O(\text{SORT}(n+m))$ 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-length 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 $e \in E \setminus T$[4]. 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. 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 $O((m/n) \cdot \text{SORT}(n)) = O(\text{SORT}(m))$ I/Os [5].

Additionally, for each vertex $v \in V$ we compute its depth in T in $O(\text{SORT}(m))$ I/Os using Euler Tours [5] 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 $O(\text{SORT}(m))$ I/Os. □

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 $O(\text{SORT}(n+m))$ I/Os given a spanning tree of G.*

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

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

Proof. Computing a spanning tree T requires $O(\text{SORT}(n+m))$ I/Os with high probability by an external memory variant of the Karger, Klein and Tarjan minimum spanning tree algorithm [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 nno's of both partitions in $O(\text{SORT}(n+m))$ I/Os using Theorem 2. □

[4] Since T is bipartite, one can think of T as a representation of a 2-coloring on T.

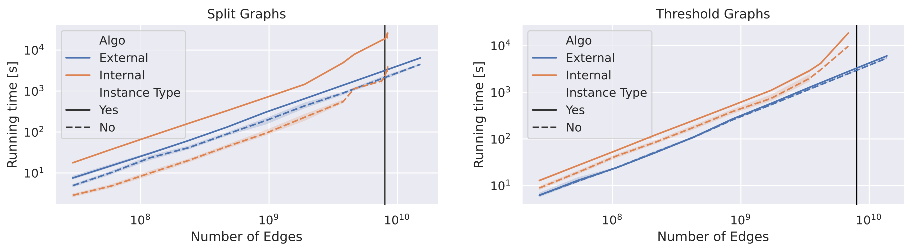

Fig. 1: Running times of the certifying algorithms for split (left) and threshold graphs (right) for different random graph instances. The black vertical lines depict the number of elements that can concurrently be held in internal memory.

4 Experimental Evaluation

We implemented our external memory certifying algorithms for split and threshold graphs in `C++` using the STXXL library [7]. To provide a comparison of our algorithms, we also implemented the internal memory state-of-the-art algorithms by Heggernes and Kratsch [10]. 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.

In order to validate the predicted scaling behaviour 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 $O(1)$ many random edges to the YES-instances. In a last step, we randomize the vertex indices to remove any biases of the generation process.

In Figure 1 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, Figure 1.

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 $O(\text{SORT}(n+m))$ 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.

Acknowledgements This work is partially supported by the International Exchanges Grant IES\R3\203041 of the Royal Society and by the Deutsche Forschungsgemeinschaft (DFG) under grant ME 2088/5-1 (FOR 2975 | Algorithms, Dynamics, and Information Flow in Networks).

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
2. Ajwani, D., Meyer, U.: Design and engineering of external memory traversal algorithms for general graphs. In: Algorithmics of Large and Complex Networks. Lecture Notes in Computer Science, vol. 5515, pp. 1–33. Springer (2009)
3. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica **37**(1), 1–24 (2003)
4. Buchsbaum, A.L., Goldwasser, M.H., Venkatasubramanian, S., Westbrook, J.R.: On external memory graph traversal. In: SODA. pp. 859–860. ACM/SIAM (2000)
5. Chiang, Y., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: SODA. pp. 139–149. ACM/SIAM (1995)
6. Chvátal, V.: Set-packing and threshold graphs. Res. Rep., Comput. Sci. Dept., Univ. Waterloo, 1973 (1973)
7. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw. Pract. Exp. **38**(6), 589–637 (2008)
8. Golumbic, M.C.: Algorithmic graph theory and perfect graphs. Elsevier (2004)
9. Hammer, P.L., Földes, S.: Split graphs. Congressus Numerantium **19**, 311–315 (1977)
10. Heggernes, P., Kratsch, D.: Linear-time certifying recognition algorithms and forbidden induced subgraphs. Nord. J. Comput. **14**(1-2), 87–108 (2007)
11. Mahadev, N.V., Peled, U.N.: Threshold graphs and related topics. Elsevier (1995)
12. Maheshwari, A., Zeh, N.: A survey of techniques for designing I/O-efficient algorithms. In: Algorithms for Memory Hierarchies. Lecture Notes in Computer Science, vol. 2625, pp. 36–61. Springer (2002)
13. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
14. Meyer, U., Tran, H., Tsakalidis, K.: Certifying induced subgraphs in large graphs. CoRR **abs/2210.13057** (2022)
15. Sanders, P., Mehlhorn, K., Dietzfelbinger, M., Dementiev, R.: Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox. Springer (2019)
16. Yannakakis, M.: Node-deletion problems on bipartite graphs. SIAM J. Comput. **10**(2), 310–327 (1981)