Maximum Independent Set for Interval Graphs and Trees in Space Efficient Models

Binay K. Bhattacharya* Minati De† Subhas C. Nandy‡ Sasanka Roy§

Abstract

Space efficient algorithms for the maximum independent set problem for interval graphs and trees are presented in this paper. For a given set of n intervals on a real line, we can compute the maximum independent set in $O(\frac{n^2}{s} + n \log s)$ time using $O(s)$ extra-space. The lower bound of the *time* $\times$ *space* product for this problem is $\Omega(n^{2-\epsilon})$, where $\epsilon = O(1/\sqrt{\log n})$. We also propose an $(1+\frac{1}{\epsilon})$ approximation algorithm for this problem that executes $O(\frac{1}{\epsilon})$ passes over the read-only input tape and uses $O(k)$ extra space, where k is the size of the reported answers. For trees with n nodes, our proposed $\frac{5}{3}$-approximation algorithm runs in $O(n)$ time using $O(1)$ extra-space.

1 Introduction

Computing the maximum independent set has been a problem of fundamental interest to different community of researchers. Independent set for a graph $G = (V, E)$ is defined as a set of vertices $I(\subseteq V)$ such that no two vertices of I are connected by an edge. A maximum independent set, called MIS, is an independent set of maximum size. In this paper, we study the problem of computing the maximum independent set for the interval graph and tree in space-economic models. We consider both the read-only model with random access and the sequential access of the input.

Elberfeld et al. [5] in their seminal work provided a logspace version of the theorems of Bodlaender [2] and Courcelle [4]. Bodlaender's theorem [2] in conjunction with Courcelle's [4] theorem provides a generic framework to find linear time algorithms for many problems in bounded tree-width graphs which also includes computing the MIS for bounded tree-width graphs. Since Elberfeld et al. [5] gives a logspace version of both Bodlaender [2] and Courcelle [4] theorems, this would imply a polynomial time algorithm for computing the MIS of bounded tree-width graphs using $O(1)$ words (i.e, $O(\log n)$ bits). But we could not analyze the correct running time of Elberfeld et al. [5] for finding the MIS, even for trees.

For computing the MIS of a given set of intervals, Emek et al. [6] proposed a multi-pass algorithm which reports $(1 + \frac{1}{2p-1})$ approximate result in p passes using $O(p.k)$ space, where k is the number of intervals reported. Here, we improve the space complexity. We propose a $(1 + \frac{2}{p})$ approximation algorithm that executes $(2p+1)$ passes on the input and takes $O(k)$ space, where k is the number of reported intervals. We also show the time-space product lower bound for this problem matches the problem of element uniqueness problem which is $\Omega(n^{2-\epsilon})$, where $\epsilon = O(1/\sqrt{\log n})$ [9]. In the random-access read-only model, we show a near optimal algorithm whose time-space product is $O(n^2)$.

For tree, an $O(n)$ time exact algorithm using $O(h)$ extra-space is easy to obtain, where h is the height of the tree. We describe a $\frac{5}{3}$-approximation algorithm for tree which takes $O(n)$ time and uses only $O(1)$ space.

2 Maximum independent set for intervals

A set of n intervals $\mathcal{I}$ is given in a read-only array $I[1, 2, \ldots, n]$. For any interval $I[t] \in \mathcal{I}$, we denote it's left end point by $left(I[t])$ and right end point by $right(I[t])$. The objective is to find the maximum sized subset of non-overlapping intervals in the set $\mathcal{I}$.

Without any space restriction, the best known result for this problem is by Snoeyink [8] which shows that it can be solved in $O(n \log k)$ time where k is the size of the maximum independent set. An $O(n \log n)$ time greedy algorithm for this problem works as follows [7]:

*School of Computing Science, Simon Fraser University, Canada, binay@cs.sfu.ca

†The Technion – Israel Institute of Technology, Haifa, Israel, minati@cs.technion.ac.il

‡Indian Statistical Institute, Kolkata, India, nandysc@isical.ac.in

§Chennai Mathematical Institute, Chennai, India, sasanka@cmi.ac.in

GREEDY-ALGO
Step 1: Sort the intervals of $\mathcal{I}$ according to their right end-points.
Step 2: Find the interval i, whose right end-point is left-most. Include i in the solution set MIS.
Step 3: Remove all intervals of set $\mathcal{I}$ which are overlapping with i.
Step 4: Repeat Step 2 & 3 until $\mathcal{I}$ is not empty.
Step 5: Report MIS.

2.1 Exact algorithms

Now, we describe algorithms in the space-efficient models for this problem.

1. An $O(nk)$ time (where k is the size of the maximum independent set) and $O(1)$ extra-space algorithm for this problem is quite obvious which mimics the GREEDY-ALGO. As it can not store the sorted intervals, the execution of Step 2 for choosing the left-most right end-point discarding all the intervals that starts before the last chosen right end-point, takes $O(n)$ time, i.e, reporting each member of MIS takes $O(n)$ time. Thus the time complexity follows. We will refer this algorithm as ALGO-2.

2. Here, we describe an algorithm which takes $O(s)$ space and $O(\frac{n^2}{s} + n \log s)$ time to report all the members of the MIS.

   Assume the input is given in the read-only array I. The algorithm uses a min-heap H of size $O(s)$ words (i.e each containing $O(\log n)$ bits). We will divide the array I into s disjoint blocks each containing $\lceil \frac{n}{s} \rceil$ consecutive elements of the array I (except the last one which may contain fewer elements). We denote the i-th block as B_i. So, for $i \in \{1, 2, \ldots, s-1\}$, B_i contains the elements $\{I[(i-1)\lceil \frac{n}{s} \rceil + 1], \ldots, I[i\lceil \frac{n}{s} \rceil]\}$, and B_s contains $\{I[(s-1)\lceil \frac{n}{s} \rceil + 1], \ldots, I[n]\}$. Throughout the algorithm, the following invariant is maintained

   Invariant 1 *At most one interval from each block is in the heap H.*

   Initialize H: From each block find the interval whose right end-point is left-most within that block and insert these intervals into the min-heap H according to the value of their right end-points. Remember that the heap will ideally contain the indices of these intervals in the array I.

Algorithm 1: Exact-MIS-For-Intervals

Input: An array $I[1, \ldots, n]$ containing n intervals
Output: Report the maximum independent set of the given intervals

```
1  Initialize H;
2  E = 0; //E indicates the number of blocks where all the intervals are processed
3  Pop an interval from H. Let t be the index in the array I of that interval.
4  ξ = right(I[t]);
5  Report t as an element of MIS;
6  while E ≠ s do
7      j = ⌈t / ⌈n/s⌉⌉; i.e, B_j is the block where the index t belongs;
8      Now find the interval, c, whose right end-point is left-most amongst all the intervals in B_j with left boundary starting after ξ; This can be done by scanning the sub-array {I[(j-1)⌈n/s⌉ + 1], ..., I[j⌈n/s⌉]} once.
9      if No such interval c is found then
10         E = E + 1;
11     else
12         Insert c into H;
13     Pop an interval from H. Let t be the index in the array I of that interval.
14     if the left end-point of the interval I[t] starts after ξ then
15         ξ = right(I[t]);
16         Report t as an element of MIS;
```

Time and Space Complexity The initialization takes $O(n)$ time. Each interval is considered to be inserted into the heap H for at most once. As each insertion needs a search within the block taking $O(\frac{n}{s})$ time and insertion/deletion in/from the heap takes $O(\log s)$ time, the total time complexity of the algorithm is $O(n(\frac{n}{s} + \log s))$, i.e, $O(\frac{n^2}{s} + n \log s)$.

The space-complexity is $O(s)$ since the size of the heap H is s and it uses a constant number of variables.

Theorem 1 *Given a set of intervals in a random-access read-only array, the maximum*

independent set can be computed in $O(\frac{n^2}{s} + n \log s)$ *time using only* $O(s)$ *extra-space.*

2.2 Lower-bound on time-space product

We will show that decision version of the element uniqueness problem can be reduced to finding maximum independent set of intervals.

Element Uniqueness Decision Problem: Given a set of n integers $a_1, a_2, \ldots, a_n$, report 1 if for all the pairs $((a_i, a_j),\ i \neq j)$ implies $a_i \neq a_j$, otherwise report 0.

If we have an algorithm A which can compute the *maximum independent set of intervals*, then we can correctly compute the *Element Uniqueness Decision Problem* for any input instance as follows. The algorithm A will consider the input instance $a_1, a_2, \ldots, a_n$ of *Element Uniqueness Decision Problem* and it considers each of them as degenerate interval having same left and right end-points. If A gives maximum independent set of size n then we report 1, otherwise we report 0.

So, if we have a good algorithm for computing *the maximum independent set of intervals*, then we have a good algorithm for *Element Uniqueness Decision Problem*. In other words,

Lemma 2 The problem of computing the maximum independent set of interval *is as hard as* Element Uniqueness Decision Problem.

Long ago, Borodin et al. [3] conjectured that the lower bound of the time-space product for *Element Uniqueness Decision Problem* is $\Omega(n^2)$. Yao [9] proved that time-space product lower bound for this problem is $\Omega(n^{2-\epsilon})$, where $\epsilon = O(1/(\log n)^{1/2})$.

2.3 Approximation Algorithms

The algorithm for computing an $O(\log n)$-approximate MIS for intervals in $O(\frac{n^2}{\log n})$ time using $O(1)$ extra-space is easy to obtain. Split the members in $\mathcal{I}$ into $\lceil \log n \rceil$ blocks, each consisting of $O(\frac{n}{\log n})$ elements, excepting the last block, which may contain fewer elements. In each block, apply ALGO-2 separately to compute the maximum independent set, which may take $O(\frac{n^2}{\log^2 n})$ time using $O(1)$ extra-space. Thus, the overall time for processing all the $\log n$ blocks is $O(\frac{n^2}{\log n})$; the space complexity remains $O(1)$. Now, report the MIS of the block for which the size of the output is maximum. The approximation ratio of this algorithm is $O(\log n)$ since (i) the size of the MIS for all the intervals in $\mathcal{I}$ is less than or equal to the sum of sizes of the MIS in these $\log n$ blocks, and (ii) the size of the reported MIS is greater than the average size of the MISs' in these $\log n$ blocks.

We can generalize this method to get the following result:

Theorem 3 *Given a set of* n *intervals* $\mathcal{I}$ *in a read-only memory, we can obtain a* α*-approximation result for the maximum independent set of* $\mathcal{I}$ *in* $O(\frac{n^2}{\alpha s} + n \log s)$ *time using* $O(s)$ *space.*

Proof. In order to get a α approximate maximum independent set, we need to split $\mathcal{I}$ into α blocks each of size $\frac{n}{\alpha}$. Now, since $O(s)$ space is available we can use Algorithm 1 instead of ALGO-2. Thus, processing each block requires $O(\frac{(n/\alpha)^2}{s} + \frac{n}{\alpha} \log s)$, and hence the overall time complexity for processing all the α blocks becomes $O(\frac{n^2}{\alpha s} + n \log s)$; the space complexity is $O(s)$. □

Conjecture 1 *Given a set of* n *intervals* $\mathcal{I}$ *in a read-only memory, the time and space product to obtain a* α*-approximation result for the maximum independent set of* $\mathcal{I}$ *is* $\Omega(n^2/\alpha)$*, where* $\alpha = o(n)$*.*

$(1+\epsilon)$-approximation algorithm in $O(\frac{1}{\epsilon})$ passes using $O(k)$ space:

Here we use the (streaming) single pass 2-approximation algorithm of Emek et al. [6] as a subroutine. This subroutine takes $O(k)$ space where k is the number of reported intervals. In the first pass, the algorithm evokes this subroutine and stores the 2-approximation of the MIS in the extra-space $Temp$. Let $Temp[1, 2, \ldots, k]$ be these intervals sorted by their left end-points.

All the input intervals lie in $[-\infty, \infty]$. We call it the *initial-interval-span*. For any positive integer $1 < p < k$, we partition this initial-interval-span into $\lceil \frac{k}{p} \rceil$ *smaller-interval-span* as follows. The intervals $Temp[p], Temp[2p], \ldots Temp[\lceil \frac{k}{p} \rceil - 1]$ are considered as the *barriers* between two consecutive smaller-interval-spans. So, the i-th smaller-interval-span is $[right(Temp[(i-1)p]), left(Temp[ip])]$, where $i \in \{2, 3, \ldots, (\lceil \frac{k}{p} \rceil - 1)\}$. The first and $\lceil \frac{k}{p} \rceil$-th smaller-interval-span are $[-\infty, left(Temp[p])]$, $[right(Temp[\lceil \frac{k}{p} \rceil - 1]), \infty]$, respectively. Let Int_i be the set of all the intervals in $\mathcal{I}$ whose both ends are in the i-th smaller-interval-span, where $i \in \{1, 2, \ldots, \lceil \frac{k}{p} \rceil\}$. Let $Opt(Int_i)$ denote the maximum independent set for all the intervals in Int_i.

As $|Opt(Int_i)| \leq 2p$, we can find the $Opt(Int_i)$ by scanning $2p$ times the entire input array I using $O(1)$ space as given in ALGO-2. Here we do the processing of Int_i, $i = 1, 2, \ldots, \lceil \frac{k}{p} \rceil$, in parallel. We use an array H_1 of size $\lceil \frac{k}{p} \rceil$. During the first scan, $H_1[i]$ stores the interval whose right end-point is reached first among all the intervals observed so far in Int_i. So, after the first scan, we can correctly identify the left most member of $Opt(Int_i)$ for each i. In a similar way, we can report the next left most member of $Opt(Int_i)$, for each i, in the next scan, and so on. Our algorithm reports all $Opt(Int_i)$ and the barriers as output.

Thus, in at most $2p + 1$ passes an independent set of the intervals in $\mathcal{I}$ is reported. The total space requirement is $O(k + \frac{k}{p}) \approx O(k)$. Now, we claim that this independent set is of size $(1 - \frac{2}{p})|OPT|$, where OPT is the maximum independent set (MIS) for the intervals in $\mathcal{I}$. If our algorithm reports A intervals, then $A \geq |OPT| - \frac{k}{p}$. The reason is that we may miss one member of MIS at each barrier as the 2-approximation algorithm of Emek et al. [6] ensures that no interval of $\mathcal{I}$ is properly contained in any of the intervals $Temp[p], Temp[2p], \ldots Temp[\lceil \frac{k}{p} \rceil - 1]$. So, we may miss at most $\frac{k}{p}$ intervals. Since $k \leq 2|OPT|$, we have $A \geq (1 - \frac{2}{p})|OPT|$. Thus, we have the following result:

Theorem 4 *Given a set $\mathcal{I}$ of intervals in a one-way read-only input tape, an $(1 + \epsilon)$-approximation result for the maximum independent set of $\mathcal{I}$ can be obtained in $O(\frac{1}{\epsilon})$ passes using only $O(k)$ space, where k is the number of reported intervals.*

3 Maximum Independent Set for Tree

Let $T = (V, E)$ be a tree where V is the set of vertices (or nodes) and E is the set of edges. The objective is to compute the maximum independent set of vertices of the tree T. As in [1], we assume that the tree T is represented as a DCEL (doubly connected edge list) in a read-only memory where for a vertex $u \in V$, we can perform the following queries in constant time using constant space:

- *Parent*(u): returns the parent of the vertex u in the tree T,
- FirstChild(u): returns the first child of u in the tree T,
- NextChild(u, v): returns the child of u which is next to v in the adjacency list of u.

Here we can perform post-order traversal starting from any vertex in $O(|V|)$ time using $O(1)$ extra-space.

3.1 Exact algorithm

If the depth of the tree is h and $O(h)$ amount of extra-space is given, then using a post-order traversal on the tree T, we can compute the maximum independent set in $O(n)$ time following a dynamic programming paradigm.

For a node v, let $v(I^+)$ denote the size of the maximum independent set of the subtree rooted at v including the node v. Similarly, let $v(I^-)$ denote the size of the maximum independent set of the subtree rooted at v excluding the node v. So, $v(I^+) = 1 + \sum_{u \in children(v)} u(I^-)$ and $v(I^-) = \sum_{u \in children(v)} \max\{u(I^-), u(I^+)\}$.

While processing a node v, we dynamically allocate two temporary variables A^+ and A^- for the node v to compute its $v(I^+)$ and $v(I^-)$ values, respectively. At the end of processing the node v (i.e., when all its children are processed), these contain $v(I^+)$ and $v(I^-)$. We propagate these to the parent of the node v to update the A^+ and A^- attached to its parent, and release the variables A^+ and A^- attached to v. Thus, we may have to maintain these temporary variables for all the nodes along a path from the root to the leaf level of the tree in the worst case. Thus, we have the following:

Theorem 5 *Given a tree T with n nodes in a read-only memory, the maximum independent set can be reported in $O(n)$ time using $O(h)$ extra-space, where h is the height of the tree T.*

3.2 $\frac{5}{3}$-approximation algorithm using constant space

We will do a post-order traversal in the given tree T. When all the children of a node p have been traversed, then we take a decision regarding whether p is to be reported as a member in the independent set or not as follows:

- If p has at least two children, then p is not reported.
- If p has 1 child, then report it if its child is not reported.
- If p has no child, then p is reported.

This reported set is obviously an independent set.

We use two one-bit variables *state* and i. We also use two pointer variables p and q, each of $O(\log n)$ bits; p stores the current node during the traversal and q is a temporary variable to be used during the backtrack. The variable *state* contains 0 or 1 depending on whether we reached p during the forward traversal or backtrack. The variable i contains 0 or 1 if any successor of p is reported. Both of them are initialized as 0 at the beginning of the execution. These variables maintain the following invariant:

Invariant 2 *If p is reached with state $= 1$, then we have visited all the children of the node pointed by the variable p.*

The pseudo-code of the algorithm is given in Algorithm 2.

Algorithm 2: Approx-MIS-For-Tree

Input: A tree $T = (V, E)$ given as DCEL
Output: Report an independent set

```
1  p = root;
2  state = 0; i = 0;
3  while p ≠ root or state ≠ 1 do
4      if state = 0 then
5          if p has a child then
6              p = FirstChild(p);
7          else
8              state = 1;
9      else
10         // state = 1 and p ≠ root
11         if {i = 0 and p has at most 1 child}
           then
12             Report p; i = 1;
13         else
14             i = 0;
15         q = Parent(p);
16         if NextChild(q, p) ≠ FirstChild(q)
           then
17             p = NextChild(q, p);
18             state = 0; i = 0;
19         else
20             p = q; state = 1;
```

Lemma 6 *Our algorithm produces a solution SOL having size $|SOL| \geq \frac{3}{5}|OPT|$, where OPT is the optimum solution.*

Proof. Let $T = (V, E)$ be a tree with n nodes. Let v be any node in T having degree $\delta(v)$. We classify v as *type-1*, *type-2* and *type-3* depending on on whether its degree $\delta(v) = 1$, $\delta(v) = 2$ or $\delta(v) > 2$, excepting the following:

- if v is the root of the tree and $\delta(v) \geq 2$, then it is considered as a *type-3* node.

Note that our algorithm does not report any *type-3* node in the independent set. Assume that our algorithm reports SOL as the independent set, and $|SOL| = (s_1 + s_2)$, where s_1 (resp. s_2) is the number of *type-1* (resp. *type-2*) nodes. So, the number of leaf-nodes in T is s_1. Similarly, let OPT be the maximum independent set, and $|OPT| = (m_1 + m_2 + m_3)$, where m_1, m_2 and m_3 are the number of *type-1*, *type-2* and *type-3* nodes, respectively. We will give an upper bound of $|OPT|$.

We can obtain a tree T' and a collection of chains $\mathcal{C}$ as follows:

> Let $P = v_0 \to v_1 \to v_2 \to \ldots \to v_k \to v_{k+1}$ $(k \geq 1)$ be a path in the tree T where $C = v_1 \to v_2 \to \ldots \to v_k$ consists of *type-2* nodes in T, and v_0 and v_{k+1} are the two *non-type-2* nodes present at the two ends of C (see Figure 1(a)). We denote C as a *chain*. Remove C from T, and connect (v_0, v_{k+1}) in the reduced tree T' (see Figure 1). Include the chain C in $\mathcal{C}$. Do this for all the chains in T.

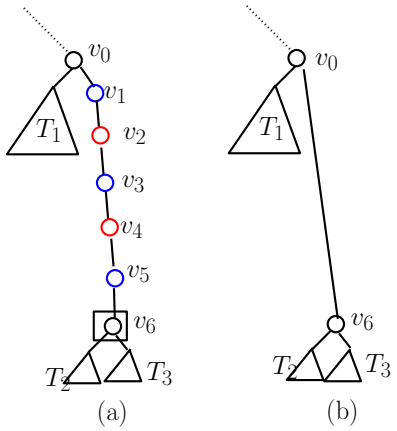

Figure 1: Proof of Lemma 6 - getting the reduced tree T' from the tree T

Now, observe the following:

- $\mathcal{C}$ consists of all the *type-2* nodes of T.
- T' consists of all the *type-1* and *type-3* nodes of T. In particular, degree of each node of T' is same as in T.

Note that for a chain $C \in \mathcal{C}$, our algorithm gives the maximum independent set of C excepting the following situation.

If C is a chain consisting of odd number of nodes and in the tree T one end of C is attached with a leaf-node (*type-1* node), then our algorithm reports $|OPT(C)| - 1$ many nodes, where $OPT(C)$ is the maximum independent set of the chain C.

Let $m_2 = (s_2 + t)$. Then at least t chains are there for which the above exception holds. Note that for each of these t chains, the corresponding attached leaf-nodes of T (*type-1*) are not reported in the optimum. So, OPT contains at most $(s_1 - t)$ *type-1* nodes. So, $m_1 + m_2 \leq (s_1 - t) + (s_2 + t)$.

As s_1 is the number of leaves in T', the number of internal nodes in T' can be at most $s_1 - 1$ because of the following reason.

Let k' be the number of internal nodes in T'. As each non-leaf nodes (excepting root of T' which may have degree at least 2) has degree at least three, so by $2|E| = \sum \delta(v)$, we get $2(k' + s_1 - 1) \geq 3(k' - 1) + 2 + s_1$. So $k' \leq s_1 - 1$.

Here, we give an upper bound of m_3 as follows.

Now, consider the tree T'' after removing all the *type-1* nodes from T'. First, we argue that $|OPT(T'')| \leq \frac{2}{3}s_1$.

We refer an edge $e \in T''$ as *covered*, if exactly one adjacent vertex of e belongs to $OPT(T'')$. Note that at most all the edges of T'' can be covered in $OPT(T'')$. We will now see at most how many nodes are enough to cover all the $s_1 - 2$ edges of T''. This will give an upper bound of $OPT(T'')$. As all the internal nodes of T'' has degree at least 3, so an internal node v is in the $OPT(T'')$ implies that three edges are covered by that node. Similarly, a leaf node of $OPT(T'')$ will cover only one edge of T''. As the leaves of T'' are actually *type-3* nodes of T and there are only s_1 leaves in T, so there may be at most $s_1/2$ leaves in T''. Hence, we need at most $\frac{s_1}{2} + \frac{s_1/2 - 2}{3} \leq \frac{2}{3}s_1$ nodes to cover all the edges of T''. So, $|OPT(T'')| \leq \frac{2}{3}s_1$

Now, observe that in the OPT if two adjacent nodes v' and v'' of T'' appears, then there must be a chain $C = v_1 \to v_2 \to \ldots \to v_k$ in between v' and v'' in T, and v_1 and v_k are not in OPT, where $k > 0$. This means that OPT contains at most $|OPT(C)| - 1$ nodes from the chain C. Remember that our algorithm always report $OPT(C)$ in this situation. So, if OPT contains $\frac{2}{3}s_1 + t_1$ *type-3* nodes, then $m_1 + m_2 \leq s_1 + s_2 - t_1$. So, $m_1 + m_2 + m_3 \leq s_1 + s_2 + \frac{2}{3}s_1$.

Thus, our algorithm reports $|SOL| = s_1 + s_2$, whereas the optimum may report $|OPT| = (m_1 + m_2 + m_3) \leq (s_1 - t) + (s_2 + t) + m_3$. So, the approximation ratio $|\frac{SOL}{OPT}| \geq \frac{s_1 + s_2}{s_1 + s_2 + \frac{2}{3}s_1} \geq \frac{s_1 + s_2}{s_1 + s_2 + \frac{2}{3}(s_1 + s_2)} \geq \frac{3}{5}$. □

As a result, we have the following:

Theorem 7 *Given a tree T with n nodes in a read-only memory, a $\frac{5}{3}$-approximation result of the maximum independent set of T can be obtained in $O(n)$ time using $O(1)$ extra-space.*

4 Conclusion

In this note, we consider the problem of designing the space efficient algorithms for two types of graphs, namely interval graphs and trees, whose maximum independent set can be computed in polynomial time. For interval graph, we assume that the interval representation is given. Our proposed exact algorithm runs in $O(\frac{n^2}{s} + n \log s)$ time using $O(s)$ extra-space. This almost achieves the lower bound for the *time* × *space* product for this problem. A k-approximate maximum independent set in $O(n^2/k)$ time using $O(1)$ extra-space is easy to obtain assuming that the read-only input array is given in a random access memory. Our proposed multi-pass algorithm assumes that the input is given in a read-only tape; it reports $(1 + \epsilon)$-approximate maximum independent set by executing $(1/\epsilon)$ passes over the input stream, and using $O(k)$ extra-space where k is the size of the output. For trees, we assume that the input is given in the form of doubly connected edge list in the read-only random access memory. The proposed $\frac{5}{3}$-approximation algorithm runs in $O(n)$ time using $O(1)$ extra-space.

Acknowledgement: The authors acknowledge Jaikumar Radhakrishnan, Venkatesh Raman and Saket Saurabh for the fruitful discussions related to this work.

References

[1] T. Asano, W. Mulzer, and Y. Wang. Constant-work-space algorithms for shortest paths in trees and simple polygons. *J. Graph Algorithms Appl.*, 15(5):569–586, 2011.

[2] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[3] A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa. A time-space tradeoff for sorting on non-oblivious machines. *J. Comput. Syst. Sci.*, 22(3):351–364, 1981.

[4] B. Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 193–242. 1990.

[5] M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *FOCS*, pages 143–152, 2010.

[6] Y. Emek, M. M. Halldórsson, and A. Rosén. Space-constrained interval selection. In *ICALP (1)*, pages 302–313, 2012.

[7] U. I. Gupta, D. T. Lee, and J. Y.-T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467, 1982.

[8] J. Snoeyink. Maximum independent set for intervals by divide and conquer with pruning. *Networks*, 49(2):158–159, 2007.

[9] A. C.-C. Yao. Near-optimal time-space tradeoff for element distinctness. *SIAM J. Comput.*, 23(5):966–975, 1994.