Space Efficient and Output Sensitive Greedy Algorithms on Intervals

Toshiki Saitoh[1,a)] Takashi Horiyama[2,b)] David Kirkpatrick[3,c)] Yota Otachi[4,d)]
Ryuhei Uehara[4,e)] Yushi Uno[5,f)] Katsuhisa Yamanaka[6,g)]

Abstract:
In this paper, we consider fundamental problems on the following machine model: An input is stored in read-only random-access memory, a limited random access workspace is available, and we report the output to a write-only media. We first propose efficient greedy algorithms using priority queues on the machine model as a general framework. Then, we apply the greedy algorithm to maximum independent set problem on intervals. Our algorithm runs in any workspace $O(s)$ and it has the same time complexity of Snoeyink's algorithm which is the best known result if $s = \Omega(n)$.

1. Introduction

Recently, there are many huge size data sets, such as biological data, social networks, and so on. Because of the huge size of the data, we demand algorithms running in a limited workspace. To consider the memory constraint, we propose algorithms on the *read-only random-access machine model* [5], [7]; the input is stored in a read-only random-access memory, a limited random access workspace is available, and we report the output to a write-only media in the machine model.

On the other hand, a greedy strategy is significant in algorithm theory because we can find optimal or approximate solutions for many combinatorial optimization problems by using the strategy, for example, shortest paths on a graph, knapsack problem, and so on [4], [9]. On greedy algorithms, we select an element which has highest priority (minimum or maximum value) as an output from candidates for each step. Borodin et al. studied a general framework of greedy algorithms [3]. They mention that there are two types of models for the priorities; fixed priority model and adaptive priority model. In the fixed priority model, a total order of the priorities of all elements in the input is specified in the beginning. On the other hand, in the adaptive priority model, the ordering depends on the elements which have already outputted. Considering the adaptive priority model on the read-only random-access model, it is difficult to compute the priorities of all candidates for each step because we cannot remember all of the values and the outputted elements on the space constraint. Thus, we consider the fixed priority model in this paper. To develop efficient greedy algorithms, it is important how to find the next element which is highest priority, efficiently. To obtain the element quickly, priority queues play a crucial role in the greedy strategy.

Priority queues are data structures that store a collection of elements and support *find-min*, *insert*, and *extract* operations. On the read-only random-access machine model, *memory adjustable* priority queues which uses $O(s)$ words or bits of workspace for a given parameter $s(\leq n)$ are proposed, for example tournament trees, navigation piles, and heaps [1], [2]. In a greedy algorithm adopting the operations of one of the data structures, we first insert all the elements in the input into the data structure. We then extract the minimum (or maximum) element from the data structure and we output the element if it is feasible. The

1 Department of Electrical and Electronic Engineering, Kobe University
2 Graduate School of Science and Engineering, Saitama University
3 Department of Computer Science, UBC
4 School of Information Science, JAIST
5 Graduate School of Science, Osaka Prefecture University
6 Department of Electrical Engineering and Computer Science, Iwate University
a) saitoh@eedept.kobe-u.ac.jp
b) horiyama@al.ics.saitama-u.ac.jp
c) kirk@cs.ubc.ca
d) otachi@jaist.ac.jp
e) uehara@jaist.ac.jp
f) uno@mi.s.osakafu-u.ac.jp
g) yamanaka@cis.iwate-u.ac.jp

algorithm repeats the process until there is no elements in the data structure. The *extract* operation of the memory adjustable priority queues takes $O(\lg s)$ [*1] time on the structure and we extract all the elements one by one. Hence, the algorithm takes $O(n \lg s)$ time on the structure and the running time does not reach output sensitiveness. It is because the operations of the data structures are versatile. We need to develop specific operations for greedy algorithms to propose faster algorithms.

In this paper, we propose a new extract operation *refresh* on memory adjustable tournament trees and navigation piles on the machine model for greedy algorithms. The operation *refresh* is simple and natural improvement of *extract* to get the next element for each step in the greedy algorithms. We show an interesting analysis of binary trees to prove that the greedy algorithms using the operation runs fast. By applying our greedy algorithm, we propose an efficient algorithm for the maximum independent set problem on intervals. The algorithm runs in $O(m(\lg sk/m + n/s))$ time where n is the size of input, k is the size of the output, s is the size of words on the tournament tree or bits on the navigation piles, and $m = \min(n, sk)$. The running time of the algorithm achieves output sensitiveness which depends on the output size. In this paper, we first define the new operation *refresh* on memory adjustable priority queues for greedy algorithms in Section 2. Then, we apply our algorithm for the maximum independent set problem on intervals in Section 3.

Related works. The multi-pass streaming model is also studied in space-constraint situation. This model is introduced by Munro and Paterson for selection and sorting problems [11]. On the model, the input is stored in the read-only sequential-access media and we analyze algorithms by counting the number of passes of the media. Emek et al. studied the maximum independent set problem of intervals on the streaming model [6]. They proposed 2-approximation streaming algorithm and $1 + 1/(2p - 1)$-approximation multi-pass streaming algorithm where p is the number of passes.

On the offline setting, there is a greedy algorithm for the maximum independent set problem on intervals and the algorithm runs in $O(n \lg n)$ time because it requires to sort the intervals by right endpoints [4], [9]. Snoeyink proposed an output sensitive algorithm by using divide-prune-and-conquer and the algorithm runs in $O(n \lg k)$ time where k is the output size of the solution [12]. However, these algorithms need extra $\Omega(n)$ workspace. Recently, Bhattacharya et al. studied the problem on the machine model with space constraint [2]. Their algorithm uses a memory adjustable min-heap with size $s(\leq n)$ words. The algorithm runs in $O(m(\lg s + n/s))$ and this implies that if $s = O(n)$, the time complexity is same as the greedy algorithm using sort, and not output sensitive. Our independent set algorithm is a improvement of the Bhattacharya's algorithm and a generalization of the Snoeyink's algorithm.

[*1] We use the symbol $\lg x$ to denote $\log_2 x$.

2. Greedy Algorithms and Priority Queues

In a greedy strategy, we choose the best candidate as an output for each step and we repeat the process until there is no candidates. On a step, an element is *feasible* if the set which includes the current output set and the element is feasible, and is *infeasible* otherwise. To find the best feasible element effectively, memory adjustable priority queues are useful in the space constraint.

In this section, we define the memory adjustable priority queues, tournament trees and navigation piles, proposed by Asano et al. [1]. Tournament trees are also called selection trees [10] and use $O(s)$ words of workspace. The second structure is a navigation pile which is proposed by Katajainen et al. [8]. The original navigation pile stores $O(n)$ bits where n is the size of an input. Asano et al. improve the data structure to memory adjustable structure and the workspace of memory adjustable navigation pile is $O(s)$ bits for $\lg n \leq s \leq n/\lg n$. The two data structures support *find-min* and *insert* in $O(1)$ time, and *extract* in $O(n/s + \lg s)$ time.

We have the following assumptions to describe the priority queues for greedy algorithms:

(1) A parameter s is a power of 2.
(2) All elements in the input are inserted in the data structures as a candidate set before extractions.
(3) The elements are extracted from the data structure in monotonic fashion. At any given point of time, we keep the latest outputted element that the elements smaller than or equal to that have been extracted from the data structure.
(4) The two manipulation of comparison and *feasibility-check* which decides whether or not an element is feasible can be computed in the workspace.

The greedy algorithms utilize comparison and feasibility-check to obtain the next elements for each step. We analyze the complexity of the greedy algorithms by counting the number of manipulations of comparisons and feasibility-checks. For the reason, we assume that these functions can be computed in the workspace. In particular, comparisons and feasibility-checks are available in $O(s)$ words while discussing the tournament trees, and in $O(s)$ bits while discussing the navigation piles.

2.1 Tournament Trees

In this subsection, we have $O(s)$ words of ex-

tra workspace. We separate an input array $A = \{e_1, \ldots, e_n\}$ into s buckets $\{B_1, \ldots, B_s\}$ such that $B_i = \{e_{(i-1)\lceil n/s \rceil + 1}, \ldots, e_{i\lceil n/s \rceil}\}$ for each i. The size of each bucket B_i is $\lceil n/s \rceil$ for $i \in \{1, \ldots, s-1\}$, and $|B_s| = n - (s-1)\lceil n/s \rceil$. For each bucket B_i, at most one element is in a tournament tree.

A tournament tree is a complete balanced binary tree with s leaves. The number of nodes in the tree is at most $2s - 1$ and the height of the tree is $\lg s$. Each leaf l_i covers a bucket B_i and each internal node covers the buckets which are covered by the leaves in the subtree rooted at the internal node. Each node v stores a direct pointer to the element and the element is denoted by $e(v)$. The element $e(v)$ of a node v is minimum in the buckets covered by v; every internal node has at most two children and stores the smaller element of them. The size of the tournament tree is $O(s)$ since the number of nodes is $2s - 1$ and each node holds the direct pointer to an element using one word.

A tournament tree supports *find-min* in $O(1)$ time and *extract* in $O(n/s + \lg s)$ time. Since the minimum element is stored at the root in a tournament tree, *find-min* can be computed in $O(1)$ time. In the *extract* operation, given an element e, we extract the element e in the tournament tree. We suppose e is in a bucket B_i. To find the next minimum element, we scan the bucket B_i and we save the element in the leaf l_i covering the bucket B_i. Then, we update nodes from the leaf l_i to the root. The number of comparisons in *extract* is $O(n/s + \lg s)$ because we manipulate in $O(n/s)$ and $O(\lg s)$ times for scanning the bucket and updating the path, respectively.

We analyze the complexity of greedy algorithms using the *extract* operations. Using the naïve *extract* operations, the number of iterations of the greedy algorithms is $\Theta(n)$ because we extract all elements in the input from a tournament tree. We can reduce the number of iterations implementing simple idea. While selecting the next minimum element from a bucket, we do not choose an infeasible element as the next element even if the element is minimum. This extraction breaks the assumption (3). However, since the nodes keep the direct pointers to the elements, we can restore the elements of nodes of a tournament tree even if the latest extracted element has changed. Hence, the number of iterations of greedy algorithms becomes $\min(n, sk)$ because we output at least one element after extracting s elements where k is the size of an output. In this extraction, the number of comparisons and feasibility-checks are $O(n/s + \lg s)$ and $O(n/s)$ times for each *extract* operation, respectively.

To obtain the next output element efficiently, we propose *refresh* operation for a tournament tree. The *refresh* is bottom up and recursive procedure. The element of the root becomes minimum and feasible after executing the *refresh* operation. We describe the refresh procedure in **Procedure 1** and guarantee that the element of a node v_r is minimum and feasible after *refresh* by the lemma below.

Procedure 1: Refresh(a node v_r in the tree)

1 Assume that the element of v_r comes from a leaf l_i and l_i covers a bucket B_i;
2 Find an element e_i which is minimum and feasible in B_i;
3 Set the leaf l_i to node v and $e(v) = e_i$;
4 **while** v *is not* v_r **do**
5 Let v_s and v_p be the sibling of v and the parent of v, respectively;
6 **if** $e(v_s)$ *is infeasible* **then** Refresh (v_s);
7 **if** $e(v)$ *is smaller than* $e(v_s)$ **then** Set $e(v_p) = e(v)$;
8 **else** Set $e(v_p) = e(v_s)$;
9 Set $v = v_p$;

Lemma 1. Procedure 1 *updates the element of a node v_r such that the element is minimum and feasible in the buckets covered by v_r.*

Proof. We prove that the element of each node v in the procedure is minimum and feasible in the buckets covered by v. First, the node v is l_i and the element of l_i is minimum and feasible in B_i since Step 2 gets the minimum and feasible element by scanning all elements in B_i. If there is no feasible elements in B_i, we set infinity to $e(l_i)$.

In **while** loop, we reset the elements of the nodes on the path from l_i to v_r. For each iteration, we first check whether or not the element of the sibling v_s of v is feasible. If the element is infeasible, we call the procedure recursively and the procedure chooses the minimum and feasible element in the buckets covered by v_s. Then, we assign the smaller element in $e(v)$ and $e(v_s)$ as the parent's element in Steps 7 and 8. Thus, the element of v_p is minimum and feasible in the buckets covered by v_p. Finally, we assign v as v_p. The element of v is minimum and feasible in the buckets covered by v. We repeat this process until v becomes the node v_r. □

Next, we analyze the time complexity of the procedure. To analyze the time complexity, we show a significant lemma.

Lemma 2. *Let T be a binary tree with height h and s' leaves. The number of nodes in T is at most $s' \lg \frac{2^h}{s'} + s' - 1$.*

Proof. We prove the lemma by induction on height. In the base case, we assume that the height h of T is 1. The tree T has only one leaf and $1 \cdot \lg \frac{2^1}{1} + 1 - 1 = 1$.

We assume that the root of T with height h has exactly two children. Let s_l be the number of leaves in the left subtree. The number of leaves in the right subtree is $s' - s_l > 0$. The nodes in T is a union of the nodes of the left subtree, the nodes of the right subtree, and the root. From the hypothesis for height, the number of nodes in T is as follows.

$$
\begin{aligned}
&\left(s_l \lg \frac{2^{h-1}}{s_l} + s_l - 1\right) \\
&+ \left((s' - s_l) \lg \frac{2^{h-1}}{s' - s_l} + (s' - s_l) - 1\right) + 1 \\
= \; & s_l \lg \frac{2^{h-1}}{s_l} + (s' - s_l) \lg \frac{2^{h-1}}{s' - s_l} + s' - 1 \\
\leq \; & \frac{s'}{2} \lg \frac{2^{h-1}}{s'/2} + \frac{s'}{2} \lg \frac{2^{h-1}}{s'/2} + s' - 1 \qquad (1) \\
= \; & s' \lg \frac{2^h}{s'} + s' - 1
\end{aligned}
$$

Inequation (1) comes from the maximizing the entropy of the coin toss, namely, the number is maximum when $s_l = s'/2$. Thus, the lemma holds when the root has two children.

Next, we discuss the case that the root of the tree T has only one child. The nodes in T are the root and the nodes of the subtree rooted the child. We have the following equation from the hypothesis and $\lg \frac{2^{h-1}}{s'} = \lg \frac{2^h}{s'} - 1$.

$$
\begin{aligned}
s' \lg \frac{2^{h-1}}{s'} + s' - 1 + 1 \; &= \; s' \left(\lg \frac{2^h}{s'} - 1\right) + s' \\
&\leq \; s' \lg \frac{2^h}{s'} + s' - 1
\end{aligned}
$$

since $s' \geq 1$. □

From the lemma above, we analyze the time complexity of **Procedure 1** below.

Lemma 3. *Let r be the root of the tournament tree with size s, and let $s'(< s)$ be the number of refreshed elements by* **Procedure 1** *at r. The number of comparisons and feasibility-checks in* **Procedure 1** *is at most $s' \cdot \frac{n}{s} + s' \lg \frac{s}{s'} + s' - 1$ and the procedure runs in $O(s)$ space.*

Proof. We estimate the total cost of Step 2. Once the procedure is called, we refresh one element in Step 2. Thus, the number of calls of **Procedure 1** is s' since the number of refreshed elements in Step 2 is s'. For every element in B_i, we check whether or not the element is feasible and minimum in Step 2, and the size of B_i is $\lceil n/s \rceil$. Therefore, the number of both of comparisons and feasibility-checks in Step 2 are at most $s' \cdot n/s$ in total.

We analyze the size of a subtree T' which is constructed from nodes on updating paths in the whole process. Since the number of refreshed elements is s', T' is a binary tree with s' leaves. From Lemma 2, the size of the subtree is at most $s' \lg \frac{s}{s'} + s' - 1$ since the height of the tournament tree is $\lg s$. For each node v in T', we check feasibility of the sibling's element $e(v_s)$ of v and compare $e(v)$ and $e(v_s)$. The total number of both of comparisons and feasibility-checks in **while** iterations is $s' \lg \frac{s}{s'} + s' - 1$. Therefore, in **Procedure 1** for r, the number of comparisons and feasibility-checks are $s' \cdot n/s + s' \lg \frac{s}{s'} + s' - 1$ in total.

We discuss the space complexity of the procedure. The tournament tree is represented in $O(s)$ words. For each call of the procedure, we need to remember the argument node v_r using a constant word and the depth of recursion is at most $\lg s$. Hence, **Procedure 1** runs in $O(s)$ words. □

2.2 Navigation Piles

A navigation pile is a compact representation of a tournament tree [1], [8]. The main differences are following. The navigation pile represents s elements by using $O(s)$ bits for workspace in $\lg n \leq s \leq \frac{n}{\lg n}$. Any node stores a partial information of the position for the minimum element in the covered buckets of the node. We add an assumption for *refresh* operation on the navigation pile.

(5) If the element of a node is infeasible, the elements of its ancestors are also infeasible.

We separate the input array $A = \{e_1, \ldots, e_n\}$ into s buckets $\{B_1, \ldots, B_s\}$ such that $B_i = \{e_{(i-1)\lceil n/s \rceil + 1}, \ldots, e_{i\lceil n/s \rceil}\}$ for each i. The size of each bucket B_i is $\lceil n/s \rceil$ for $i \in \{1, \ldots, s-1\}$, and $|B_s| = n - (s-1)\lceil n/s \rceil$.

A navigation pile is a complete binary tree with $s/2$ leaves. The leaves have *height* 1 and internal nodes have height h if the nodes have children of height $h-1$. Each leaf covers two buckets and an internal node covers the buckets which are covered by the leaves in the subtree rooted at this internal node. Thus, each node with height h covers 2^h buckets and the size of 2^h buckets is at most $2^h \times \lceil n/s \rceil$. Each node with height h stores $2h$ bits. We divide the buckets covered by the node into $\lceil n/(s \cdot 2^h) \rceil$ contiguous elements, called *quantile*. Each node points to the quantile which contains the minimum element in the buckets covered by the node. The first h bits represents the bucket which contains the quantile and the second h bits represents the index of the quantile in the bucket. If $2h > \lceil \lg n \rceil$, nodes with height h have $\lceil \lg n \rceil$ bits and point to the minimum element in the buckets, directly. The number of nodes with height h is $s/2^h$ and each node stores $\min(2h, \lceil \lg n \rceil)$ bits. Therefore, the total number of bits of the navigation pile is $\sum_{h=1}^{\lg s} (S/2^h) \cdot \min(2h, \lceil \lg n \rceil) = O(s)$.

A navigation pile supports *find-min* and *extract* operations. To support the *find-min* operation in $O(1)$ time, we keep a separate pointer to the minimum element in the navigation pile by using $\lg n$ bits. This minimum pointer is updated with every *extract* operation.

Next, we explain how to extract an element in $O(\lg s)$ time from a navigation pile. We first get the bucket containing the extracted element. Then, we find the next minimum element of the bucket by scanning the bucket, and update the leaf covering the bucket pointer of the quantile which contains the element, that is, we update the information of the nodes on the path from the leaf to the root. To update each internal node on the path, we compute the two

elements of its children by scanning the quantiles pointed by the children. Then, we compare the two elements and save the pointer to the quantile which contains the smaller element at the node. We can access the quantile of an internal node in constant time because navigation bits are stored in a bit vector in breadth-first order. After getting to this position, we get the elements by scanning the quantile. For a node of height h, the size of the quantile is at most $\lceil n/(s \cdot 2^h) \rceil$. By summing on this formula over the updating path, both of comparisons and feasibility-checks is required $O(n/s + \lg s)$ times for the *extract* operation.

From the *extract* operation, we figure out that the *refresh* operation described in the previous section can be performed on the navigation pile in the same way. For each node on a updating path, we check whether or not the element of the sibling of the node is feasible. If the element is infeasible, we update the sibling, recursively. However, since we have only $O(s)$ bits for workspace, we cannot execute the recursive call, simply; we need to keep an argument node v_r using $\lg n$ bits and the depth of the recursive call is $\lg s$. In the greedy algorithm, we only extract the minimum element of the root of a navigation pile. Thus, we can expand the recursive call in the refresh procedure on $O(s)$ bits workspace. For each height, we manipulate comparisons and feasibility-checks in at most $s' \cdot \lceil n/(s \cdot 2^h) \rceil$ times because the quantile size of nodes with height h is $\lceil n/(s \cdot 2^h) \rceil$ and the number of nodes with height h is at most s' where s' is the number of updated leaves in the operation. Hence, $\sum_{h=1}^{\lg s} s' \cdot \lceil n/(s \cdot 2^h) \rceil = O(s' \cdot n/s)$. In addition the size of the touched nodes is $s' \lg \frac{s}{s'}$ from Lemma 2. Therefore, the running time of the procedure is $O(s' \cdot n/s + s' \lg \frac{s}{s'})$.

If the navigation pile satisfy the assumption (5), we can extract all infeasible elements by adopting *refresh*, simply. However, if the navigation pile does not guarantee the assumption (5), our refresh operation does not work correctly in a greedy algorithm. We use the latest output as a boundary value while scanning the quantile to find the corresponding element from a quantile. After changing the latest output, we cannot get back the corresponding elements which become infeasible before since nodes of the navigation pile have pointers to the quantile, not the direct pointers to the elements. In our refresh operation on the navigation pile, we need to extract all infeasible elements in the navigation pile. However, the navigation pile representing candidates may not satisfy the assumption (5) on some problems.

To get rid of the assumption (5), we use *restricted pile* which is also a navigation pile with $s/2$ leaves. We call a *candidate pile* an ordinary navigation pile for candidates. The restricted pile is isomorphic to the candidate pile and each node in the restricted pile covers the buckets which covered by the corresponding node in the candidate pile. Each node of the restricted pile represents a *restricted element* which is more restricted than the candidate, that is, if the candidate of a node is infeasible, the restricted element corresponding to the node is also infeasible. We assume that the restricted pile have the assumption (5), namely, if a restricted element of a node are infeasible, the elements of its ancestors are also infeasible.

We next discuss how to use a restricted pile for refresh. In a greedy algorithm, we first output the minimum and feasible candidate from a candidate pile on a step. Then, we refresh the restricted pile for refreshing the candidate pile. We can extract all infeasible elements in the restricted pile because the restricted pile hold the assumption (5). We have the condition that the restricted elements are infeasible if the candidates are infeasible. Hence, we can extract all infeasible elements in the candidate pile by updating the nodes which are infeasible in the restricted pile. The time complexity of this refresh of the candidate pile is same as that of the restricted pile. From above discussion, we have the following lemma.

Lemma 4. *Let T and T' be a candidate pile and its restricted pile. We can extract all infeasible elements in T by applying* refresh *operation for T' and we manipulate comparisons and feasible-checks in $O(s' \cdot n/s + s' \lg \frac{s'}{s})$ times where s' is the number of updated leaves in the restricted pile T'.*

We give an example using a restricted pile in the next section.

3. Algorithms for Maximum Independent Set on Intervals

In this section, we apply the method in Section 2 to the problem of finding the maximum independent set on intervals. A set of intervals is *independent* if for any two distinct intervals I and I' in the set, I do not intersect I'. The *maximum independent set problem* is to find a independent set which has the maximum cardinarity from an input intervals $\mathcal{I}$. We define $l(I)$ and $r(I)$ as the coordinate of the left and right endpoints of the interval $I \in \mathcal{I}$, respectively.

It is known that there is a greedy algorithm for the maximum independent set problem on intervals. Let $\mathcal{I}_A$ be the set of intervals which have been outputted. An interval $I \in \mathcal{I} \setminus \mathcal{I}_A$ is *infeasible* if there exists an interval $I' \in \mathcal{I}_A$ such that I intersect I'. Otherwise, I is *feasible*. For each iteration, the greedy algorithm outputs an interval with the minimum right endpoint in the feasible intervals. We can compare the right endpoints of two intervals in constant time. In the greedy algorithm described in **Algorithm 2**, we only check the feasibility by comparing the left endpoint of a interval with the right endpoint of the latest output. Hence, it takes $O(1)$ time for the feasibility-check.

Algorithm 2: Greedy Algorithm for Maximum Independent Set on Intervals

```
1 r = −∞;
2 while there is a feasible interval do
3     Find an interval I with minimum right endpoint in feasible
      intervals that r < l(I);
4     Output I and set r = r(I);
```

For implementing the algorithm, we first sort intervals with increasing order of the right endpoints simply. Then, we can find output intervals by sweeping the sorted array once. However, the implementation runs in $O(n \lg n)$ time and $\Theta(n)$ words for workspace. Thus, the implementation does not work in the space constraint.

In this section, we propose two algorithms for the maximum independent set problem on intervals by using memory adjustable priority queues described in Section 2.

3.1 Algorithm using Tournament Trees

We construct a tournament tree for storing intervals. The key of the tournament tree is right endpoints of intervals. For the problem, we apply refresh procedure on the tournament tree in **Algorithm** 2.

Theorem 5. *Our algorithm using tournament tree runs in* $O(m \cdot (\lg \frac{sk}{m} + \frac{n}{s})$ *time where* $m = \min(n, sk)$ *and* k *is the size of the optimal solution.*

Proof. From Lemma 3, the step 3 takes $s_i \cdot \frac{n}{s} + s_i \lg \frac{2s}{s_i} + s_i - 1$ time where s_i is the number of refreshed intervals for each iteration i. The algorithm takes $\sum_{i=1}^{k} s_i \cdot \frac{n}{s} + \sum_{i=1}^{k} s_i \lg \frac{s}{s_i} + \sum_{i=1}^{k} s_i - \sum_{i=1}^{k} 1$ time. We next prove that $\sum_{i=1}^{k} s_i \cdot \frac{n}{s} = O(m \cdot \frac{n}{s})$ and $\sum_{i=1}^{k} s_i \lg \frac{s}{s_i} = O(m \lg \frac{sk}{m})$ where $m = \min(n, sk)$.

Here, we prove that $\sum_{i=1}^{k} s_i \le \min(n, sk)$. For each iteration in the algorithm, we refreshed at most s intervals, that is, $s_i \le s$ because the tournament has at most s intervals. On the other hand, for each interval, the interval is stored into the tournament at most once in the algorithm since the threshold r increases monotonically. Thus the total number of refreshed intervals is at most $\min(n, sk)$. Therefore, $\sum_{i=1}^{k} s_i \cdot \frac{n}{s} = O(m \cdot \frac{n}{s})$.

Next, we prove that $\sum_{i=1}^{k} s_i \lg \frac{s}{s_i} \le m \lg \frac{sk}{m}$ inductively. We assume that $sk < n$, that is, $k < n/s$. Because the derivative of $s_i \lg \frac{s}{s_i}$ with respect to s_i is $\frac{\log s/s_i - 1}{\log 2}$, the root of this function is $\frac{s}{e}$ where e is a Napier's constant. $\sum_{i=1}^{k} s_i \lg \frac{s}{s_i} \le \sum_{i=1}^{k} \frac{s}{e} = \frac{\lg e}{e} sk = O(sk)$.

We assume that $sk \ge n$, that is, $k \ge n/s$. We prove that $\sum_{i=1}^{k} s_i \lg \frac{s}{s_i} \le m \lg \frac{sk}{m}$ by induction on k. We can prove the base case from the above discussion. From hypothesis,

$$
\begin{aligned}
&\sum_{i=1}^{k} s_i \lg \frac{s}{s_i} = \sum_{i=1}^{k-1} s_i \lg \frac{s}{s_i} + s_k \lg \frac{s}{s_k} \\
&\le (m - s_k) \lg \frac{s(k-1)}{m - s_k} + s_k \lg \frac{s}{s_k} \\
&\le \left(m - \frac{s}{k}\right) \lg \frac{s(k-1)}{m - s/k} + \frac{s}{k} \lg \frac{s}{s/k} \qquad (2) \\
&= m \lg \frac{sk}{m}.
\end{aligned}
$$

By using the discussion of the maximizing entropy, we set $s_i = \frac{s}{k}$ for each s_i in the inequality (2). □

Here, we have some discussion for the time complexity of our algorithm. If $n/s \le k$ then it takes $O(n \lg \frac{sk}{n} + n^2/s)$ time. This running time is better than Bhattacharya's algorithm because $k \le n$. The time-space product is $O(ns \lg \frac{sk}{n} + n^2)$. If $n/s > k$ then the algorithm runs in $O(kn)$ time. The time-space product is $O(ksn)$ and $ksn < n^2$. From this observation, if k and s are small, the time-space product is $o(n^2)$. If $s = n$, that is, the space is enough for the problem, our algorithm runs in $O(n \lg k)$ time. The running time is output sensitive and same as that of Snoeyink's algorithm.

3.2 Algorithm using Navigation Piles

We have $O(s)$ bits as workspace in this subsection. We construct two navigation piles, candidate pile and restricted pile in Section 2.2. The key of the candidate pile and the restricted pile is right endpoint and left endpoint, respectively. We have the following lemmas.

Lemma 6. *If the candidate interval of a node of the candidate pile is infeasible, the restricted interval of the corresponding node of the restricted pile is also infeasible.*

Proof. Since the candidate interval I of a node of the candidate pile is infeasible, $l(I)$ is smaller than or equal to r. Let J be the restricted interval of the corresponding node of the restricted pile. Because the key of restricted intervals is left endpoint, the left endpoint of J is smaller than or equal to that of the candidate interval, that is, $l(J) \le l(I) \le r$. Thus, J is also infeasible. □

Lemma 7. *If the restricted interval of a node is infeasible, the intervals of its ancestors are also infeasible.*

Proof. We suppose that the restricted interval I of a node is infeasible. Since the key of restricted intervals is left endpoint, the left endpoint of the interval of any ancestor is smaller than or equal to $l(I)$. □

From Lemmas 4, 6, and 7, we can extract all infeasible intervals from the candidate pile by refreshing the restricted pile. Thus, we can describe an algorithm using navigation piles for the maximum independent set problem on intervals and the analysis of the algorithms is similar to

Theorem 5.

Theorem 8. *Our algorithm using navigation piles runs in* $O\left(m \cdot \left(\lg \frac{sk}{m} + \frac{n}{s}\right)\right)$ *time where* $m = \min(n, sk)$ *and* k *is the size of the optimal solution.*

4. Conclusions

We have presented an efficient greedy algorithm with memory constraint using priority queues, and we have proposed a new operation *refresh* for designing efficient algorithms. We have applied our method to maximum independent set problem.

We can apply our algorithms for several dominating set problems on intervals, for example minimum (connected, totally, or paired) dominating set. For almost all of the problems, it is known that there are greedy algorithms for these problems. By applying our ideas, these problems can be solved in the same time of ours.

In this paper, we treat greedy algorithms with fixed priority model. It is an interesting future work to consider greedy algorithms with the adaptive priority model that the value of an element depends on the outputted elements. Many greedy algorithms, for example the shortest paths or minimum spanning tree on a graph, are on this model. The weighted problems on intervals can be solved in polynomial time using dynamic programming. However, it is difficult to construct a computational table with space constraint. We should consider how to realize dynamic programming on space constraint.

References

[1] Tetsuo Asano, Amr Elmasry, and Jyrki Katajainen. Priority queues and sorting for read-only data. In *Theory and Applications of Models of Computation, 10th International Conference, TAMC 2013, Hong Kong, China, May 20-22, 2013. Proceedings*, pages 32–41, 2013.

[2] Binay K. Bhattacharya, Minati De, Subhas C. Nandy, and Sasanka Roy. Maximum independent set for interval graphs and trees in space efficient models. In *Proceedings of the 26th Canadian Conference on Computational Geometry, CCCG 2014, Halifax, Nova Scotia, Canada, 2014*, 2014.

[3] Allan Borodin, Joan Boyar, Kim S. Larsen, and Nazanin Mirmohammadi. Priority algorithms for graph optimization problems. *Theor. Comput. Sci.*, 411(1):239–258, 2010.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] Omar Darwish and Amr Elmasry. Optimal time-space tradeoff for the 2d convex-hull problem. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 284–295, 2014.

[6] Yuval Emek, Magnús M. Halldórsson, and Adi Rosén. Space-constrained interval selection. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part I*, ICALP'12, pages 302–313, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] Greg N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *J. Comput. Syst. Sci.*, 34(1):19–26, 1987.

[8] Jyrki Katajainen and Fabio Vitale. Navigation piles with applications to sorting, priority queues, and priority deques. *Nordic J. of Computing*, 10(3):238–262, September 2003.

[9] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[11] J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[12] Jack Snoeyink. Maximum independent set for intervals by divide and conquer with pruning. *Netw.*, 49(2):158–159, March 2007.