Lower bounds on the space requirement of a class of geometric problems

R. Fessler, Th. Ottmann*, P. Widmayer
Departement Informatik
ETH Zentrum
CH-8092 Zurich, Switzerland

October 27, 1994

Abstract

The minimal slot assignment problem (MSAP) for a set of horizontal line segments in the plane asks for the computation of a minimal number of storage locations for the segments needed in a scanline algorithm which uses a semidynamic skeleton structure. We discuss various algorithms for solving the MSAP and are particularly interested in reducing their internal space requirements. As main results we derive linear space bounds for algorithms which solve the MSAP in such a way that they are allowed to access each externally stored segment exactly once. We present a new technique for deriving lower space bounds for a class of online algorithms for which the online restriction results from the assumption that not all input data can be held in core.

Our results show that the idea of unifying and simplifying a scanline algorithm by first computing a minimal slot assignment, then constructing a semidynamic skeleton structure based on the slot numbering, and finally to carry out the scanline algorithm using the semidynamic structure is infeasible if the internal space requirement must be of order $o(n)$ and each segment can be read only once.

*On leave from the University of Freiburg.

Keywords: plane sweep, skeleton structures, space reduction, minimal slot assignment, lower bounds, online algorithms.
Categories: E. 1, F. 2, F. 2.2, I. 3.5

1 Online algorithms: time, spatial, and storage restrictions

Currently there is a lot of interest in socalled *online* algorithms, that is, algorithms where decisions have to be made in temporal coherence with a given input sequence without knowing the future. A typical example is the online bin-packing problem where a sequence of objects of size less than one is presented one after the other. The objects have to be packed into bins of size one and the goal is to use the smallest possible number of bins for storing the objects. The next object has to be stored immediately after it appears in the input sequence. Therefore, the decision where to store it (in one of the already opened bins or in a new one) has to be made *online*, i. e. only based on the inputs seen and the decisions made so far. Once a decision is made it cannot be revised in the future, i. e. any rearrangement of objects is not allowed. There are many other problems with a similar flavor. We just mention the k-server problem first suggested in [MMS90].

Online algorithms for solving those problems are subject to a *temporal* constraint: Decisions have to be made on the past and the algorithms neither know nor can influence the future.

There is another class of online algorithms which are subject to a spatial rather than a temporal constraint: Consider the motion planning problem for a robot walking in an unknown terrain. The robot can see only some part of its environment and has to base its decision where to go next only on what he has seen so far. However, the robot may have influence on future inputs by its current decisions. See [K91] for an example of an algorithm of this kind.

Sleator and Tarjan [ST85] have introduced the concept of competitive analysis for online algorithms. (See also [MMS88].) They compare an online algorithm with the optimal offline algorithm solving the same problem rather than comparing any two online algorithms.

In this paper we discuss examples of problems where the online restriction results from the fact that only a limited amount of internal space is available. That is the algorithm is confronted with the situation that it is impossible to hold the whole input in core. Hence, the algorithm has to make decisions based on partial knowledge of relevant inputs, - a property which it shares with online algorithms subject to a temporal or spatial constraint. However, it may be possible that the algorithm has influence on the order in which inputs are read from external store into core.

To be a bit more specific, let us discuss the *online topological sorting* problem as an example for illustration: A topological sort of a directed acyclic graph (a dag, for short) $G = (V, E)$ with vertex set V and edge set $E \subseteq V \times V$ is a total linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. Now assume that input graphs are so large that it is impossible to store them as a whole in core. We may however assume that input graphs are stored externally and the corresponding outputs are also produced by writing to an external output medium. More specifically, let the input of any algorithm for solving the online topological sorting problem be a sequence of m pairs (u, v) representing edges of G stored on a disc or tape and assume that the output is a labelling of the vertices by integers s. t. $label(u) \leq label(v)$, if (u, v) is an edge of G.

An algorithm for solving the online topological sorting problem may read input pairs from external store, compute the edge-labelling, and write it to the external storage. Let us assume that the algorithm is subject to the constraint that it reads each input pair exactly once and that it can store at most k input pairs in core where $k << m$. There is no restriction on when the algorithm has to assign labels to vertices it has seen already (as first or second components of an input pair). However, once it has assigned a label to a vertex v, $(v, label(v))$ is written to external storage and no future change of $label(v)$ is possible anymore.

We give an intuitive argument why it might be impossible to compute a correct labelling for any given dag in this way, if the internal storage capacity k is fixed and independent of the number m of edges of the given dag, even, if we assume that $m = O(n)$ where n is the number of vertices of the dag. For assume on the contrary that there is an algorithm A that computes a correct labelling for any given dag as input sequence. If A has read more than k inputs, it must have "forgotten" at least one of them. More precisely, assume that the dag $G = (V, E)$ has node set $V \supseteq \{1, \ldots, k+2\}$ and that A has read $k+1$ input pairs $(1, 2)$, $(2, 3)$, $\ldots$, $(k+1, k+2)$ in some order.

Then there is at least one i such that the labelling computed by A does not depend on the pair $(i, i+1)$ with $1 \leq i \leq k+1$. Hence, A must compute the same labelling, if it reads the input sequence where the pair $(i, i+1)$ is replaced by the reverse pair $(i+1, i)$. Therefore the labelling computed by A is not a correct topological order either for the dag $G = (V, E)$ with node set $V \supseteq \{1, \ldots, k+1\}$ and edge set $E \supseteq \{(i, i+1) | 1 \leq i \leq k+1\}$ or for the dag $G\prime$ which results from G by replacing the edge $(i, i+1)$ by the reverse edge.

In order to turn this intuitive argument into a rigorous proof, one has to precisely formalize the computational model. We will not do this for the online topological sorting problem discussed above but for a geometrical problem with a very similar flavor. We conjecture that along the lines of arguments presented in the rest of this paper one can also show that the online topological sorting problem for arbitrary dags of size n cannot be solved using sublinear space, i. e. with an algorithm that uses only $o(n)$ space.

In the next section we present the geometrical problem which we discuss in this paper. Section 3 contains the formalization of the computational model and Section 4 a first lower space bound based on this model. Section 5 contains a second lower space bound for a more restricted class of algorithms. In the final Section 6 we conclude with a short discussion of the results and some open problems.

2 Scan line algorithms, skeleton structures and the slot assignment problem

Scanline, also called plane-sweep algorithms for solving geometrical problems have the advantage that it is typically not necessary to store all input data in core. In most cases it is sufficient to store the currently active objects, i. e. the objects currently cut by the scanline. Beyond that, a set of problem specific data is stored and maintained during the sweep (in the sweep history structure) which is also usually much smaller than the size of the whole input. This is the main reason why plane–sweep algorithms are attractive for solving geometrical problems occurring in VLSI-design, see [OW86] for a more profound motivation. For a general introduction into plane-sweep see

[NP82].

Plane–sweep solutions for geometrical problems often go hand in hand with the use of semidynamic skeleton structures. A semidynamic skeleton structure consists of an initially empty skeleton which is then dynamically filled during the sweep and provides room for all active objects encountered during the sweep ordered along the scanline. They support the same operations as their fully dynamic counterparts. Segment-, interval-, priority search trees, segment–range-, segment–segment–trees, and others are a few examples of structures existing in both variants. Semidynamic skeleton structures have the advantage that they are not only conceptually easier to understand but also have simpler maintenance algorithms. On the other hand it is a serious drawback of these structures that they typically have size linear in the number of line segments in the problem at hand. Hence, the construction of space–economical skeleton structures is a desirable goal which might please both practitioners and theoreticians alike. A crucial step in the construction of those structures is the solution of a problem which has been called the minimal slot assignment problem (MSAP), see [OW86]. It can be stated as follows.

> Given a set of n nonoverlapping horizontal line segments in the plane, find the smallest positive integer m which satisfies the condition that the segments can be assigned slots in the range $1 \ldots m$ such that the total order of line segments at each scanline position equals the total order given by their slot assignments.

A solution of the MSAP should not only determine the minimal number m of slots required but also assign labels (slot numbers) to the given input segments. The main problem which has been left open in [OW86] is the question whether a minimal slot assignment for line segments can be computed in sublinear space.

As stated above the problem is not formulated precisely enough for a rigorous answer. In particular any negative answer requires a careful reasoning about the underlying computational model. In this section we will discuss four possible solutions of MSAP based on different assumptions.

Algorithm 1: Let us assume first that the input segments are stored externally in left-to-right order, i. e. sorted according to increasing left endpoints. We assume that the segments are accessible in this order from external store.

For this first solution we additionally assume that we have direct access to the externally stored elements, i. e. once we have read an input segment we know its external address and can attach a slot number to a stored segment at any time without destroying the external storage order. Based on these assumptions we can easily solve the MSAP using only $O(1)$ space by the following "multiple sweep algorithm": Sweep m times over the set of inputs; during the i-th left-to-right scan slot number i is attached to all segments which consistently fit into the i-th slot ($1 \leq i \leq m$). These segments are determined by scanning the segments in external storage order, ignoring all segments which have already obtained a slot number and maintaining the currently topmost segment (without a slot number). Whenever an input segment is read from external store whose left endpoint is to the right of the right endpoint of the currently topmost segment S, slot number i can be attached to S.

This multiple sweep algorithm with direct access to the externally stored segments solves the MSAP in time $O(mn)$; the algorithm uses only $O(1)$ internal space.

Algorithm 2: Next let us assume that we have only sequential access to the externally stored segments. Think of a tape which holds the input segments in left-to-right order according to their left endpoints. Access to input segments is possible in sequential read-only mode. Attaching a slot number to a segment is difficult, if not impossible because it may require to rewind the tape, use a second tape for intermediate storage, and to partially copy the input tape. Of course, we may nevertheless again determine the respective topmost segments which have not yet obtained a slot number and attach the respective next slot number to them. For that purpose we first read the segments in left-to-right order from external store and always maintain the topmost (unnumbered) segment. Because we do not know whether or not a segment is topmost before we have passed its right endpoint we store always the currently active segments in core and write a segment to a separate external sto! re (a second tape, say) once the scan has passed its right endpoint.

Now assume that we can use the output tape as input tape for the next scan which will be carried out in the reverse direction. That is, we assume that we can read the segments in decreasing order, sorted according to their right endpoints. As before we can determine the segments which fit into the next slot and, at the same time, produce an output which can be used as input for the next scan in the reverse direction. Thus, by carrying out m sweeps back and forth using two tapes which can be sequentially read from both ends we can solve the MSAP in time $O(m \cdot n \log C)$ and internal space $O(C)$. Here, C is the so called cross section number, that is the maximum number of segments cut by the vertical scanline when sweeping over the n input segments. For, we maintain during the sweep the set of at most $O(C)$ currently active line segments without a slot number and store them sorted according to their right endpoints (left endpoints) when sweeping in left-to-right (! right-to-left) direction.

Remarks: In many applications, in particular in VLSI design, we may assume that $O(C) = O(\sqrt{n})$, if n is the number of input segments. But nothing is known about the minimal number m of slots required.

In most applications computing the slot assignment is just a preprocessing step for building a semidynamic skeleton structure of size $O(m)$ which will be used in a scanline algorithm for solving a more difficult problem. This, of course, pays off only if m is not too large; $m = O(\sqrt{n})$ is certainly acceptable but $m = \Omega(n)$ is not. Furthermore, the time required to compute the slot assignment by an algorithm like the algorithm sketched above may be not acceptable.

If we assume that we have sequential access but are not allowed to carry out back and forth sweeps we have to rearrange the segments after each scan such that the input for the next scan is again sorted with respect to increasing x-values on external store. Assuming that this is possible efficiently using not too much internal space we can obviously solve the MSAP by carrying out m sweeps in only one direction.

We state it as an open problem whether the rearrangement of the input segments after each sweep of the algorithm which carries out multiple sweeps in one direction can be avoided. A bit more generally stated, the question is the following: Assume that n input segments are stored externally according to increasing x-values of their left endpoints and that m is the minimal slot number required for the given input. Is it possible to compute a slot assignment using $O(C)$ space (C the cross section number) and time $O(n \cdot m \cdot \log C)$?

Algorithm 3: It is easy to see that we can avoid rearranging the segments after the i-th sweep in order to restore the correct x-order of segments, if we allow a redundant storage of the segments and carry out back and forth sweeps: Assume that each segment appears twice in the ordered input sequence on external store, first at the position of its left endpoint and second at the position of its right endpoint. For assigning slot numbers carry out pairs of left-to-right and right-to-left sweeps. During the i-th pair of sweeps the input is first scanned from left-to-right and slot number i is assigned to a segment when it becomes inactive. That is, slot number i is attached to a segment when the scanline has reached the right endpoint of the segment and no unnumbered segment above it has been detected during the time when the segment was active. Because we have to maintain only the topmost currently active, nonnumbered segments during the left-to-right scan, time at most $O(n)$ and internal space $O(1)$ is sufficient for the i-th left-to-right scan.

After completion of the i-th left-to-right scan the input is scanned again from right-to-left (in decreasing x-order). During this sweep we meet first the endpoints of segments which have just obtained (by the preceeding left-to-right scan) slot number i and, therefore, we can attach the same number i to the respective starting points of segments. (Observe that segments obtaining slot number i do not overlap; hence, the right-to-left scan can be carried out using only constant internal space!) After completion of the i-th right-to-left scan we are ready for the next pair of sweeps without any rearrangement of segments.

The time complexity of this algorithm is $O(m \cdot n)$.

Algorithm 4: In [OW86] an algorithm has been presented which computes the minimal slot assignment in optimal time $O(n \log n)$ and internal space $O(n)$. The algorithm presupposes that the input segments are sorted and stored externally according to decreasing y-values and can be read in this order from external store into internal memory. We will show in this paper that the space requirement of any such algorithm is $\Omega(n)$ even if the algorithm knows the number n of segments in advance and may read the input segments in any order from external store.

The above observations show that there is a tradeoff between time, space, input access mode, and output production when solving the slot assignment problem. Because we do not take the external space requirement into account, we have expressed the time and space complexity of the algorithms discussed so far in terms of n, the number of input segments, and not in terms of the total length of their description. Similarly, the internal space requirement ignores the fact that storing a single segment in core from a set of n segments requires $\Omega(\log n)$ space. Taking these facts into account we can summarize the characteristics of the discussed algorithms for computing a minimal slot assignment by the following table:

	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
number of sweeps	m	m	$2m$	1
external storage order	x-order of left endpoints	x-order of left/ right end-points	x-order of left and right end-points	y-order
external access mode	direct	sequential	sequential	sequential
external space	n	$2n$	$2n$	n
internal space	$O(1)$	$O(C)$	$O(1)$	$O(n)$
time	$O(mn)$	$O(mn \log C)$	$O(mn)$	$O(n \log n)$

The ultimate goal may be to find an algorithm which inspects each segment at most a constant number of times and uses space $O(C)$ and time $O(n \log n)$. The time bound $O(n \log n)$ corresponds to the time bounds obtained for optimal offline algorithms used in the analysis of competitive online algorithms. Unfortunately, we were unable to achieve this goal.

It should be noted once again that we have tacitly assumed that an input segment can be stored using constant space. This “unit-cost-model” is the usual model in most computational geometry algorithms, but it is not reasonable, if n is really large. Then we need $\Omega(\log n)$ space for each segment in order to store it either externally or in core. Thus, we have to multiply the space requirements for all algorithms discussed in this section by a factor of $\log n$.

Our lower bound proofs in the next sections will be based on similar assumptions. That is, we determine lower bounds for the space requirement of algorithms in terms of n, the number of inputs, and not in terms of the total length of input descriptions.

3 The computational model

Now we investigate the minimal amount of internal space required for a certain class of algorithms in order to calculate a minimal slot assignment for a given set of n horizontal line segments in the plane. We want to derive an $\Omega(n)$ internal space bound for every algorithm of the class which is able to compute a minimal slot assignment for the given set of n segments. This requires to formalize the class of algorithms, the input and output format, and the access mode.

Hyp 1 *The input is a set of n segments, stored off-line in a separate and addressible memory, which we will call external memory. We assume that the addresses are natural numbers, that every segment is stored in one address and that the n segments fill up the first n addresses whereas any remaining addresses are left empty.*

Hyp 2 *The algorithm is allowed to know the number n of segments in advance (either from a pre-calculation or from an additional input to the internal memory).*

Hyp 3 *The algorithm reads every input segment from the external memory exactly once. (A simple consideration shows that it has to read every input segment at least once.)*

Hyp 4 *There is no restriction in producing the output. More precisely this means: Between any two consecutive readings of input segments the algorithm calculates the slot-assignment of an arbitrary number of segments (may be none at all) and writes the result as output to external memory.*

If m is the minimal number of slots required we will not assume that the algorithm must use exactly the slot numbers $1, \ldots, m$. Assigning any slots from a totally ordered universe is sufficient, under the assumption that we do not use extremely large numbers to circumvent the problem. All we require is that the number of different slots assigned by the algorithm is minimal. The slot names can then be normalized to the integers $1, \ldots, m$ in a postprocessing step which is not considered to be a part of the algorithm solving the slot assignment problem.

Our assumptions Hyp 1, 2, 3, 4 imply that the storage for the input and output data itself is not taken into account; only the space needed for data held or produced during the computation of the output is accounted for.

First we give precise definitions of some concepts already used in previous sections. We assume that all start- and endpoints of segments belong to a discrete countable universe which is, for the sake of simplicity, the set of integers. Furthermore, we assume that each segment occurs at most once in any input sequence. Moreover, we suppose that segments are pairwise disjoint.

Definition 1 *(i) A segment is a triple* $\sigma = (x_1, x_2, y)$ *of integers such that* $x_1 \leq x_2$.
(ii) $\Sigma = \{\sigma \mid \sigma \text{ is a segment}\}$ *is called the input alphabet. (Every input sequence stored externally is encoded as a word of* Σ^**. However, not all words in* Σ^* *are considered to be legal input sequences.) Furthermore,* Σ^n *denotes the subset of* Σ^* *that consists of words of length* n *and the* i*-th letter of* $S \in \Sigma^n (i \leq n)$ *is denoted by* $S(i)$. $S(i)$ *is undefined, if* $i > |S|$*, where* $|S|$ *denotes the length of the word* S*. An input sequence* $S \in \Sigma^*$ *is called* legal *if for all* $i, j, 1 \leq i < j \leq n$*, the segments* $S(i)$ *and* $S(j)$ *have no point in common. ¿From now on we will consider only legal input sequences and, therefore, omit the adjective "legal" when talking about input sequences.*
(iii) Let Δ *be a countable, totally ordered set. (The sets of (positive) integers or rational numbers are possible choices for* Δ*.)* Δ *is called the set of slots. Let* $\prec_\Delta$ *be the total linear order defined on* Δ. $\Gamma = \{(\sigma, \delta) \mid \sigma \text{ is a segment}, \delta \in \Delta\}$ *is called the output alphabet;* δ *will be called the slot assigned to* σ*, if* (σ, δ) *is a letter of the output.*

Definition 2 *A slot assignment belonging to an input sequence* $S \in \Sigma^*$ *is a map* $\alpha_S : \{\sigma \mid \sigma \text{ is a segment of } S\} \to \Delta$ *such that the following properties (i) and (ii) hold:*
(i) Consider for an arbitrary $x \in \mathbb{R}$ *the set* $\{\sigma = (x_1, x_2, y) \mid x_1 \leq x \leq x_2, \sigma \text{ segment of } S\}$ *of segments cut by the vertical line at* x*. Then the* y*-order of segments in this set along the vertical line is compatible with the total order* $\prec_\Delta$ *of slots assigned to the segments in the set; i. e. if* $\sigma = (x_1, x_2, y)$ *and* $\sigma' = (x'_1, x'_2, y')$ *are two segments from this set and* $y < y'$*, then* $\alpha_S(\sigma) \prec_\Delta \alpha_S(\sigma')$*. (Note that* $y = y'$ *is impossible, because we assume that any two input segments are pairwise disjoint.)*
(ii) The set $\{\alpha_S(\sigma) \mid \sigma \text{ is a segment of } S\} \subseteq \Delta$ *has minimal cardinality; its cardinality is called the minimal slot number and denoted by* m_S *or, simply, by* m*, if it is clear to which input sequence it belongs.*

Algorithms for solving the slot assignment problem will be identified with their sequences of computational steps carried out on the given input sequences. We consider only deterministic algorithms, that is the respective next computational step is determined completely by the state of the algorithm and the next input read from external store; no randomization is allowed. The states include everything that the algorithm can store internally during the computation. Therefore, the state set is not assumed to be fixed but may grow with the length of the input sequence. We will finally argue that the growth-rate must be at least exponentional in n if the algorithm is able to correctly compute a minimal slot assignment for all input sequences of length n. This, in turn, will require at least linear internal space used by the algorithm.

Note that the algorithm may know the length n of the input sequence in advance. Hence, the sequence of states may depend on the total length of the input sequence and not only on the inputs seen until a certain stage.

In order to formalize our intuition we start with

Definition 3 *An algorithm of class $\mathcal{C}$ is a collection of quadruples ($q_0 \in Q, \nu : Q \to I\!N, \delta : \Sigma \times Q \to Q, \lambda : Q \to \Gamma^*$), one quadruple for every set Σ^n of input words of length n. (Note that for reasons of simplicity our notation does not distinguish the different quadruples of the collection depending on the different values of n.) Furthermore:*

(i) The elements of the set Q are called the (internal) states of the algorithm. The element q_0 is the distinguished initial state of the algorithm.

(ii) The map ν is called the address map and $\nu(q)$ is the external address of the segment that the algorithm will read when being in the state q.

(iii) The map δ is called transition function of the algorithm. This means that the algorithm being in the state q and reading the letter σ from the input will change to the state $\delta(\sigma, q)$.

(iv) The map λ is called output function of the algorithm, meaning that it will write the word $\lambda(q)$ on the output tape when being in the state q. (Recall that according to Hyp 4 the algorithm may produce an arbitrary word over the output alphabet, not only just a single letter.)

It is useful to extend the domain of the maps ν, δ, λ from the input alphabet Σ to the set of all input strings Σ^*:

Definition 4 *For every quadruple ($q_0 \in Q, \nu : Q \to I\!N, \delta : \Sigma \times Q \to Q, \lambda : Q \to \Gamma^*$) we define the induced quadruple ($q_0 \in Q, \nu^* : I\!N \times \Sigma^* \to I\!N, \delta^* :$*

$I\!N_0 \times \Sigma^* \to Q, \lambda^* : I\!N_0 \times \Sigma^* \to \Gamma^*)$ *recursively as follows:*

$$\delta^*(0, S) := q_0,$$

$$\delta^*(r+1, S) := \delta(S(\nu(\delta^*(r, S))), \delta^*(r, S)),$$
$$\nu^*(r+1, S) := \nu(\delta^*(r, S)),$$
$$\lambda^*(r, S) := \lambda(\delta^*(r, S)).$$

The maps ν^*, δ^* and λ^* are called the induced maps. $\nu^*(1, S), \nu^*(2, S), \ldots$ is the access-order whereas $S(1), \ldots, S(|S|)$ is the external storage order of the segments in input sequence S.

Finally we are able to formalize the class $\mathcal{C}_1$ of algorithms to be considered in this section.

Definition 5 *An algorithm of class $\mathcal{C}$ is an algorithm of class $\mathcal{C}_1$ if for every (legal) $S \in \Sigma^*$ the induced map ν^* is a bijection from $\{1, \ldots, |S|\} \times \{S\}$ onto $\{1, \ldots, |S|\}$ (with $|S|$ being the number of segments of S).*

This means that each segment of an input sequence S is read exactly once from external store. We do not require that the access order coincides with the storage order of segments on external memory. Reading segments in storage order is a special case which can be subsumed as follows into the above framework:

Choose $Q = I\!N_0 \times Q'$ for some Q'; define $\nu(q) = \nu((i, q')) = i + 1$ for each $q = (i, q') \in Q, i \in I\!N_0$.

It is obvious that the class $\mathcal{C}_1$ is precisely the formalization of the class of algorithms given intuitively by the hypothesis Hyp 1,2,3,4 above. We are now able to define precisely what it means that an algorithm solves the slot assignment problem for a certain class of inputs. We will consider only reasonable sets of inputs where the space requirement to store the segments is determined by the number of segments and not by a pathological choice of their coordinates. (Choosing the coordinate values of the n segments so large that it is impossible to denote them with numbers of length $O(\log n)$ is an example of a pathological choice.) Recall that we assume the unit cost model which says that external or internal space $O(n)$ is sufficient to store n segments. No further restriction on the input sets is imposed. That is, an input set is just an arbitrary subset of Σ^*.

Definition 6 *An algorithm of class $\mathcal{C}$ is called a slot-assignment algorithm* $:\Leftrightarrow$
For every input sequence $S \in \Sigma^*$ the relation given by the set of all pairs (σ, i) being letters of the output word $\lambda^*(1, S) \ldots \lambda^*(n, S)$, $n = |S|$, defines a (minimal) slot-assignment belonging to S.

Let $\mathcal{I}_y$ ($\mathcal{I}_{y,M}$, resp.) denote the class of all input sets with all their segments being sorted with respect to the y-coordinate (and with $m_S \leq M$ for every input sequence S, resp.) In this section we are interested in algorithms which are able to solve the slot-assignment problem for input sets belonging to one of these two classes. That is, we restrict ourselves to input sequences where the external storage order coincides with the y-order of segments. That is, for all i, the y-value of segment $S(i) \leq$ the y -value of segment $S(i+1)$. Note that we do not assume that the external storage order coincides with the access order. The latter one is determined by the function ν^* of an algorithm under consideration.

We use the following abbreviation:

$$\nu^{*s}_{r}(S) := \{\nu^*(t, S) \mid t = r+1, \ldots, s\}$$

for all $0 \leq r \leq s \leq |S|$. If $s = r$ then $\nu^{*s}_{r}(S) = \emptyset$. That is $\nu^{*s}_{r}(S)$ is the set of external addresses of the segments that the algorithm will read after having read r segments and until having read s segments or equivalently after having reached the r-th state and until reaching the s-th state.

Notice also that $|\nu^{*s}_{r}(S)| = s - r$, because each segment is read exactly once.

4 A first lower bound

As immediate consequences of the definitions given in the previous section we provide the following two lemmata, needed as tools in order to prove Theorem 1.

Lemma 1 *Let $S, S' \in \Sigma^n$ be two inputs of length n, let $a, b, a', b' \in \mathbb{N}_0$, $a \leq b \leq n$, $a' \leq b' \leq n$ and let be given an algorithm of class $\mathcal{C}$ such that*

(i) $\delta^*(a, S) = \delta^*(a', S')$

(ii) For all $k \in {\nu^*}_a^b(S) \cap {\nu^*}_{a'}^{b'}(S')$ it is true that $S(k) = S'(k)$.

Then it holds that

(i) $\nu^*(a+t, S) = \nu^*(a'+t, S')$ for all $1 \leq t \leq |{\nu^*}_a^b(S)| + 1 = b - a + 1$ and if both sides are defined.

(ii) $\delta^*(a+t, S) = \delta^*(a'+t, S')$ for all $0 \leq t \leq |{\nu^*}_a^b(S)|$ and if both sides are defined.

(This means that the algorithm, after having read r segments, only depends on the state that it has reached and on the segments that it will read afterwards.)

Proof: Using definition 4 we prove by induction on t:
$t = 0$: $\delta^*(a+t, S) = \delta^*(a'+t, S')$ due to assumption (i).
$t \to t+1 : \nu^*(a+t+1, S) = \nu(\delta^*(a+t, S)) = \nu(\delta^*(a'+t, S')) = \nu^*(a'+t+1, S')$ (due to the induction hypothesis)
$\Rightarrow \nu^*(a+t+1, S) \in {\nu^*}_a^b(S) \cap {\nu^*}_{a'}^{b'}(S') \quad \forall\ 0 \leq t \leq b - a - 1$
$\Rightarrow S(\nu^*(a+t+1, S)) = S'(\nu^*(a'+t+1, S')) \quad \forall\ 0 \leq t \leq b - a - 1$ (due to assumption (ii))
$\Rightarrow \delta^*(a+t+1, S) = \delta(S(\nu^*(a+t+1, S)), \delta^*(a+t, S)) = \delta(S'(\nu^*(a'+t+1, S')), \delta^*(a'+t, S')) = \delta^*(a'+t+1, S') \quad \forall\ 0 \leq t \leq b - a - 1$ (due to the induction hypothesis)

$\square$

As a corollary of Lemma 1 we conclude that the states of the algorithm after having read exactly r segments from the input strings S and S', resp. must be different if the following holds:

- All the addresses that are left to be read in both input strings S and S' contain the same segment each (i).
- Additionally, if all these addresses of S and S' are equal the further output of the algorithm is different (ii).

We state this more precisely in Lemma 2:

Lemma 2 *Let $S, S' \in \Sigma^n$ be two inputs of length n, let $r \in \mathbb{N}_0$, $r \leq n$, and let be given an algorithm of class $\mathcal{C}_1$ such that* *(i) For all $k \in \nu^{*n}_{r}(S) \cap \nu^{*n}_{r}(S')$ it holds that $S(k) = S'(k)$.*

(ii) Additionally, if $\nu^{*n}_{r}(S) = \nu^{*n}_{r}(S')$ then the set of letters in the output word $\lambda^*(r+1, S) \ldots \lambda^*(n, S)$ is not equal to the set of letters in the output word $\lambda^*(r+1, S') \ldots \lambda^*(n, S')$. (Remind that the $\lambda^*(r, S)$ are words over the alphabet Γ itself, rather than letters.)

Then it holds that

$$\delta^*(r, S) \neq \delta^*(r, S').$$

Proof: Let $\delta^*(r, S) = \delta^*(r, S')$. Using assumption (i), Lemma 1 and Definition 4 we conclude that $\nu^{*n}_{r}(S) = \nu^{*n}_{r}(S')$ and $\lambda^*(r + t, S) = \lambda(\delta^*(r + t, S)) = \lambda(\delta^*(r + t, S')) = \lambda^*(r + t, S') \quad \forall\, 0 \leq t \leq n - r$.
However, this contradicts assumption (ii).

□

Now we are ready to prove the first lower space bound for algorithms of class $\mathcal{C}_1$.

Theorem 1 *Every slot-assignment algorithm of class $\mathcal{C}_1$ requires $\Omega(n)$ (n being the number of segments) internal space. This remains true even if the set of inputs is in $\mathcal{I}_y$ or $\mathcal{I}_{y,M}$ (with $M \geq 2$), resp.*

Proof: Obviously it is enough to prove the assertion for the smallest class of input sets $\mathcal{I}_{y,M}$ (with $M \geq 2$).
We define for every $n \in I\!N$:

$$\tilde{\Sigma^n} := \{S \in \Sigma^* \mid \; |S| = n \text{ and either } S(i) = (4i-4, 4i-3, -i) \text{ or } S(i) = (4i-2, 4i-1, -i) \text{ for every } i = 1, \ldots, n\}$$

This means the following: Consider the set Σ_2 of $2n$ segments shown in Figure 1 which has exactly two segments of length 1 at each y-value. Then $\tilde{\Sigma^n}$ is obtained from Σ_2 by choosing exactly one of the two segments with y-value i, $-1 \geq i \geq -n$.

It is easy to see that $\bigcup_{n=1}^{\infty} \tilde{\Sigma^n}$ is a subset of every input set of class $\mathcal{I}_{y,M}$ (with $M \geq 1$).

Now we assume that the algorithm has already read exactly $l \in \{1, \ldots, n-1\}$ segments (=letters) from the input string (not necessarily in storage-order!).

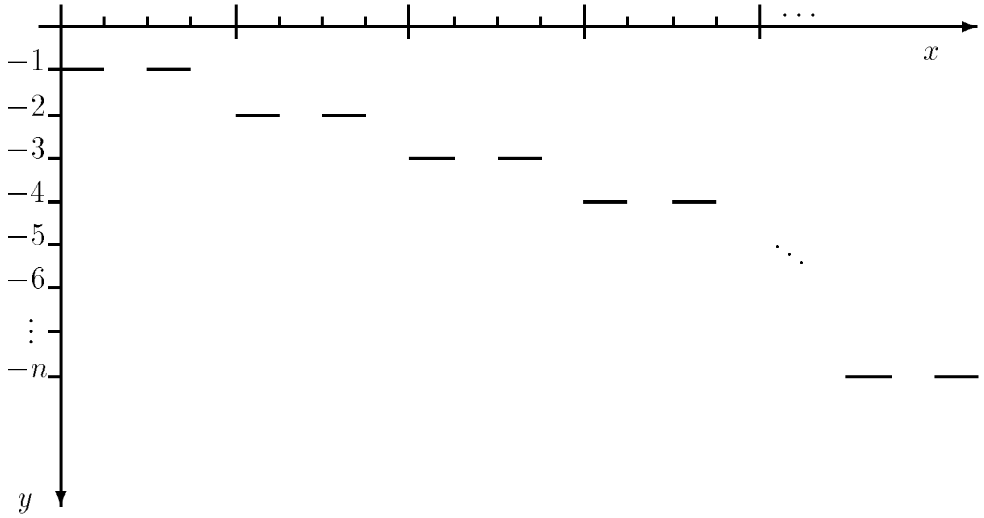

Figure 1:

Lemma 1 shows with $a = a' = 0$ and $b = b' = l$ that we may modify every input string S without changing the state $\delta^*(l, S)$ if we let all segments $S(k)$ with $k \in \nu^{*l}_{\mathbf{0}}(S)$ unchanged (i.e. if we only change segments that the algorithm has not read yet).
Thus, for every pair of input strings $S, T \in \tilde{\Sigma}^n$, $S \neq T$ we will construct a modified pair $S_{mod}, T_{mod} \in \Sigma^n$ in such a way that Lemma 2 yields

$$\delta^*(l, S_{mod}) \neq \delta^*(l, T_{mod})$$

for at least $2^{\mathbf{\Omega}(n)}$ pairs $S, T \in \tilde{\Sigma}^n$. (Note that both S_{mod} and T_{mod} may depend on the pair (S, T), i.e. T_{mod} may change if S changes for example.)
Since $\delta^*(l, S_{mod}) = \delta^*(l, S)$ and $\delta^*(l, T_{mod}) = \delta^*(l, T)$ by construction, we conclude that

$$\delta^*(l, S) \neq \delta^*(l, T).$$

Thus we have found at least $2^{\mathbf{\Omega}(n)}$ different states $\delta^*(l, S)$, $S \in \tilde{\Sigma}^n$, which means that the internal space is at least $\log 2^{\mathbf{\Omega}(n)} \in \Omega(n)$.

Now, after having outlined the main idea of the rest of the proof we will present it in detail.
At first we give the precise definition of the modified pair $S_{mod}, T_{mod} \in \Sigma^n$ assigned to every pair $S, T \in \tilde{\Sigma}^n$. Thereby, we use the fact that for every

fixed $S \in \Sigma^*$ the map $r \mapsto \nu^*(r, S)$ is a bijection (see Definition 5). So the inverse map exists and we will denote it by ν_S^{-1} (i.e. the i-th segment of the input string S is the $\nu_S^{-1}(i)$-th one that the algorithm reads.)
Let $S, T \in \tilde{\Sigma^n}$. Then we define:

$$S_{mod}(i) := \begin{cases} S(i) & \text{if} \quad i \in {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S) \\ (4k(i)-4, 4k(i)-3, -i) & \text{if} \quad i \in {\nu^*}_{\lfloor n/2 \rfloor}^{n}(S) \text{ and } S(k(i)) = \\ & \qquad (4k(i)-2, 4k(i)-1, -k(i)) \\ (4k(i)-2, 4k(i)-1, -i) & \text{if} \quad i \in {\nu^*}_{\lfloor n/2 \rfloor}^{n}(S) \text{ and } S(k(i)) = \\ & \qquad (4k(i)-4, 4k(i)-3, -k(i)) \end{cases}$$

with the abbreviation $k(i) := \nu^*(\nu_S^{-1}(i) - \lfloor n/2 \rfloor, S)$, i.e. $k(i)$ is the external address of the segment that the algorithm has read $\lfloor n/2 \rfloor$ steps before the segment with address i.

$$T_{mod}(i) := \begin{cases} S_{mod}(i) & \text{if } i \in {\nu^*}_{\lfloor n/2 \rfloor}^{n}(S) \cap {\nu^*}_{\lfloor n/2 \rfloor}^{n}(T) \\ T(i) & \text{otherwise} \end{cases}$$

Notice that the modified input strings are still elements of the same input set in the class $\mathcal{I}_{y,2}$.

Now, let ${\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S) = {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(T)$ and $S(j) \neq T(j)$ for some $j \in {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S)$. Then, since $k^{-1}(j) \in {\nu^*}_{\lfloor n/2 \rfloor}^{n}(S) \cap {\nu^*}_{\lfloor n/2 \rfloor}^{n}(T)$ we see from the definition that

$$T_{mod}(k^{-1}(j)) = S_{mod}(k^{-1}(j)) =$$

$$= \begin{cases} (4j-4, 4j-3, -(k^{-1}(j))) & \text{if } S(j) = (4j-2, 4j-1, -j) \\ (4j-2, 4j-1, -(k^{-1}(j))) & \text{if } S(j) = (4j-4, 4j-3, -j). \end{cases}$$

Since at the same time we know that

$$T_{mod}(j) = T(j) \neq S(j)$$

we conclude that

$$T_{mod}(k^{-1}(j)) = S_{mod}(k^{-1}(j)) =$$

$$= \begin{cases} (4j-4, 4j-3, -(k^{-1}(j))) & \text{if } T(j) = (4j-4, 4j-3, -j) \\ (4j-2, 4j-1, -(k^{-1}(j))) & \text{if } T(j) = (4j-2, 4j-1, -j), \end{cases}$$

(i.e. the endpoints of $T_{mod}(k^{-1}(j))$ and $T_{mod}(j)$ have the same x-coordinates but have different y-coordinates).
This implies that

$$| \{ \alpha_{T_{mod}}(T_{mod}(k^{-1}(j))), \alpha_{T_{mod}}(T_{mod}(j)) \} | = 2$$

On the other hand the definition of S_{mod} always guarantees that

$$\alpha_{S_{mod}}(S_{mod}(k^{-1}(j))) = \alpha_{S_{mod}}(S_{mod}(j))$$

That is, any correct slot assignment for T_{mod} requires two different slots while one slot is sufficient for S_{mod}.

If $k^{-1}(j) \geq j$ it is obvious that the algorithm cannot calculate the slot-assignments $\alpha_{T_{mod}}(T_{mod}(k^{-1}(j)))$ and $\alpha_{S_{mod}}(S_{mod}(k^{-1}(j)))$, resp. before reading $T_{mod}(k^{-1}(j))$ and $S_{mod}(k^{-1}(j))$, resp. from the input.
Therefore $(T_{mod}(k^{-1}(j)), \alpha_{T_{mod}}(T_{mod}(k^{-1}(j))))$ and $(S_{mod}(k^{-1}(j)), \alpha_{S_{mod}}(S_{mod}(k^{-1}(j))))$, resp. are letters of $\lambda^*(\lfloor n/2 \rfloor + 1, T_{mod}) \ldots \lambda^*(n, T_{mod})$ and $\lambda^*(\lfloor n/2 \rfloor + 1, S_{mod}) \ldots \lambda^*(n, S_{mod})$, resp.
If $k^{-1}(j) \leq j$, a similar consideration shows that $(\alpha_{T_{mod}}(T_{mod}(j)), j)$ and $(\alpha_{S_{mod}}(S_{mod}(j)), j)$, resp. are letters of $\lambda^*(\lfloor n/2 \rfloor + 1, T_{mod}) \ldots \lambda^*(n, T_{mod})$ and $\lambda^*(\lfloor n/2 \rfloor + 1, S_{mod}) \ldots \lambda^*(n, S_{mod})$, resp. as well.
Thus, the two input strings $S_{mod}, T_{mod} \in \Sigma^n$ together with $r = \lfloor n/2 \rfloor$ fulfill the assumptions of Lemma 2 and we may conclude that

$$\delta^*(\lfloor n/2 \rfloor, S_{mod}) \neq \delta^*(\lfloor n/2 \rfloor, T_{mod}).$$

Summing up what we have proved so far we get that

$$\delta^*(\lfloor n/2 \rfloor, S) = \delta^*(\lfloor n/2 \rfloor, S_{mod}) \neq \delta^*(\lfloor n/2 \rfloor, T_{mod}) = \delta^*(\lfloor n/2 \rfloor, T)$$

if ${\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S) \neq {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(T)$ or if ${\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S) = {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(T)$ and $S(j) \neq T(j)$ for some $j \in {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S)$.
Proceeding further this means precisely that the equivalence classes of the equivalence relation $\sim_1$ on $\tilde{\Sigma^n}$

$$S \sim_1 T :\Leftrightarrow \delta^*(\lfloor n/2 \rfloor, S) = \delta^*(\lfloor n/2 \rfloor, T)$$

are all subsets of the equivalence classes of the equivalence relation $\sim_2$ on $\tilde{\Sigma^n}$

$$S \sim_2 T :\Leftrightarrow {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S) = {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(T) \text{ and } S(j) = T(j) \text{ for all } j \in {\nu^*}_{\mathbf{0}}^{\lfloor n/2 \rfloor}(S).$$

It is obvious from the definition of $\tilde{\Sigma^n}$ that the cardinality of every equivalence class of the equivalence relation $\sim_2$ is at most $2^{\lceil n/2 \rceil}$. Since $|\tilde{\Sigma^n}| = 2^n$ there are at least $2^n/2^{\lceil n/2 \rceil} = 2^{\lfloor n/2 \rfloor}$ equivalence classes of the equivalence relation $\sim_2$ and therefore at least $2^{\lfloor n/2 \rfloor}$ equivalence classes of the equivalence relation

$\sim_1$ as well. But this means that there are at least $2^{\lfloor n/2 \rfloor}$ different states $\delta^*(\lfloor n/2 \rfloor, S)$, $S \in \tilde{\Sigma^n} \subset \Sigma^n$, of the algorithm.
So the internal space cannot be less than $\Omega(\log 2^{\lfloor n/2 \rfloor}) = \Omega(n)$.

$\square$

Example: To clarify the idea of the proof we give an example for the case $n = 8$. Hence, let us assume that we have two input sequences S and T in $\tilde{\Sigma^8}$ with the property that they differ in one of the first 4 segments being read by a given algorithm which is able to compute a correct slot assignment for each input sequence of length 8 belonging to the class $\mathcal{I}_{y,2}$. Input sequences in this class have start and endpoints in the set $\{i \mid 0 \leq i \leq 4 \cdot 8\}$; the segments are stored externally according to decreasing y-order and they can consistently be numbered using at most two different slots.

Figure 2 and 3 show instances of S and T where the numbers attached to the segments reflect the access order of the segments by the given algorithm, i. e. $\nu^*(i)$ is the external address of the i-th segment accessed by the algorithm. (Note that S and T differ at the 4-th segment being read into internal store.)

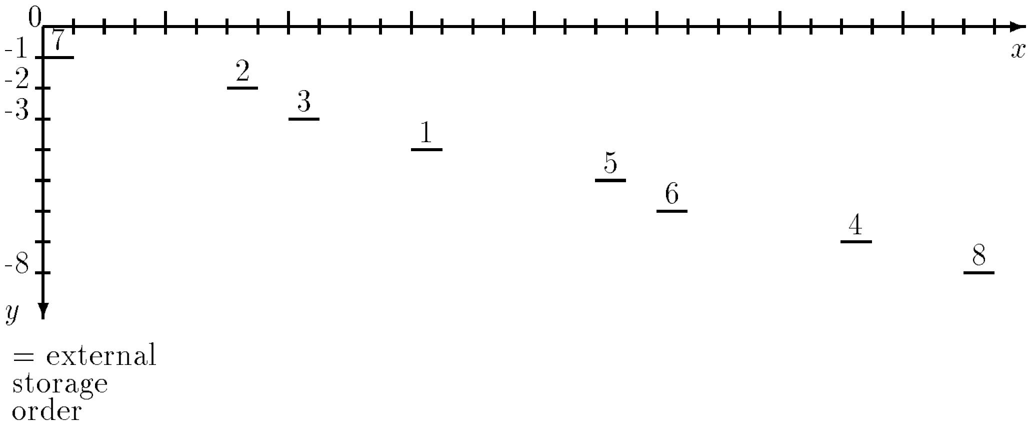

Figure 2:

Figure 4 and 5 show the input sequences S_{mod} and T_{mod}. Note that S_{mod} and T_{mod} coincide with S and T, respectively, for the first 4 segments being read from external store into core. S_{mod} and T_{mod} are identical for the remaining 4 segments. The access order of the last 4 segments is unknown. But we know that the given algorithm having read the first 4 segments of

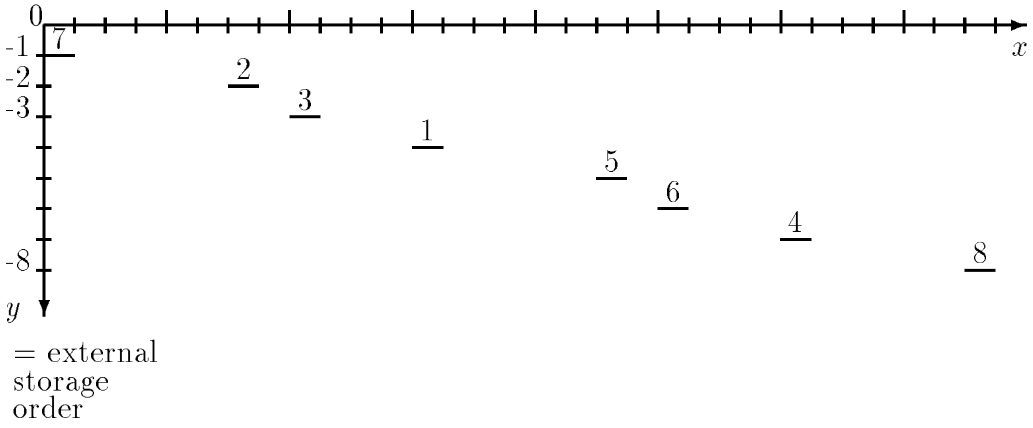

Figure 3:

S_{mod} and T_{mod} must have reached the same state which is reached after it has read the first 4 segments of S and T, respectively.

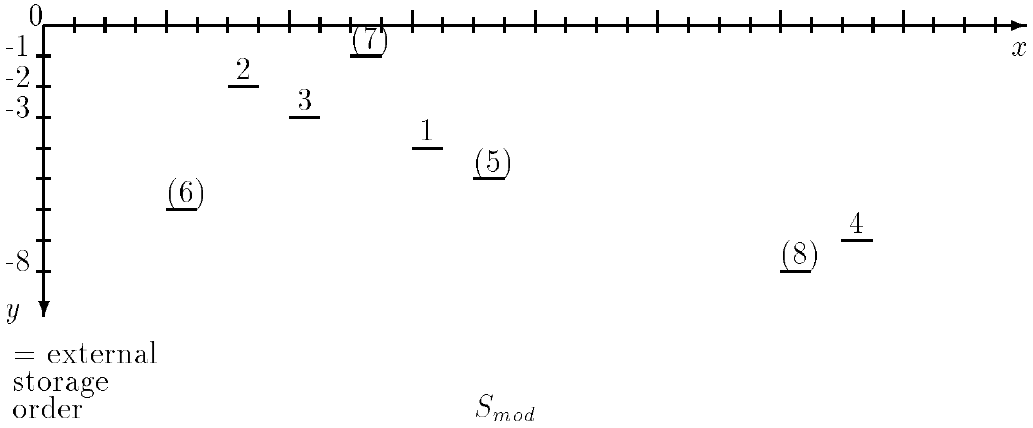

Figure 4:

Note that the state of the algorithm may depend only on the total length of an input sequence and on the segments it has read so far. S_{mod} and T_{mod} have been chosen such that they require a different slot assignment: One slot suffices to store the elements of S_{mod}, but T_{mod} requires two slots.

Therefore, the states which the algorithm has reached after having read the first 4 segments of S_{mod} and T_{mod} must be different. For otherwise it would produce the same output for any access sequence of the last 4 segments, which is incorrect.

5 Left to right scan of x-ordered segments

Now, we will prove the same lower space bounds in the case when the input segments in the external memory are stored according to their x-coordinate. We may even assume that every segment (x_1, x_2, y) occurs twice in the input, namely one time it is stored at the position corresponding to its x_1-coordinate and the second time at the position given by its x_2-coordinate. Moreover, we only consider algorithms that read the external input storage exactly once from left to right (i. e. in increasing address- and x-order, resp.). More precisely, δ^* and λ^* are as before, however $\nu^*(r+1, S) := r+1$. Such algorithms are said to be of class $\mathcal{C}_{1,x}$. Since it is quite similar as before the proof will be given only informally.

We consider the set of $2n+13$ segments $\Sigma_{2,x}$ in Figure 6.

Let $\tilde{\Sigma}_x^n$ be the set of all inputs which are obtained from the input $\Sigma_{2,x}$ by leaving out exactly one segment from each pair of segments in parentheses, one on top of the other.

Figure 7 shows an example of such an input in $\tilde{\Sigma}_x^n$.

We also assume that the sweep-line of our algorithm has already reached a position which is indicated by the dotted vertical line. This means that the algorithm has already read all start- and endpoints of the segments left of the sweep-line. For the 3 segments crossing that line only their left endpoints have been passed, and the 3 segments to the right of the sweep-line have not been detected at all.

Lemma 3 *At this stage the algorithm cannot have output the slot of any of the segments except for the six leftmost ones.*

Proof: The six leftmost segments require six different slots. In any slot assignment the number of different slots must be minimal. Therefore, all other segments which are not to the right of the sweep-line have to be labelled using the smallest five or the greatest five of these slots depending on whether segment σ_a or segment σ_b occurs in the input which is considered. Since at the considered stage of the algorithm σ_a or σ_b, respectively, is unknown, none of the segments except for the six leftmost ones can be assigned a slot.

$\square$

Thus Lemma 3 shows that the slots of all the $n - 12$ short segments which are to the left of the sweep-line except for the six leftmost ones must be encoded in the internal state $q \in Q$ that the algorithm has reached. Since these segments can be arranged in 2^{n-12} different ways leading to 2^{n-12} different slot assignments we know that there must be at least 2^{n-12} internal states, i. e. $|Q| \geq 2^{n-12}$. Thus we have proven the following theorem.

Theorem 2 *Every slot-assignment algorithm of class $\mathcal{C}_{1,x}$ working on input sets which contain $\bigcup_{n=1}^{\infty} \tilde{\Sigma}_x^n$ requires $\Omega(n)$ internal space.*

6 Conclusion

We have seen that it is impossible to beat the linear space bound obtained by the optimal algorithm presented in [OW86] for computing a minimal slot assignment when we assume that the input is stored externally in decreasing y-order and each input segment is read exactly once into core. Because a slot assignment for horizontal line segments will be used as the basis for the construction of a skeleton structure used in scanline algorithms with a vertical scanline, the assumption that the input segments are stored externally according to decreasing y-values is not very realistic. For, a scanline algorithm with a vertical scanline requires the inputs to be stored in sorted x-order. Therefore it would be more desirable to compute a minimal slot assignment for inputs of this kind. As we have seen, also in this case linear space is necessary if we assume that each segment is read exactly once (start- and endpoints in increasing x-order) into core from external memory. However there is still much room for other attempts: We may carry out a fixed number of left to right and/or right to left scans, we may augment the segments with additional information which may facilitate the computation of a minimal slot assignment, we may be satisfied with a small number of slots instead of the minimal number, finally we may be interested in the design of a randomized algorithm with a good expected behavior instead of a deterministic one yielding the optimal result. Hence, the idea of unifying and simplifying a scanline algorithm by first computing a (minimal) slot assignment, then constructing a semidynamic skeleton structure based on the slot numbering, and finally to carry out the scanline algorithm using the semidynamic structure may be still feasible. The results of this paper only show that the idea requires more than a straightforward approach.

References

[K91] R. Klein. Walking an unknown street with bounded detour. *32nd FOCS*, 304-313,1991

[MMS88] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for online problems *20th STOC*, 322-333, 1988

[MMS90] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems *Journal of Algorithms*, 11:208-230, 1990

[NP82] J. Nievergelt and F. P. Preparata. Plane–sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25:739-747, 1982

[OW86] Th. Ottmann and D. Wood. Space-Economical Plane-Sweep Algorithms. *Computer Vision, Graphics, and Image Processing*, 34(1): 35-51, 1986

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202-208, 1985

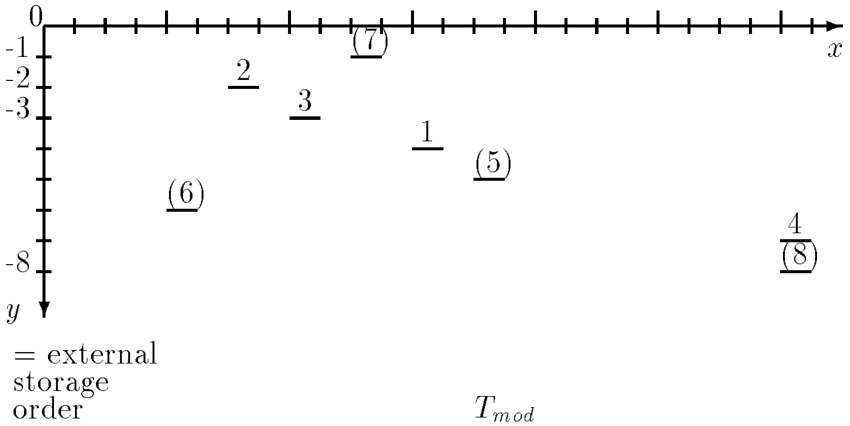

Figure 5:

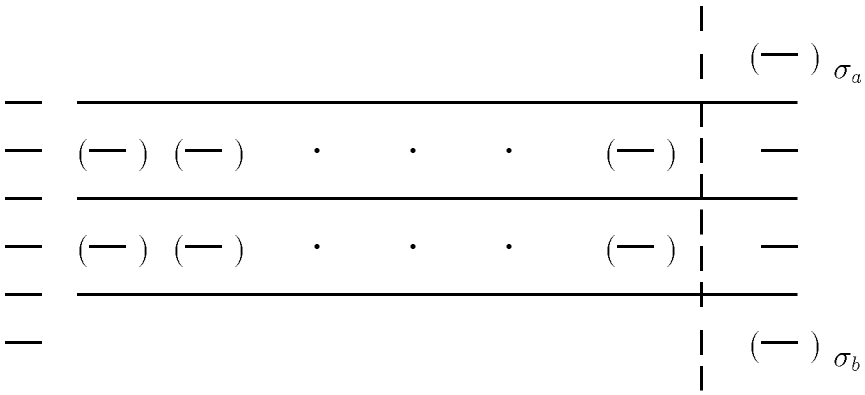

Figure 6:

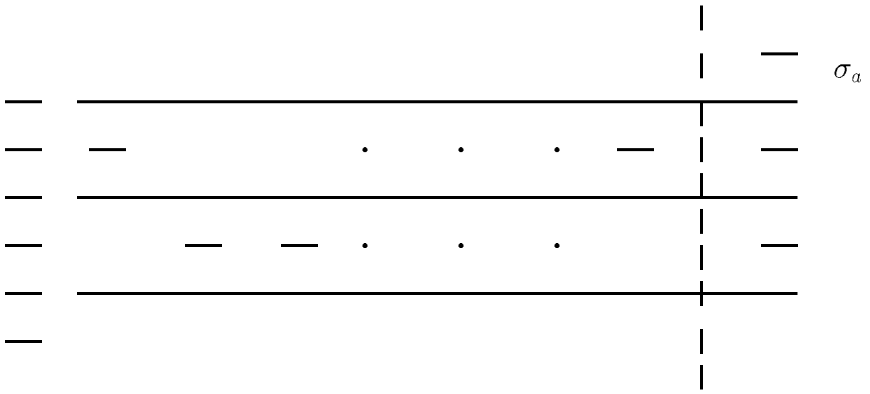

Figure 7: