Generating Random Monotone Polygons

Jack Snoeyink Chong Zhu

Technical Report 93-28
September 1993

Department of Computer Science
The University of British Columbia
Vancouver, B. C. V6T 1Z4
Canada

Abstract

We proposed an algorithm that generates x-monotone polygons for any given set of n points uniformly at random. The time complexity of our algorithm is $O(K)$, where $n \leq K \leq n^2$ is the number edges of the visibility graph of the x-monotone chain whose vertices are the given n points. The space complexity of our algorithm is $O(n)$.

1 Introduction

This paper details some recent results that we have obtained in our study of generating random polygons. In particular, we describe an algorithm for generating x-monotone polygons uniformly at random. The remainder of this section provides motivation for this research and a detailed description of this problem. In Section 2, we give the general notation and definitions of our algorithm. In Section 3, we present our monotone polygon generating algorithm with the counting procedure and generating procedures. In Section 4 we prove that our algorithm can generate monotone polygons uniformly at random. In Section 5, we give the visibility computing procedures with the correctness proofs. In Section 6 we analyze the time and space complexity of our algorithm. A summary of our results and related open problems are presented in Section 7.

1.1 Motivation

As well as being of theoretical interest, the generation of random geometric objects has applications which include the testing and verification of time complexity for computational geometry algorithms.

Algorithm Testing: The most direct use for a stream of geometric objects generated at random is for testing computational geometry algorithms. We can test such algorithms in two ways. The first involves the construction of geometric objects that the implementer considers difficult cases for the algorithm. For example, our polygon-nesting algorithm, based on a plane sweep, may require special case code for some polygons. It is important to make those polygons candidates for exposing errors of the algorithm. The second approach to testing involves executing the algorithm on a large set of geometric objects generated at random. We expect errors to be exposed if enough different valid inputs are applied to the algorithm.

Verification of Average Time Complexity: In implementation-oriented computational geometry research, we are often given the problem of verifying that an implementation of an algorithm achieves the stated algorithm time complexity. This is done by timing the execution of the algorithm for various inputs of different sizes. There are many possible inputs of any given size, and the choice is important, since an algorithm may take more time on some inputs than others of the same size. If an average execution time is computed over a set of randomly generated objects of a given size, the relationship between time and problem size will typically follow a curve corresponding to its complexity. We can then check this complexity against the stated algorithm's complexity.

Research has been done on generating geometric objects at random, such as Epstein [1]. This paper gives an algorithm to generate monotone polygons at random.

1.2 Problem

Let $S_n = \{s_1, s_2, ..., s_n\}$ be a set of n arbitrary points sorted according to their x coordinate. We want to generate a simple polygon defined by S_n at random. At this beginning stage we only consider generating a monotone polygon from S_n. Figure 1 shows a monotone polygon generated from a set of 12 points.

In [1] Epstein gives an $O(n^4)$ algorithm to generate triangulation of a given simple polygon at random. His algorithm, although not generating simple polygons at random, inspires us in constructing our algorithms for generating monotone polygons at random.

In Section 3, we will give an algorithm that generates a monotone polygon randomly on a set of n points in $O(K)$ time and in $O(n)$ space, where K is the total number of *above-visible* and *below-visible* points (see Section 2 for definitions) of the points in the point set.

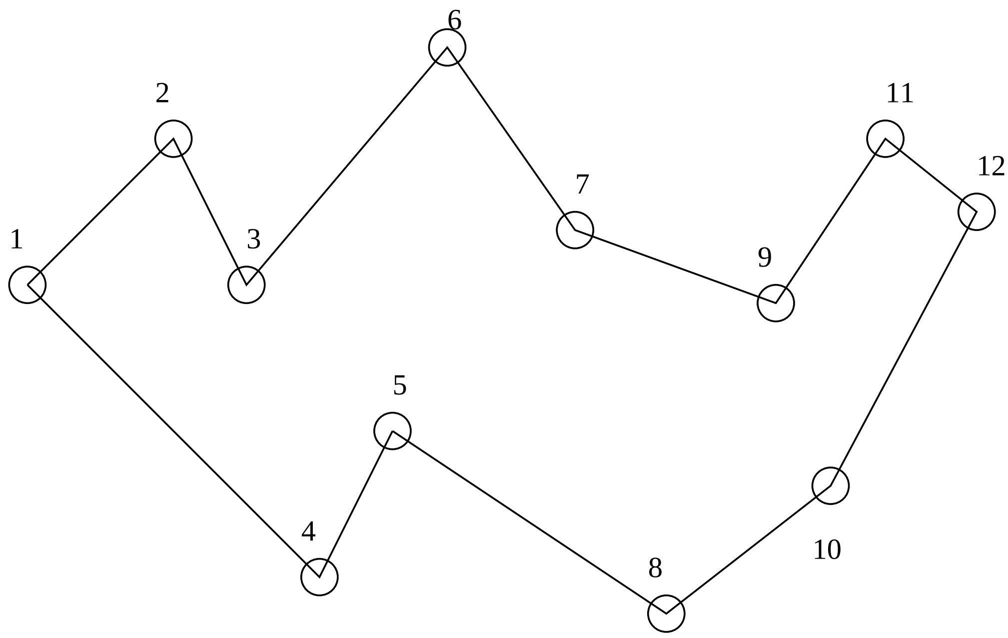


Figure 1: A monotone polygon generated from S_{12}

In related work, Meijer and Rappaport [4] study monotone traveling salesmen tours and show that the number of x-monotone polygons on n vertices is between $(2+\sqrt{5})^{(n-3)/2}$ and $(\sqrt{5})^{(n-2)}$. Mitchell and Sundaram [5] have independently developed a routine to generate random monotone polygons in $O(n)$ space and $O(n^2)$ time.

2 Preliminaries

Notation. We refer to a *probability space* as (Ω, E, P_r), where Ω is the *sample space*, E is the *event space*, and P_r is the *probability function*. The sample space Ω is the set of all *elementary events* that are the possible outcomes of the experiment being described. The event space E is the set of all subsets of Ω that are assigned a probability. The function $P_r : E \rightarrow \Re_0^+$ defines the probability of events.

A geometric object generator is an algorithm that produces a stream of geometric objects of a given type. We say that a generator is *complete* if it can produce every object in a given sample space Ω.

The Uniform Probability Distributions. Probability theory defines both *discrete* and *continuous* uniform probability distributions. We are interested only in the discrete case: the *discrete uniform probability space* for a finite sample space Ω_U is defined as (Ω_U, E_U, P_{rU}), where E_U is the set of all subsets of Ω_U, and $P_{rU}(A) = 1/|\Omega_U|$ for all $A \in \Omega_U$. In other words, in a finite sample space, a uniform distribution is one in which each elementary event is equally likely.

Since the sample space we deal with is finite, we use the discrete uniform probability distribution.

We define that a monotone polygon generator is *uniform* if each of the monotone polygons has the same probability of being generated.

Definitions. Let $S_n = \{s_1, s_2, ..., s_n\}$ be a set of n arbitrary points sorted according to their x coordinate. Let $S_i = \{s_1, s_2, ..., s_i\}$ be a subset of S_n for $1 \leq i \leq n$. The total number of monotone polygons can be generated with point set S_i is denoted as $N(i)$.

Any monotone polygon constructed from S_i can be divided into two monotone chains of which the leftmost vertex is s_1 and rightmost vertex is s_i. In Figure 2 the top monotone chain is $\{1, 2, 3, 6, 7, 11, 12\}$ and bottom monotone chain is $\{1, 4, 5, 8, 10, 12\}$. Any point in S_i is either on the top or bottom chain, except s_1 and s_i are on both chains because they are the beginning and ending points of the chains.

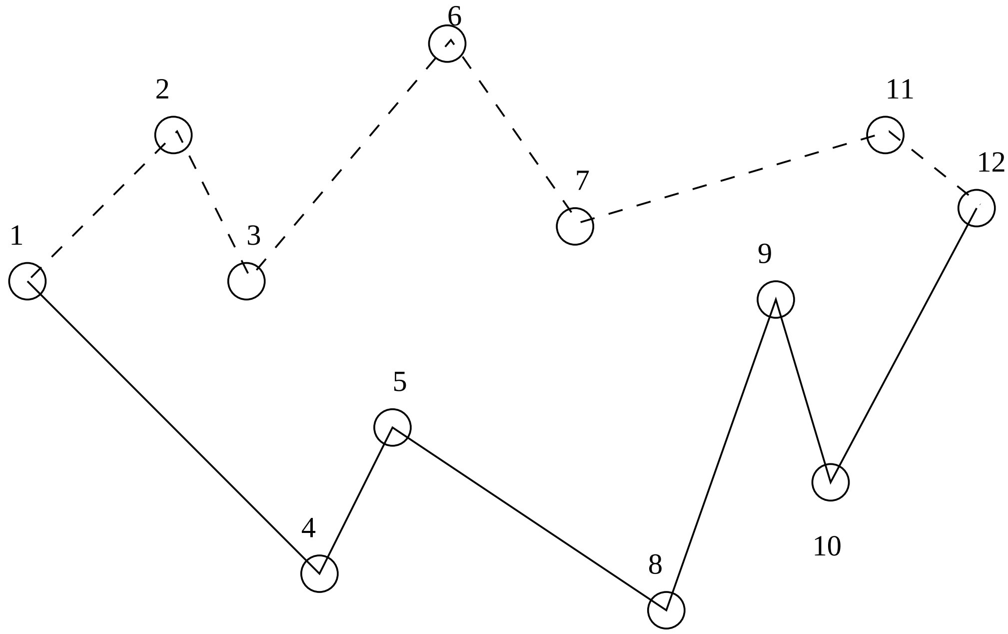


Figure 2: The top and bottom monotone chains

Let $T(i)$ be the set of monotone polygons that are generated from S_i with the edge $(i-1, i)$ on their top chains. Let $B(i)$ be the set of monotone polygons that are generated from S_i with the edge $(i-1, i)$ on their bottom chains. We define $TN(i) = |T(i)|$ to be the total number of the monotone polygons included in $T(i)$ and $BN(i) = |B(i)|$ to be the total number of the monotone polygons included in $B(i)$.

Let $l(j, i)$ be the line determined by s_j and s_i. Now we define *above-visible* or *below-visible* for a point. We say that a point s_k is *above-visible* from s_i if s_k is above all $l(j, i)$, for $j = i-1, ..., k-1$. And a point s_k is *below-visible* from s_i if s_k is below all $l(j, i)$, for $j = i-1, ..., k-1$. For example, in Figure 3, s_{10} is *above-visible* from s_{12}, and $\{s_9, s_7\}$ are *below-visible* from s_{12}.

Let $V_t(i)$ be the set of all the points that are *above-visible* from point s_i. Let $V_b(i)$ be the set of all the points that are *below-visible* from point s_i. For example, in Figure 3, the $V_t(12) = \{10\}$ and $V_b(12) = \{9, 7\}$. The number of points in $V_t(i)$ and $V_b(i)$ is denoted $|V_t(i)|$ and $|V_b(i)|$ respectively.

3 Generating Monotone Polygons at Random

We have two steps to generate monotone polygons randomly from S_n. The first one is to calculate the number of monotone polygons that can be generated from S_n. Then we scan S_n backward to generate monotone polygons.

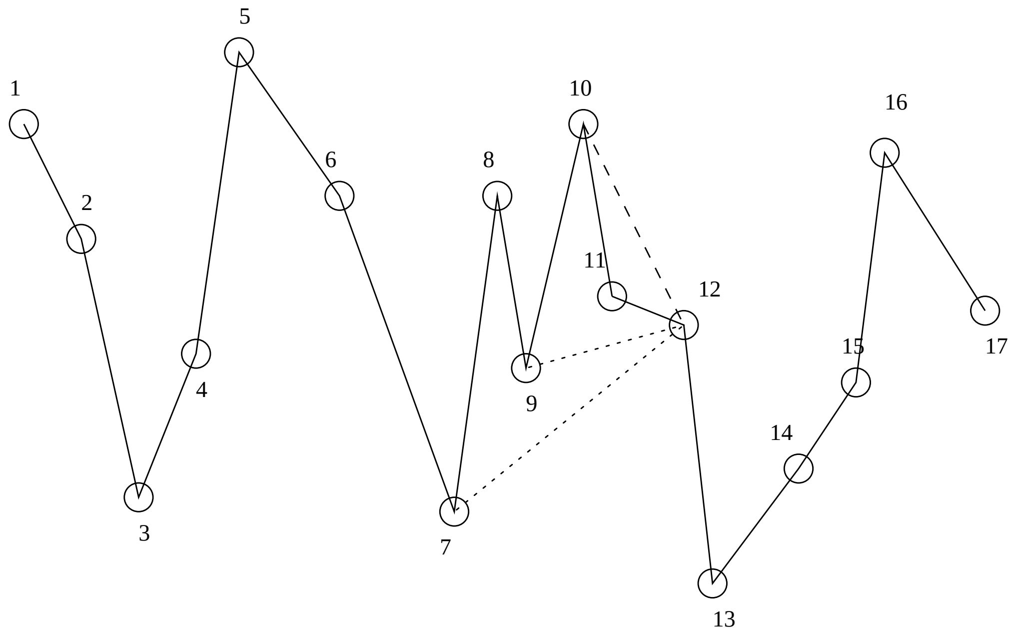


Figure 3: The *above-visible* and *below-visible* points from point $\{12\}$

3.1 Counting Monotone Polygons

Before we give the procedure to count monotone polygons we prove several theorems to build up the theoretical background.

Lemma 3.1 *The set of monotone polygons that are generated from S_k with edge $(k-1,k)$ on their top chains is disjoint from the set of monotone polygons that are generated from S_k with edge $(k-1,k)$ on their bottom chains. That is*

$$T(k)\bigcap B(k)=\emptyset.$$

Proof. Clearly, there is no polygon in $T(k)$ that could include edge $(k-1,k)$ in its bottom chain. And there is no polygon in $B(k)$ that could include edge $(k-1,k)$ in its top chain. So $T(k)\bigcap B(k)=\emptyset$. □

From this lemma we get the following result.

Theorem 3.2 *For any point set S_k, with $k>2$, the number of monotone polygons generated with point set S_k is*

$$N(k)=TN(k)+BN(k) \tag{1}$$

Proof. Let P be any monotone polygon that is generated from S_k. Then we know that the edge $(k-1,k)$ is either on the top chain of P which means $P\in T(k)$ or on the bottom chain of P which means $P\in B(k)$. In both cases P is counted by either $TN(k)$ or $BN(k)$. According to Lemma 3.1, we have $N(S_k)=TN(k)+BN(k)$. □

For any simple monotone polygon generated from S_k, its top chain and bottom chain are paths from s_1 to s_k. The edge $(k-1,k)$ is either on the top chain or on the bottom chain of the monotone polygon. For the chain that does not contain edge $(k-1,k)$, there exists a point s_j, $(j<k-1)$, that connects to s_k. For the point s_j we have the following results.

Lemma 3.3 *Let P be any simple monotone polygon that is generated from S_k.*
(1) *If the edge $(k-1,k)$ is on the top chain of P and s_j, $(j<k-1)$, is the point that connects to s_k, then s_j is below-visible from s_k.*

(2) *If the edge* $(k-1,k)$ *is on the bottom chain of* P *and* s_j, $(j < k-1)$, *is the point of the top chain that connects to* s_k, *then* s_j *is above-visible from* s_k.

Proof. We prove (1). If s_j is below-visible from s_k then the lemma holds. If s_j is not below-visible from s_k, there exists a line $l(i,k)$ such that s_j is above $l(i,k)$, where $j < i < k-1$. Because P is a monotone polygon, s_i is on the top chain of P. But s_i is below $l(j,k)$. Hence P can not be a simple monotone polygon. This contradiction proves that (1) is true. □

The proof for (2) is the same as that for (1).

Lemma 3.4 *Let* $P(j,k) = T(k) \bigcap \{$*edge* (j,k) *is on the bottom chain*$\}$ *for* $j \in V_b(k)$. *Then the number of monotone polygons in the set of* $P(j,k)$ *is* $BN(j+1)$.

Proof. For the monotone polygons in $P(j,k)$, we know that points s_j and s_k are on the bottom chains, and $s_{j+1}, \ldots, s_k$ are on the top chains. So the path of $s_j, s_k, s_{k-1}, \rightsquigarrow s_{j+1}$ is fixed. We can treat the path $s_j, s_k, s_{k-1}, \ldots, s_{j+1}$ as an edge $(j, j+1)$ that is on the bottom chain. Figure 4 shows an example. Now we know that the number of monotone polygons in the set of $P(j,k)$ equals the number of monotone polygons generated from S_{j+1} with the edge $(j,j+1)$ on the bottom chains. Hence the lemma is true. We call the set of $B(j+1)$ the *equivalent set* for $P(j,k)$. □

Using a similar proof we have the following result.

Lemma 3.5 *The number of polygons in* $B(k) \bigcap \{$*edge* (j,k) *is on the top chain*$\}$ *for* $j \in V_t(k)$ *is* $TN(j+1)$.

Theorem 3.6 *For any point set* S_k, *we have*

$$TN(k) = \sum_{j \in V_b(k)} BN(j+1) \tag{2}$$

$$BN(k) = \sum_{j \in V_t(k)} TN(j+1) \tag{3}$$

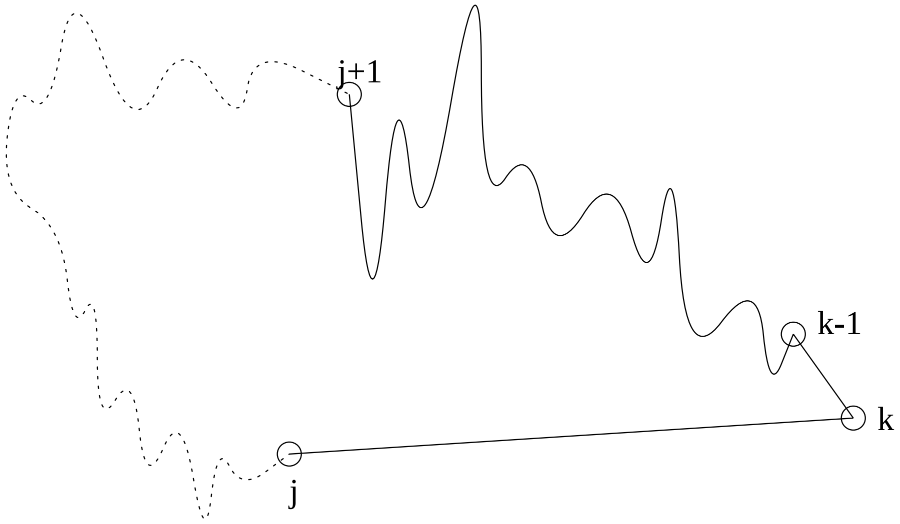
Proof. We prove formula 2. According to lemma 3.3, for any $P \in T(k)$, its bottom chain must use one of the points of $V_b(k)$. Let s_j be the point. Obviously $P \in P(j,k)$. From lemma 3.4, we know that the number of monotone polygons in $P(j,k)$ is $BN(j+1)$. Then the total number of different monotone polygons is $\sum_{j \in V_b(k)} BN(j+1)$. So 2 holds. □

The proof for formula 3 is the same as that for formula 2.

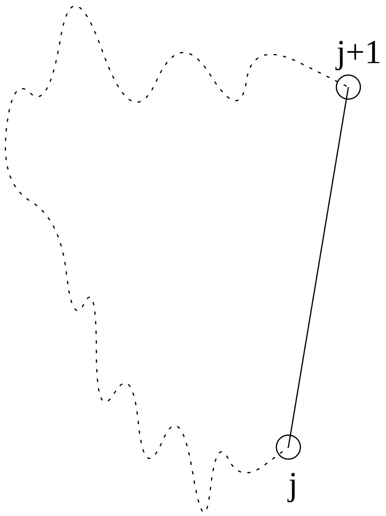
This theorem gives us the idea to calculate TN and BN, assuming that we have $V_b(k)$ and $V_t(k)$. The following is the procedure.

getTNandBN(n)
- $TN(2) = 1;$
- $BN(2) = 1;$
- FOR $i = 3$, TO n
    - $TN(i) = 0;$
    - $BN(i) = 0;$
    - FOR ALL $j \in V_b(i)$
        - $TN(i) = TN(i) + BN(j+1);$
    - FOR ALL $j \in V_t(i)$
        - $BN(i) = BN(i) + TN(j+1);$
- $N(n) = TN(n) + BN(n);$

After we get $TN(i)$ and $BN(i)$ for $i = 2, \ldots, n$, we start to generate a monotone polygon on $S(n)$ at random, under the uniform distribution. The following section gives the details.



(a) The original monotone polygon in $P(k)$



(b) The equivalent set of monotone polygons $B(j+1)$

Figure 4: The original set and its equivalent set.

3.2 Generating Monotone Polygons

For the general case, we give an algorithm to generate monotone polygons from S_n at random. Again we assume that we have $V_b(k)$ and $V_t(k)$, the *below-visible* and *above-visible* vertices. The algorithm scans the point set S_n backward from the right to the left to generate monotone polygons.

Generate

PICK AN x WITHIN $[1, N(n)]$ UNIFORMLY AT RANDOM;
ADD s_n TO top_chain; ADD s_n TO bottom_chain;

IF $x \leq TN(n)$
    ADD s_{n-1} TO top_chain;
    Generate_Top(n, x);
    ADD s_1 TO bottom_chain;
ELSE
    $x = x - TN(n)$;
    ADD s_{n-1} TO bottom_chain;
    Generate_Bottom(n, x);
    ADD s_1 TO top_chain;
END IF

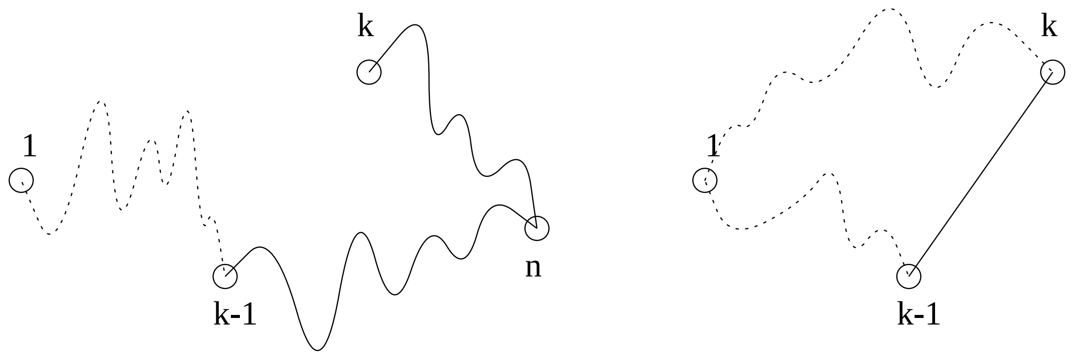
Generate_Top and **Generate_Bottom** deal with two cases. **Generate_Top** deals with the case in which s_{k-1} is on the bottom chain and s_k is on the top chain of the monotone polygon. In this case the undetermined points are $\{s_1, \ldots, s_{k-2}\}$. Then the set of all monotone polygons that can be generated from the original set is equivalent to that from the subset $S(k)$ with edge $(k-1, k)$ on the bottom chains; that is $B(k)$. **Generate_Bottom** deals with the case in which s_k is on the bottom chain and s_{k-1} is on the top chain. In this case the set of all monotone polygons that can be generated is equivalent to $T(k)$. These two cases are shown in Figure 5.

Generate_Top$(k,\ x)$
1. IF $k \leq 2$ RETURN;
2. FIND THE SMALLEST i SUCH THAT i SATISFIES:
        $x \leq \sum_{j \in V_b(k) \wedge j \leq i} BN(j+1)$;
3. ADD POINT s_i TO bottom_chain;
4. ADD ALL THE POINTS $s_{k-2}, s_{k-3}, \ldots, s_{i+1}$ TO top_chain;
5. $k = i + 1$;
6. $x = x - \sum_{j \in V_b(k) \wedge j < i} BN(j+1)$;
7. **Generate_Bottom**$(k,\ x)$

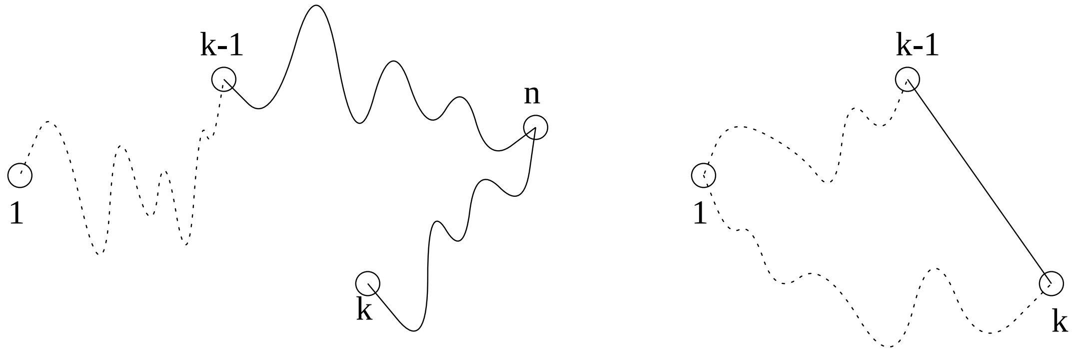
Generate_Bottom$(k,\ x)$
1. IF $k \leq 2$ RETURN;
2. FIND THE SMALLEST i SUCH THAT i SATISFIES:
        $x \leq \sum_{j \in V_t(k) \wedge j \leq i} TN(j+1)$;
3. ADD POINT s_i TO top_chain;
4. ADD ALL THE POINTS $s_{k-2}, s_{k-3}, \ldots, s_{i+1}$ TO bottom_chain;
5. $k = i + 1$;
6. $x = x - \sum_{j \in V_b(k) \wedge j < i} TN(j+1)$;
    END IF
7. **Generate_Top**$(k,\ x)$;

Our generating algorithm is to combine **getTNandBN** and **Generate** together.

Algorithm
    getTNandBN(n)
    Generate



(a) s_k is on the top chain and its equivalent set $B(k)$



(b) s_k is on the bottom chain and its equivalent set $T(k)$

Figure 5: The generating process.

The following section will show us that our **Algorithm** can generate monotone polygons uniformly at random.

4 The Analysis of the Algorithm

Let all the monotone polygons that can be generated from S_n be $\{P_1, P_2, \ldots, P_{N(n)}\}$. Let $\Omega(n) = \{P_1, P_2, \ldots, P_{N(n)}\}$. Then $\Omega(n)$ is a sample space. Each event in $\Omega(n)$ is an unique monotone polygon P_i that can be generated from S_n. We map $\Omega(n)$ to an integer set of $[1, N(n)]$. Each $x \in [1, N(n)]$ corresponds to an unique monotone polygon $P_x \in \Omega(n)$. Now we have the following results.

Lemma 4.1 *For $n \geq 2$ and $\forall x \in [1, TN(n)]$,* **Generate_Top** *generates an unique monotone polygon $P_x \in T(n) \subseteq \Omega(n)$; For $n \geq 2$ and $\forall x' \in [1, BN(n)]$,* **Generate_Bottom** *generates an unique monotone polygon $P_{x'} \in B(n) \subseteq \Omega(n)$.*

Proof We use induction on n (the size of the point set). Our base case is $n = 2$. Because of $TN(2) = 1$ and $BN(2) = 1$, we know that x is 1. From the procedure **Generate** the input of **Generate_Top** is that s_1 and s_2 are on the top chain and $x = 1$, and the input of **Generate_Bottom** is that s_1 and s_2 are on the bottom chain and $x = 1$. For this trivial base case **Generate_Top** and **Generate_Bottom** generate the correct trivial monotone polygon by simply returning to **Generate**.

Now for all $k < n$, we assume that $\forall x \in [1, TN(n)]$ **Generate_Top** generates an unique monotone polygon $P_x \in T(k)$ and $\forall x^{'} \in [1, BN(n)]$, **Generate_Bottom** generates an unique monotone polygon $P_{x^{'}} \in B(k)$.

For $k = n$, let $x_1, x_2 \in [1, TN(n)]$ and $P_{x_1}, P_{x_2} \in T(n)$ be the monotone polygons that are generated by **Generate_Top** according to x_1 and x_2. Now we prove that if $x_1 \neq x_2$, then $P_{x_1} \neq P_{x_2}$.

From **Generate** we know that s_{n-1} and s_n are on both top chains of P_{x_1} and P_{x_2}. Let $i_1 \geq 1$ and $i_2 \geq 1$ be the below_visible points found in **Generate_Top**. There are two cases in this situation.

Case 1: $i_1 \neq i_2$. Without loss of generality, let $i_1 < i_2$. From **Generate_Top** we know that for P_{x_1}, point s_{i_1} is on the bottom chain and point s_{i_2} is on the top chain. For P_{x_2}, point s_{i_2} is on the bottom chain. This proves $P_{x_1} \neq P_{x_2}$.

Case 2: $i_1 = i_2$. From **Generate_Top** we know that $k_1' = k_2' = i_1 + 1$, Since $x_1 \leq TN(n)$ and $x_2 \leq TN(n)$, we have

$$x_1' = x_1 - \sum_{j \in V_b(k) \wedge j < i_1} BN(j+1) \leq BN(i_1 + 1)$$

and

$$x_2' = x_2 - \sum_{j \in V_b(k) \wedge j < i_2} BN(j+1) \leq BN(i_2 + 1).$$

Because of $x_1 > \sum_{j \in V_b(k) \wedge j < i_1} BN(j+1)$ and $x_2 > \sum_{j \in V_b(k) \wedge j < i_1} BN(j+1)$, we have $x_1' \geq 1$. Then we have $x_1' \neq x_2'$, and $x_1' \in [1, BN(k_1')]$ and $x_2' \in [1, BN(k_2')]$. From our assumption, **Generate_Bottom** generates two different monotone polygons $P_{x_1'}$ and $P_{x_2'}$ with edge $(i_1, i_1 + 1)$ on the bottom chains. From lemma 3.4 and lemma 3.5, we know that $P_{x_1'}, P_{x_2'} \in B(k_1')$ and $B(k_1')$ is the *equivalent set* of $P(k_1', k)$. Then we know that the part of polygons of $P_{x_1'}$ and $P_{x_2'}$ without edge $(i_1, i_1 + 1)$ are on the monotone polygons of P_{x_1} and P_{x_2}. Hence $P_{x_1} \neq P_{x_2}$. $\Box$

Using the similar proof, we can prove that for $\forall x' \in [1, BN(n)]$, **Generate_Bottom** generates an unique monotone polygon $P_{x'} \in B(k)$.

From this lemma we immediately get the following result.

Theorem 4.2 *For* $n \geq 2$ **Generate** *generates monotone polygons from* $\Omega(n)$ *uniformly at random.*

Proof **Generate** picks an $x \in [1, N(n)]$ uniformly at random. If $x \leq TN(n)$ **Generate** calls **Generate_top**. If $x > TN(n)$ **Generate** calls **Generate_bottom**. From lemma 4.1 **Generate** generates an unique monotone polygon $P_x \in \Omega(n)$; We know that the x picking behavior determines the generating behavior of **Generate**. Hence the probability of a monotone polygon generated by **Generate** equals to the probability of picking a x from $[1, N(n)]$. So the theorem is true. In other words, **Generate** retains an uniform monotone polygon generator. $\Box$

Corollary 4.3 **Generate** is *complete.*

5 Computing Visibility

The algorithms of the previous section assumed that the *above-visible* and *below-visible* sets, $V_t(i)$ and $V_b(i)$ for $i = 1, \ldots, n$, were available. A closer look, however, shows that these sets are only needed for one index i at a time: algorithm **getTNandBN** needs the sets in increasing order and algorithms **Generate_top** and **Generate_top** need them in decreasing order.

In this section, we show how to calculate each of the sets $V_t(i)$ incrementally as i increases (In Section 5.1) and as i decreases (In Section 5.2), using time proportional to $|V_t(i)|$ and $O(n)$ space to compute $V_t(i)$ from $V_t(i-1)$ or $V_t(i+1)$.

The idea is the following. Let S_k denote the monotone chain with vertices $s_1, s_2, \ldots, s_k$. If we think of S_k as a fence and compute the shortest paths in the plane above S_k from s_k to each s_i with $i \leq k$, then we obtain a tree that is known as the *shortest path tree rooted at* s_k [2, 3]. The *above-visible* set $V_t(i)$ is exactly the set of children of s_k in the shortest path tree rooted at s_k. Thus, we will incrementally compute shortest path trees rooted at $s_1, s_2, \ldots, s_k$ to get the *above-visible* sets.

We represent shortest path trees (in which a node may have many children) by binary trees in which each node has pointers to its uppermost child and next sibling. Section 5.1 gives the details for computing these trees in the forward direction: computing $V_t(i)$ from $V_t(i-1)$. Section 5.2 gives the details for the reverse direction: computing $V_t(i)$ from $V_t(i+1)$.

5.1 Computing Visibility Forward

We use a tree data structure to calculate $V_t(k)$ and $V_b(k)$ recursively. Assuming $V_t(k-1)$ and $V_b(k-1)$ have been calculated, we calculate $V_t(k)$ and $V_b(k)$ according to the results of $V_t(k-1)$ and $V_b(k-1)$. The data structure that we use in the calculation is the tree of the shortest paths rooted at vertex k.

We store $top\_tree(i)$ and $bot\_tree(i)$ using child and sibling pointers. For each vertex $j \in [1, n]$, we have a record for top_tree

j:		
	ptr	*ptr* stores the coordinates of vertex j
	upc	*upc* is a pointer pointing the upper child of j in $top\_tree(k)$
	sib	*sib* is a pointer pointing the sibling of j in $top\_tree(k)$

and a record for bot_tree

j:		
	ptr	*ptr* stores the coordinates of vertex j
	lwc	*lwc* is a pointer pointing the lower child of j in $bot\_tree(k)$
	sib	*sib* is a pointer pointing the sibling of j in $bot\_tree(k)$

We define $Children(k)$ be the set of points that contains the upper and lower children of k and their siblings in the top_tree and bot_tree. The initial value of top_tree for the recursive calculation is $1.ptr = s_1$, $1.upc = nil$ and $1.sib = nil$. We assume that $top\_tree(i-1)$ has been completed. Then we call $\mathbf{Make\_V}_t$ to calculate the *above-visible* set, V_t. In order to get V_t, the procedure $\mathbf{Make\_V}_t$ calls the procedure **Make_top** to calculate the $top\_tree(i)$.

> $\mathbf{Make\_V}_t(i)$
>
> $\quad t = tmp;$
>
> $\quad \mathbf{Make\_top}(i-1, i, \mathbf{Var}:\ t);$
>
> $\quad i.upc = tmp.sib;$

Procedure $\mathbf{Make\_top}(i-1, i, lastsib)$ makes the tree edge from k to j in $top\_tree(k)$, and puts it as the sibling of *lastsib* and updates *lastsib*. Then it recursively build the $top\_tree(k)$. One example is shown in Figure 6.

> $\mathbf{Make\_top}(j, k, \mathbf{Var}:\ lastsib)$

WHILE $j.upc \neq nil$ and k is above $l(j.upc, j)$

  Make_top$(j.upc, k, \textbf{Var}: \; lastsib)$;

  /* make subtree for this child of j, which can be seen by k. */

  $j.upc = j.upc.sib$; /* consider next child of j */

END WHILE

$lastsib.sib = j$; /* make the connection to j, one of the children of k */

$lastsib = j$;

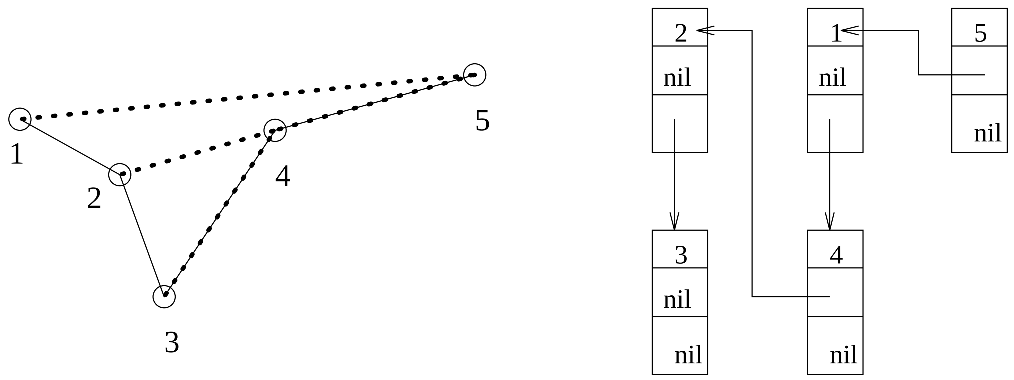


Figure 6: A point set S_5 and the data of *top_tree*(5).

To compute the *bot_tree* is similar to computing the *top_tree*. We need only change *upc* and 'above' in procedure **Make_V**$_t(i)$ and **Make_top** into *lwc* and 'below' to get the procedures **Make_V**$_b(i)$ and **Make_bot**. We use **Make_V**$_b(i)$ and **Make_bot** to compute *bot_tree*(i) from *bot_tree*$(i-1)$. One example is shown in Figure 7.

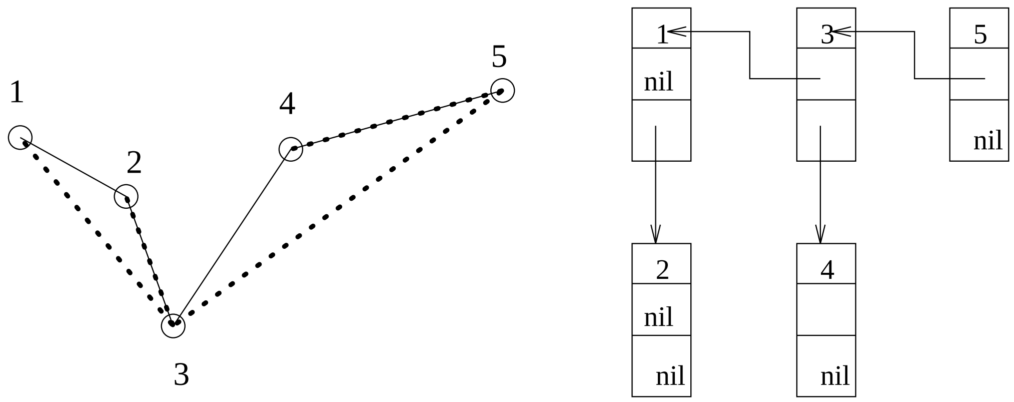


Figure 7: A point set S_5 and the data of *bot_tree*(5).

Knowing *top_tree*(k) and *bot_tree*(k), we know the *above-visible* and *below-visible* point sets, $V_t(k)$ and $V_b(k)$ of vertex k. Now we give the theorem to show us how to get $V_t(k)$ and $V_b(k)$ from *top_tree*(k) and *bot_tree*(k).

Let r be a record in the *top_tree* or *bot_tree*. We define that $r.sib^i = r.sib^{i-1}.sib$, for any integer $i \geq 0$, and $r.sib^0 = r$. Then we know that the upper child of k and its siblings are this kind of format. Now we claim that the upper child of k and its siblings are the vertices visible from k, and any vertex that is visible from k is either the upper child of k or its sibling. This is proved in the next theorem.

Theorem 5.1 *Let $CT(k)$ be the set of points in top_tree(k) that satisfy $\forall j \in Children(k), \exists i$ such that $j = k.upc.(sib)^i$. Let $CB(k)$ be the set of points in bot_tree(k) that satisfy $\forall j \in Children(k), \exists i$ such that $j = k.lwc.(sib)^i$. We have $V_t(k) = CT(k) - \{k-1\}$ and $V_b(k) = CB(k) - \{k-1\}$.*

Proof First we prove $V_t(k) = CT(k) - \{k-1\}$. If $V_t(k) = \emptyset$, there is no point that is above line $l(k-1,k)$. This means that there is no $l(i,k-1)$ that is below k. From **Make_top**, we know that $CT(k) = \{k-1\}$. hence $V_t(k) = CT(k) - \{k-1\}$. If $CT(k) = \{k-1\}$ there is no $l(i,k-1)$ that is below k for $i = 1,\ldots,k-2$. So there exists no point that is above $l(k-1,k)$. Hence $V_t(k) = \emptyset = CT(k) - \{k-1\}$.

For the general situation, $\forall j \in V_t(k)$, we have j is above all $l(i,k)$ for $i = j+1,\ldots,k-1$ that implies that k is above all $l(j,i)$ for $i = j+1,\ldots,k-1$. Now we prove $j = k.upc.(sib)^i$, $i \geq 0$. If there is no $j' \in V_t(k)$ and $j' < j$ such that k is above $l(j',j)$ then $j = k.upc$. Otherwise, $j = j'.sib$. Similarly this induction can be applied to j', that is, $j' = k.upc.(sib)^{i'}$. Then we have $j = k.upc.(sib)^{i'+1}$. So $V_t(k) \subseteq CT(k) - \{k-1\}$.

$\forall j \in CT(k) - \{k-1\}$, we know $j = k.upc.(sib)^i$. Then j is above all $l(i,k)$, for $i = j+1,\ldots,k-1$. Otherwise, there exists a point, say j', such that $j' > j$ and j is below $l(j',k)$. Then $l(j,k)$ is below $l(j',k)$ that means k is below $l(j,j')$. From **Make_top** we know that j can not be the format of $k.upc.(sib)^i$, $i \geq 0$. This contradiction proves that j is above all $l(i,k)$, for $i = j+1,\ldots,k-1$. Then we have that $j \in V_t(k)$ that implies $V_t(k) \supseteq CT(k) - \{k-1\}$. Now we have $V_t(k) = CT(k) - \{k-1\}$. □

The proof for $V_b(k) = CB(k) - \{k-1\}$ is similar to the proof above.

5.2 Computing Visibility backward

In procedure **Generate_Top** and **Generate_Bottom**, we need to find the smallest i in line 2. Here we assume that *top_tree*$(k+1)$ and *bot_tree*$(k+1)$ have been completed, we use procedures **Back_top** and **Back_bot** to generate *top_tree*(k) and *bot_tree*(k). Let $t_{i-j} = (k+1).upc.sib^j$, for $j = 0,1,\ldots,i$. Then $t_i = (k+1).upc$ and $t_0 = k$. Let $Q = \{t_j, j = 0,\ldots,i\}$. From theorem 5.1, we know $Q = V_t(k+1) - \{k\}$. If we take t_0 as the origin of of coordinates, according to the *above-visible* definition, the points in Q are sorted lexicographically by polar angle and distance from t_0. Then from Graham-Scan we can get the correct *top_tree*(k). This is similar for calculating *bot_tree*(k). The following are the procedures.

Back_top$(k+1, k)$
FIND i, SUCH THAT $(k+1).upc.sib^i = k$;
FOR $j = 0$ TO i
  $t_{i-j} = (k+1).upc.sib^j$;
IF $i = 0$ RETURN;
ELSE IF $i \geq 1$
    Graham-Scan-Top$(i, t_0, \ldots, t_i)$;
END IF

Graham-Scan-Top$(i, t_0, \ldots, t_i)$
Push(t_0,S); /* S is a stack */
$t_1 = t_0.upc$;

```
3      t_0.upc = t_1;
4.     Push(t_1,S);
5.     FOR j = 2 TO i
6.          WHILE the angle formed by points NEXT-TO-Top(S), Top(S),
                     and t_j makes nonleft turn
7.               Pop(S);
            END WHILE
8.          t_j = Top(S).upc;
9.          Push(S,t_j);
```

One example to calculate *top_tree*(k) from *top_tree*$(k+1)$ is shown in Figure 8.

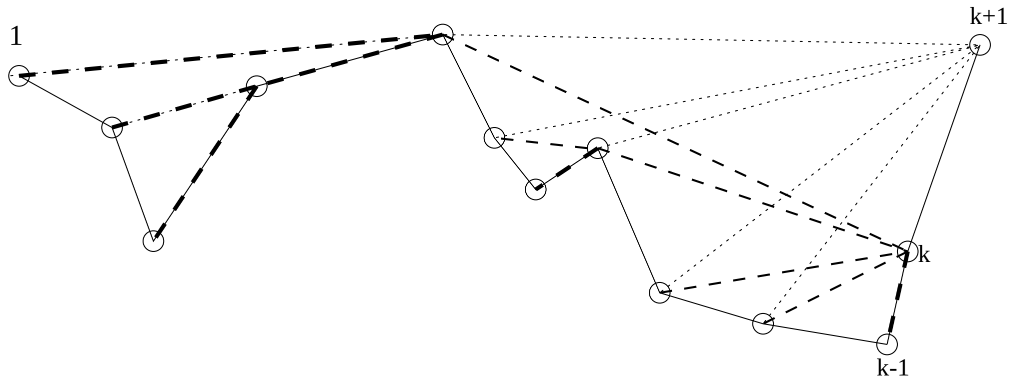


Figure 8: *top_tree*(k) is generated from *top_tree*$(k+1)$.

Similarly we have the procedure to compute *bot_tree*(k) from *bot_tree*$(k+1)$. They are called **Back_bot** and **Graham-Scan-Bot**(i, Q). We get them simply by changing *upc* and 'nonleft turn' of **Back_top** and **Graham-Scan-Top**(i, Q) into *lwc* and 'nonright turn'.

Now we prove that these procedures compute correct results.

Theorem 5.2 **Back_top** *and* **Graham-Scan-Top**(i, Q) *correctly compute top_tree*(k) *from top_tree*$(k+1)$*.* **Back_bot** *and* **Graham-Scan-Bot**(i, Q) *correctly compute bot_tree*(k) *from bot_tree*$(k+1)$*.*

Proof We prove that **Back_top** and **Graham-Scan-Top**(i, Q) correctly compute *top_tree*(k) from *top_tree*$(k+1)$. In **Back_bot** we first find the upper child of $k+1$ and its siblings. In order to get *top_tree*(k) from *top_tree*$(k+1)$ we must cut the edges of these vertices with $k+1$ and reconnect them with appropriate vertices. These points are the only points that need to be reconnected.

In **Graham-Scan-Top**(i, Q) point k is always kept in the bottom of the stack S. For any vertex visible from $k+1$, there two cases. Case 1 is that it is visible from k. Case 2 is that it is not visible.

Case 1: point j, is visible from k, then all the points in the stack S are popped out but k. Now we output edge (j, k) and point j is pushed into S. Now there are at least two points in the stack S.

Case 2: point j is not visible from k. We know that j must be visible from a vertex in S, say j'. Then all the points on top of j' are popped out, and we output the edge (j, j') and j is pushed into S.

After we checked all the points visible from $k+1$, we reconnect the points correctly. □

Similarly we can that prove **Back_bot** and **Graham-Scan-Bot**(i, Q) correctly compute *bot_tree*(k) from *bot_tree*$(k+1)$.

Now we have all the procedures to build up our algorithm. Next we give its time and space complexity.

6 Time and Space Complexity Analysis

Lemma 6.1 *The runtime of* **Make_top***$(k-1, k,$* **Var***$:\ t)$ is $O(|V_t(k)|)$. And the runtime of* **Make_bot***$(k-1, k,$* **Var***$:\ t)$ is $O(|V_b(k)|)$.*

Proof Because of the similarity, we only prove the runtime of **Make_top**$(k-1, k,$ **Var**$:\ t)$ is $O(|V_t(k)|)$.

Let us assign the following amortized costs:

WHILE checking	1
updating $j.upc$	1
updating $lastsib$	1

and each time we encounter the upper child, $k.upc$ or its sibling $k.upc.sib^i$, but excluding $k-1$, we get 3 credits. Clearly from theorem 5.1, we know that the total number of $k.upc$ and $k.upc.sib^(i)$, excluding $k-1$, is $|V_t(k)|$.

We shall now show that we can pay any operation costs by charging the amortized costs. We start from **Make_top**$(k-1, k,$ **Var**$:\ t)$ and we have 3 credits. Clearly if j is visible from k, WHILE checking succeeds. From this we get 3 more credits to pass to the next call to **Make_top**$(j, k,$ **Var**$:\ lastsib)$. Then this call receives 3 credits to pay for its own checking and updating costs. If j is not visible from k, WHILE checking fails. Then the current call to **Make_top** saves 1 credit for upper level **Make_top** to pay another WHILE checking. We know that **Make_top**$(j, k,$ **Var**$:\ lastsib)$ with 3 credits can pay their own costs and the number of total recursive calling for **Make_top**$(j, k,$ **Var**$:\ lastsib)$ is $|V_t(k)|$. Then $3 * |V_t(k)|$ will pay all the costs. So the runtime of **Make_top**$(k-1, k,$ **Var**$:\ t)$ is $O(|V_t(k)|)$. □

Theorem 6.2 Algorithm *has time complexity of $O(K)$ and space in $O(n)$. where K is the total number of above-visible and below-visible points of the points in the point set.*

Proof From lemma 6.1 we have the runtime of **getTNandBN** is, for some constant c,

$$\sum_{k=3}^{n} c * (|V_t(k)| + |V_b(k)|) \leq cK = O(K).$$

Clearly the runtime of **Back_top** is $O(|V_t(k)|)$ and the runtime of **Back_bot** is $O(|V_b(k)|)$.

The time complexity of **Generate** depends on the time complexity of **Generate_Top** and **Generate_Bottom**. Because they have a similar structure the time complexity of **Generate_Top** and **Generate_Bottom** is the same. Let t_k be the run time of **Generate_Top**(k, x) From line 2 to 6, the time depends on the number of *above-visible* and *below-visible* points of s_k. Then we have, for some constant c

$$t_k = \sum_{j=i+1}^{k} c * (|V_t(k)| + |V_t(k)| + k - i) + t_{i+1}.$$

So

$$t_n \leq \sum_{k=1}^{n} c * (|V_t(k)| + |V_b(k)|) + n \leq c * (K + n).$$

Hence the run time of **Generate** is $O(n + K)$. Obviously, $n \leq K \leq n^2$. The time complexity of our **Algorithm** is $O(n + K) + O(K) = O(K)$.

In the process of generating we need only to store the point set S_n, $top\_tree(n)$, $bot\_tree(n)$, and $TN(i)$ with $BN(i)$, for $i = 2, \ldots, n$. Since each of the data structures use no more than $O(n)$ memory space, we have that the memory space of **Algorithm** is $O(n)$. □

7 Conclusion

We have presented an algorithm to generate monotone polygons uniformly at random. The time complexity of our algorithm is $O(K)$. The space complexity of our algorithm is $O(n)$. We have given the detail analysis of the algorithm and the proof of its correctness. A random monotone polygon generator is useful for testing the many algorithms that accept a simple polygon or a group of simple polygons as input.

We are also interested in finding a polynomial algorithm to generate general simple polygons randomly from an arbitrary set of points. We have not found any useful property for generating general simple polygons.

References

[1] P. Epstein and J. Sack. Generating triangulation at random. In *4th CCCG*, pages 305 – 310, 1992.

[2] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[3] Leonidas J. Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, October 1989.

[4] Henk Meijer and David Rappaport. Upper and lower bounds for the number of monotone crossing free hamiltonian cycles from a set of points. *ARS Combinatoria*, 30:203–208, 1990.

[5] Joseph S. B. Mitchell and Gopalakrishnan Sundaram. Generating random geometric objects. *unpublished manuscript*, 1993.