

Implementations of Algorithms For Line-Segment Intersection

Sebastian Kanthak

December 18, 2003

Abstract

In this paper, I analyze the constant factors for implementations of line segment intersection algorithms. I explore how these algorithms could be applied to stochastic local search algorithms to determine the rectilinear crossing number of complete graphs.

1 Introduction

The problem of finding all intersecting pairs in a collection of n line segments has been well studied and asymptotically optimal algorithms have been proposed. Determining the rectilinear crossing-number of graphs is a hard problem to which stochastic local search algorithms have been successfully applied [9].

In this paper I will explore, how algorithms for reporting line segment intersection could be applied successfully to this problem domain. I will make a first step into this direction by implementing a naive and two more sophisticated algorithms and analyzing the constants normally hidden in the Big-Oh notation.

In section 2, I will describe the problem of minimizing crossings in complete rectilinear graphs. In sections 3 and 4, I will survey algorithms for line segment intersection and present my implementation of three algorithms. The performance of this implementation will be evaluated in section 5. I will conclude with a discussion of the results and future and related work in sections 6 and 7.

2 Application Area

Determining whether a graph can be drawn in the plane without edge-crossings can be solved in polynomial time. Finding the minimum number of edge-crossings $cr(G)$ for all drawings in the plane is NP-complete [11]. When one restricts the possible drawings to those that only have straight edges — so called rectilinear drawings — one arrives at the problem of minimizing the rectilinear crossing number. It is not known, whether this problem is still NP-hard, but no efficient algorithms are known.

In [9] stochastic local search algorithms have been applied successfully to this problem. These algorithms start with a random drawing and try to minimize the number of edge-crossings by making small modifications to the drawing. The authors proposed three different local search steps.

1. Locate the vertex that is responsible for the maximum number of crossings and randomly place it to new positions until the number of crossings is reduced. This requires a time of $O(n^4)$ (where n is the number of vertices) since the responsibilities of all vertices have to be recomputed.
2. Chose a vertex at random and continuously place it at random positions until a better drawing is achieved. As the responsibilities do not have to be recomputed only the edges incident to the newly placed vertex have to be tested for intersection, resulting in $O(n^3)$ time per search step.
3. Start with a good drawing for $n - 1$ nodes and repeatedly add the n th vertex at random positions. Each step takes $O(n^3)$ time again.

For all these strategies a line segment intersection problem has to be solved. The above time bounds were achieved by naively testing pairs of line segments for intersection. In the following sections, I will analyze, how more sophisticated algorithms could be used to improve the performance of the search steps.

3 Algorithms

Finding all intersecting pairs in a collection of n line segments can be done in $O(n^2)$ time easily by testing all pairs for intersection. As there can be $O(n^2)$ intersections in the worst-case, this is optimal with respect to n . If one performs an *output-sensitive* analysis and considers the number k of intersections in the input, however, one can achieve an optimal algorithm with running time $O(n \log n + k)$. In this section, n will always denote the number of line segments and k will stand for the number intersections among these segments.

Various algorithms with different running-time and space-requirements have been proposed:

3.1 Early deterministic algorithms

The classical Bentley-Ottmann algorithm [3] proceeds by sweeping a vertical line from left to right over the segments and maintaining the intersections of the segments with the sweep line. Event points are segment endpoints and intersections points. The intersection points are detected by testing segments adjacent on the sweep line for intersections and inserting an event if necessary. These checks are performed whenever segments become newly adjacent. This is sufficient since segments have to be adjacent on the sweep line prior to their intersection.

Due to the dynamic nature of the event queue and the current status on the sweep line, balanced binary search trees are used for both structures. The algorithm requires $O((n+k)\log n)$ time and $O(n+k)$ space.

As the dependence on $k\log n$ is not optimal, future research has focused on reducing the dependence on k to linear.

3.2 Optimal deterministic algorithms

An optimal deterministic algorithm has been given by Balaban [2]. The algorithm works on a tree of vertical slabs, similar to a segment tree. A form of fractional cascading is used to locate endpoints quickly in segments. To reduce the working memory of the algorithm, the tree is only implicit in the algorithm through its recursive structure. As only one branch is active at a time, the whole tree never has to be held in memory. The algorithm is asymptotically optimal with a running time of $O(n\log n+k)$ and space $O(n)$.

Another deterministic algorithm with optimal running time has been given by Chazelle and Edelsbrunner [4]. It uses a lot of sophisticated algorithm techniques and data structures. The algorithm is an enhanced sweep-line algorithm that makes use of pre-processing information and maintains some additional information besides the intersections with the sweep line. It requires $O(n+k)$ space in the worst-case.

3.3 Optimal randomized algorithms

As the optimal deterministic algorithms are fairly complicated and could involve high constants, simpler randomized algorithms with the same expected running-time have been proposed. The expectation holds in general and is not based on any input distribution.

A randomized divide-and-conquer algorithm has been proposed by Clarkson [5]. This algorithm is optimal with an expected running time of $O(n\log n+k)$ and $O(n)$ space.

A simpler randomized incremental algorithm has been proposed by Mulmuley [12]. It incrementally adds line segments in random order and maintains the partition induced by these segments and *contractable* vertical lines through all segment endpoints. These vertical lines, called attachments, extend in both directions to the next input segment. The algorithm uses only very simple data structures that represent the planar subdivision. The expected running time is $O(n\log n+k)$ with $O(n+k)$ space requirements.

3.4 Application to Crossing Number Problem

These algorithms could be applied to replace the naive line segment intersection in the stochastic local search algorithms described before. As these algorithms spend most of their time computing the number of intersections, a considerable speed-up could be expected by improving the performance of this part.

A complete graph with n vertices has $m = n(n - 1)/2$ edges. Computing all intersections by simply checking all pairs thus requires $m(m - 1)/2 = O(n^4)$ comparisons, whereas an optimal algorithm could do the same work in $O(m \log m + k) = O(n^2 \log n + k)$. It remains, however, to estimate the number of intersections k . In the worst-case, k could be of the order m^2 which would dominate the $m \log m$ part, so even an optimal algorithm would not run faster than $O(n^4)$.

Unfortunately, according to [9] there is a lower bound for the rectilinear crossing number of a complete graph K_n . The following relationship holds for $n \geq 4$.

$$\overline{\text{cr}}(K_n) \geq \frac{n}{n - 4} \cdot \overline{\text{cr}}(K_{n-1})$$

Therefore, we get a lower bound of $O(n^4)$ for the number of edge crossings in any rectilinear drawing of K_n .

$$\overline{\text{cr}}(K_n) \geq c \cdot \prod_{i=4}^n \frac{n}{n - 4} = c \cdot \frac{n!}{(n - 4)!4!} = c' \cdot n(n - 1)(n - 2)(n - 3) = O(n^4)$$

This means, that even an optimal algorithm can not find the number of edge crossings asymptotically faster than the simple test-all-pairs algorithm. However, as one tries to *minimize* the number of edge crossings, one can still expect k to be considerably smaller than $m \cdot (m - 1)/2$, the maximum number of intersections. In fact, empirical results [14, 9] show, that the optimal rectilinear crossing number is of the order $0.1 \cdot m(m - 1)/2$, that is 10% of the worst-case.

While the simple line segment intersection algorithm is not sensitive to this fact, optimal algorithms are, so one could expect them to run faster by a *constant* factor. However, one still has to consider the constants hidden in the Big-Oh notation which can be done best by implementing these algorithms.

4 Implementation

As I was interested in small constants, I decided to implement only simple algorithms. This is especially important since the constants of the simple test-all-pairs algorithm are very low as it mainly consists of two for-loops and some sidedness-tests. I finally implemented the following three algorithms:

1. The simple test-all-pairs algorithm as a reference
2. The classical Bentley-Ottmann sweep-line algorithm as a deterministic algorithm
3. The algorithm proposed by Mulmuley as a simple, yet optimal randomized algorithm

I decided against the optimal deterministic algorithms as they use fairly complicated data-structures and techniques which lead one to expect high constants in their implementation. Randomized algorithms promised to give the same optimal run-time at the cost of losing determinism. This price, however, did not seem very high, especially since stochastic local search algorithms already rely on randomness inherently.

I implemented all algorithms in a common framework in Java. The source code can be downloaded at [7]. I mainly chose Java to simplify the implementation of maintaining the planar subdivision in Mulmuley's algorithm by using Java's built-in garbage collection. In the following I will briefly describe the implementation. I will omit the usual `ca.ubc.cs.lineseg` prefix on all package- and class-names for brevity.

4.1 Geometric Primitives

All algorithms use some geometric primitives implemented in the `common` package. The most important ones are sidedness-tests and intersection detecting between two line segments.

Given two points a and b and a query point q the side of q with respect to the directed line through a and b is determined with the sign of the determinant

$$\left| \begin{pmatrix} b.x - a.x & q.x - a.x \\ b.y - a.y & q.y - a.y \end{pmatrix} \right|$$

To determine, whether two line segments intersect, I test whether the lines that support the segments separate the two endpoints of the other segment. Special care had to be taken to handle degenerate cases where one endpoint lies on the other segment or where both segments lie on the same line.

The point of intersection is computed by solving a two-dimensional linear equation. If the two segments overlap, the point of intersection is defined to be the left-most common point or the lowest common point in case of vertical segments.

4.2 Simple Test-All-Pairs

Using the above primitives, this algorithm is reduced to two nested for-loops. It is very robust against all kinds of degeneracies and can handle everything from multiple segments intersecting in one point to endpoints lying on another segment and overlapping segments. Pseudo code for this algorithm can be seen in algorithm 1.

4.3 Bentley-Ottmann

This algorithm requires two balanced search trees as data structures. One for the event queue and one to maintain the intersections on the sweep line. I used Java's built-in `TreeSet` and `TreeMap` classes that are implemented using red-black-trees internally.

Algorithm 1 SimpleIntersection

```
for all pairs  $(s, t)$  of line segments do  
  if  $s$  and  $t$  intersect then  
    report intersection  
  end if  
end for
```

Special care had to be taken to make the algorithm robust against degeneracies. I followed the implementation sketched in [8] and altered it to handle overlapping segments so that it now handles all degeneracies gracefully.

As the algorithm uses a vertical sweep line it has problems with vertical segments. This is solved by symbolically rotating the sweep line slightly in counter-clockwise order. Events are presented as points that are interpreted as the moment when the (rotated) sweep line contains that point. Points are handled from left to right and bottom to top. This is consistent with the fact that the sweep line intersects the lower endpoint of a vertical segment first due to the slight rotation.

For every event I store all segments that have their starting point at this event point. Using the balanced search tree I make sure that every event point is contained at most once in the event queue.

The order of the segments intersecting the sweep line is maintained in a balanced search tree, too. A `Comparator` object that defines the order of the segments is used. It is parameterized by the current event point. It compares line segments by comparing the y -coordinate of their intersection with the sweep line. Ties are broken in the following way: If the the point of intersection is below the current event point (meaning that the event is “in the past”) the comparison is *symbolically* performed as if it would have occurred at an even slightly to the right, if it is above slightly to the left. Ties on the current event point are broken by a flag that tells, whether the comparison should be performed slightly on the left or slightly on the right. Overlapping segments are considered to be equal and are ordered according to some fixed order on the input segments for consistency.

Pseudo code can be found in algorithm 2. The algorithm proceeds as follows: It first creates events for all start- and endpoints of input segments. It then handles all events in sorted order. For each event, it fetches the associated list L of segments starting at that point and finds and removes all segments in the search tree that intersect the current event point. It reports all those segments as intersections in that point. It then switches the order of the comparison function by changing the flag to be slightly after the current event. It reinserts all previously removed segments that do not have their endpoint at the current event point (because they have to be removed from the structure at this point anyways) and additionally insert the segments from L (that are starting at this point). It checks the top and bottom neighbors above and below the current event point for new intersections and adds them as event points if it finds any.

This way the algorithm is robust against all kinds of degeneracies, including

Algorithm 2 Bentley-Ottmann

```
for all segments  $s$  do
    Create events for the endpoints of  $s$ 
end for
while event queue not empty do
    Remove the smallest event point  $p$  from the queue
    Let  $L$  be the set of segments that start at  $p$ 
     $I \leftarrow$  all segments in the sweep line structure that contain  $p$ 
    remove  $I$  from sweep line structure
    if  $|L| + |I| \geq 2$  then
        Report intersections of  $L \cup I$ 
    end if
     $C \leftarrow \{s \in I \mid p \text{ is not endpoint of } s\}$ 
    Insert  $C$  in sweep line structure in reversed order
    Check for new intersections among segments above and below  $p$ 
end while
```

vertical segments and overlapping segments, as well as segments intersecting on their endpoints.

After I finished the implementation of the algorithm, I found a tech report [10] describing an implementation in C++ with a similar approach to handle degeneracies.

4.4 Mulmuley's randomized algorithm

I chose this algorithm because it promised to have low constant factors due to its simple data structure and the use of randomization. The original paper [12] makes the non-degeneracy assumptions that no two endpoints share the same x -coordinate¹ and that no more than two segments intersect in one point².

I got rid of the first assumption by slightly rotating the input segments in a pre-processing step. Note that this preserves all intersections, even degenerate ones. One should be able to do this rotation *symbolically*, but one has to be careful to distinguish symbolically rotated points (all endpoints and intersection points of input segments) from non-rotated points (endpoints of vertical attachments).

I generalized the algorithm to handle the case of multiple segments intersecting in one point, as I will describe below. My implementation can not handle overlapping segments. Note, however, that these segments could be found in a preprocessing step in $O(n \log n)$ time by sorting segments into buckets corresponding to lines and then split up segments on the same line that overlap.

The algorithm proceeds by maintaining the planar subdivision induced by the segments and *vertical attachments* passed through segment endpoints that

¹thereby excluding vertical segments

²this excludes overlapping segments as well

extend up to the next input segment. The faces of the subdivision are represented as their vertices linked in counter clockwise order. Note that a point is only present in a face if it is *visible* in that face. This means that the border of the face has a tangent discontinuity at this point. This way, the upper endpoint of a vertical attachment is not visible lying in the face directly above it, so there is not vertex corresponding to this point in that face, thereby reducing the number of vertices and the average face length. To allow for traversals between faces, every vertex has a reference to the vertex on the same point in the next face in counter clockwise direction. This is the only data structure the algorithm uses.

The algorithm starts with the partition induced by the attachments of all endpoints. It then adds segments in random order. Every endpoint has a reference to its attachment, thus the starting face can be found quickly. The algorithm inserts a segment by traversing all the faces the segment intersects. Once it knows the current face, it can simply traverse the vertices of this face until it reaches the place where the segment exits this face and split up the face. If the segment passes through a vertical attachment, it contracts the attachment, thereby merging two faces.

I had to generalize the algorithm as originally described by Mulmuley to handle the case of multiple segments intersecting in one point. Pseudo case for the algorithm can be found in algorithm 3. The real code has to consider even more cases than shown in the pseudo code to handle all degeneracies. The reader is referred to the actual Java code for details.

Algorithm 3 Mulmuley's Algorithm

```

sort endpoints of segments and create initial partition
for all segments  $s$  in random order do
  {insert segment into partition}
  while current point not right endpoint of  $s$  do
    Create new vertices above and below the entry point of  $s$ 
     $(u, v) \leftarrow$  vertices where  $s$  leaves the current face
    if  $s$  passes through attachment above or below associated endpoint then
      Split current face at  $(u, v)$ 
      contract attachment
      Merge faces above or below attachment
    else
      Split current face  $(u, v)$ 
    end if
    Transition to next face  $s$  passes through
  end while
end for

```

While traversing the faces the algorithm builds the new partition by merging and splitting faces. It always has two vertices `dangBottom` and `dangTop` that represent the vertices above and below the segment inserted currently at the start of the face. These vertices are *dangling* because the successor of the top

and the predecessor of the bottom vertex³ are not yet known. Once the face is left, the references to and from these dangling vertices can be set.

Note that although this algorithm works incrementally, it is not *on-line* because it has to know all endpoints in advance.

4.5 Implementation Issues

While implementing the algorithms I learned (sometimes painfully), that one has to carefully consider a lot of details that are not obvious from the high-level descriptions.

4.5.1 Degeneracies

Handling degeneracies has to be considered carefully before implementing the algorithm. Otherwise, they quickly invalidate the invariants of the algorithms and it is hard to fix the algorithms afterwards.

4.5.2 Numerical stability

I used double precision floating point arithmetic⁴ for all computations. When comparing against zero, I always used a small epsilon. To compute the point of intersections between two segments, I always used the *original* segments instead of intermediate results. However, I still ran into problems for bigger inputs, that I finally tracked down to floating point inconsistencies. For example, the intersection point of two segments was suddenly recognized as lying *left* of one of the segments. I could overcome these problems by delaying divisions as long as possible in the implementation of the geometric primitives and increasing the epsilon from 10^{-9} to 10^{-6} .

The C++ library CGAL [6] provides geometric primitives using exact computation and floating point filters. The usage of this library would have solved this problem in a more robust way. However, too much Java code had already been written when I discovered the floating point problems.

4.5.3 Linked Data Structures

Mulmuley's algorithm uses data structures that work by linking vertices together. Although these data structures only use the minimum number of links required to keep all vertices connected, one has to be very careful in the implementation of splitting and merging nodes to set all required links correctly. In fact, most of the code of the implementation consists of assignments, that modify these links.

³Note that vertices do not have a reference to its predecessor. However, the successor reference of the predecessor of the dangling top vertex is not set appropriately.

⁴as provided by Java's `double` data-type

4.5.4 Testing

All implementations are automatically tested against a lot of test cases with known results. This revealed bugs introduced during optimizations and ensures that the implementations handle a lot of cases correctly.

5 Performance

In this section, I will empirically validate the asymptotic running-time of the three algorithms and compare the constant factors. n and k will be used as defined in section 3.

5.1 Generation of Test Data

To test the dependence on n and k I had to be able to construct arrangements of n line segments with k intersections. As I did not want to construct them in a special way⁵ I used a random construction process as follows. I used a method $\text{gen}(n, p)$ that adds n segments to a list with approximately $p \cdot n(n-1)/2$ intersections. That is p controls the percentage of intersections. It does this by recursively calling itself. It then adds one random line and shortens this line to a segment to get the required number of intersections. This is done by computing the intersection points of all segments with that line and discarding points at both ends until the desired number of intersections is reached. This works well for $p < 0.3$ and was used to construct test-cases with a fixed k .

To test the dependence of the algorithms on parameters n and k , I created three different kinds of test-cases. In the first scenario, I varied n and chose k to be a fixed fraction of n . In the second scenario, n was varied again and k was chosen to be a fixed fraction of $n(n-1)/2$, the maximum number of intersections. In the third scenario, n was fixed and k was varied.

5.2 Results

Figure 1 shows the running time for increasing values of n with $k = 0.5 \cdot n$. As k depends on n only linear, the performance of both advanced algorithms becomes $O(n \log n)$. This is clearly visible in the results compared to the quadratic behavior of the simple algorithm. The logarithmic factor can not be observed because n is too small. The constant factor for the $n \log n$ part seems to be relatively small for both the Bentley-Ottmann and Mulmuley's algorithm.

The running times for letting k depend on n quadratically can be seen in figure 2. The running time of all algorithms is $O(n^2)$ now. The constant factor of the dependence on $k \log n$ in the Bentley-Ottmann algorithms seems to be considerable higher than the constant in the k term of Mulmuley's algorithm. This is probably due to the operations on the balanced binary search tree that have to be performed for every intersection. Note that the the simple algorithm

⁵the special structure of such a construction could favor one of the algorithms

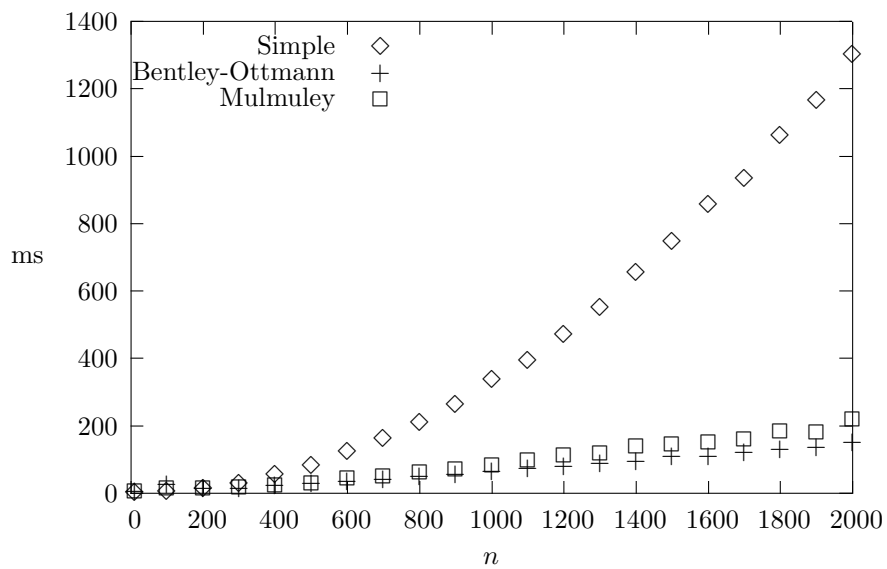


Figure 1: running time for $k = 0.5 \cdot n$

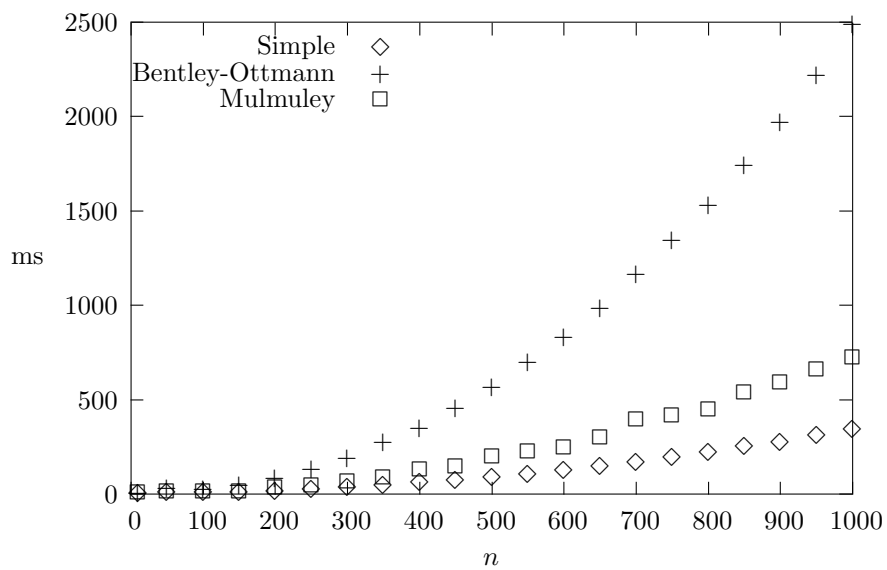


Figure 2: running time for $k = 0.1 \cdot n(n-1)/2$

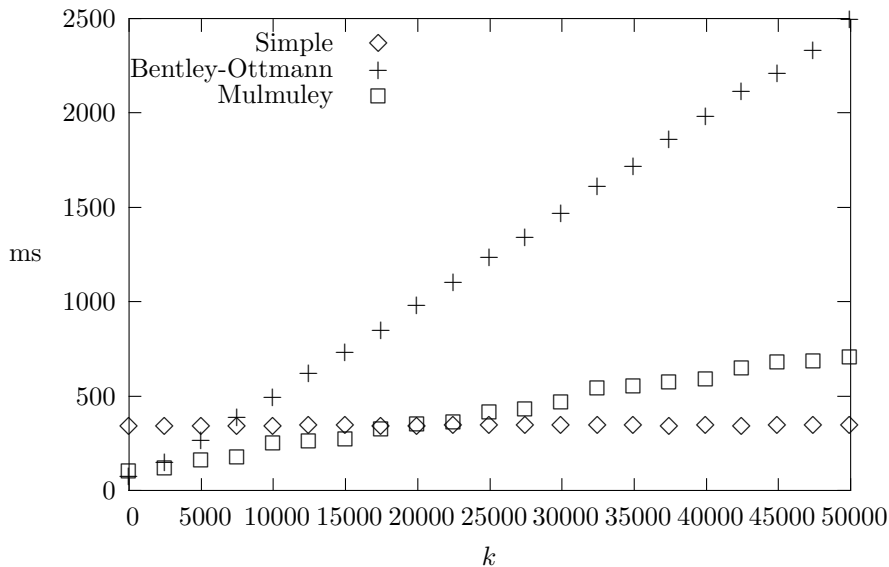


Figure 3: running time for $n = 1000$

is still faster by a small factor, although there are only 10% of the maximum number of intersections.

This phenomenon can be observed better in figure 3. It shows the running time for various values of k for a fixed value of n . It clearly shows that the simple algorithm is independent of k while the other two algorithms depend linearly on it. In fact, the Bentley-Ottmann algorithm has a $k \log n$ dependence, but this is not visible because n is fixed. This could, however, be one reason for the clearly better constants of Mulmuley's algorithm. The "break-even" point for Mulmuley's algorithm with respect to the linear algorithm seems to be at a fixed fraction of 0.03 of the maximum number of intersections. This assumption is confirmed by figure 4 where it occurs at the same fraction.

6 Discussion & Future Work

The performance analysis on section 5 shows, that even the low constant factors of Mulmuley's simple randomized algorithm are too big to use it in the current form for determining the number of line segment intersections for a complete graph. However, for graphs with fewer edges, it could be useful. Using the fraction of 0.03 of the maximum number of intersections obtained from the performance analysis, a local search algorithm could automatically choose the faster algorithm.

Future work could be done to improve the constant factors of the implementation as it is already very close to the simple algorithm for interesting values

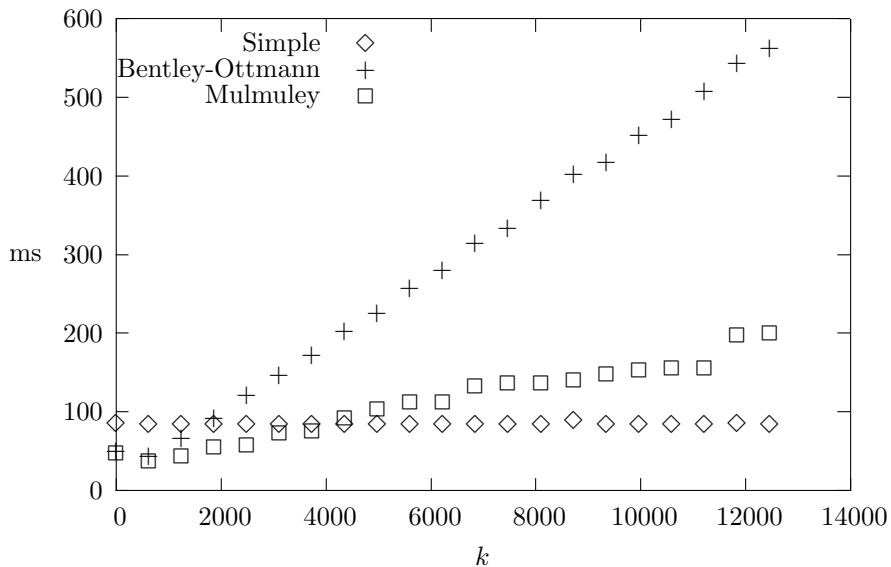


Figure 4: running time for $n = 500$

of k . Another approach would be to implement another randomized algorithm that uses trapezoidal decompositions [13]⁶. Although the constants will probably be higher because of the associated search structure necessary, one could exploit the fact that this algorithm is *on-line*. If, for example, the third local search strategy is used, one could build the trapezoidal decomposition of the first $n - 1$ vertices, save this state and repeatedly add a new vertex, checking only for intersections of this new vertex with the given set of vertices.

The underlying problem is similar to that of *red-blue* segment intersection. The set of blue segments is the set of edges between old vertices and — in contrast to the normal red-blue scenario — can intersect each other. The red set is the set of edges of the newly placed vertex. Obviously, the red segments do only intersect at their common endpoint. One is interested in the number of intersections between the red and blue segments. If, similar to the original red-blue segment intersection problem, an algorithm could be found that does not depend on k to *count* the number of intersections, this could greatly improve the performance of the local search step. However, I am not aware of such an algorithm.

7 Related Work

In [1] Andrew et. al compare the efficiency of different algorithm for line segment intersection. However, they focus on the red-blue segment intersection problem

⁶This is in fact the algorithm that we analyzed in assignment 2

and the application to geographic information systems.

A robust implementation of the Bentley-Ottmann algorithm is described in [10]. Its main concern is on robustness and it does not present empirical results.

References

- [1] D. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. Further comparisons of algorithms for geometric intersection problems, 1994.
- [2] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the eleventh annual symposium on Computational geometry*, pages 211–219. ACM Press, 1995.
- [3] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979.
- [4] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.
- [5] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4(1):387–421, 1989.
- [6] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Software Practice and Experience*, 30(11):1167–1202, 2000.
- [7] S. Kanthak. Implementation of line segment intersections algorithms in java. Available electronically at <http://www.muehlheim.de/~skanthak/university/lineseg.tar.gz>, 2003.
- [8] M. O. M. de Berg, M. van Kreveld and O. Schwarzkopf. *Computational Geometry*. Springer, 2 edition, 2000.
- [9] L. M. Maxwell Young, James Cook. Stochastic local search algorithms for minimizing edge crossings in complete rectilinear graphs. Published electronically at <http://www.cs.ubc.ca/labs/beta/Div/CPSC532D-WS-03/YouCooMah-final.pdf>, 04 2003.
- [10] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, 1994.
- [11] M.R.Garey and D.S.Johnson. Crossing number is np-complete. *SIAM Journal of Algebraic and Discrete Methods*, 4:312–316, 1983.

- [12] K. Mulmuley. A fast planar partition algorithm, i. In *Proc. 29th Annu. IEEE Sympos. Found. Comp. Sci.*, 10, pages 580–589, 1988.
- [13] K. Mulmuley. *Computational geometry: an introduction through randomized algorithms*. Prentice-Hall, 1994.
- [14] F. A. O. Aichholzer and H. Krasser. Published electronically at <http://www.cis.tugraz.at/igi/~oaich/publications.html>, 2002.