Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Русский
Edit links

Bentley–Ottmann algorithm

From Wikipedia, the free encyclopedia

In computational geometry, the **Bentley–Ottmann algorithm** is a sweep line algorithm for listing all crossings in a set of line segments. It extends the Shamos–Hoey algorithm,[1] a similar previous algorithm for testing whether or not a set of line segments has any crossings. For an input consisting of n line segments with k crossings, the Bentley–Ottmann algorithm takes time $O((n + k) \log n)$. In cases where $k = o(n^2 / \log n)$, this is an improvement on a naïve algorithm that tests every pair of segments, which takes $O(n^2)$.

The algorithm was initially developed by Jon Bentley and Thomas Ottmann (1979); it is described in more detail in the textbooks Preparata & Shamos (1985), O'Rourke (1998), and de Berg et al. (2000). Although asymptotically faster algorithms are now known, the Bentley–Ottman algorithm remains a practical choice due to its simplicity and low memory requirements.

Contents [hide]

Overall strategy [edit]

The main idea of the Bentley–Ottmann algorithm is to use a sweep line approach, in which a vertical line L moves from left to right across the plane, intersecting the input line segments in sequence as it moves.[2] The algorithm is described most easily in its general position, meaning:

1. No two line segment endpoints or crossings have the same x-coordinate
2. No line segment endpoint lies upon another line segment
3. No three line segments intersect at a single point.

In such a case, L will always intersect the input line segments in a set of points whose vertical ordering changes only at a finite set of discrete *events*. Thus, the continuous motion of L can be broken down into a finite sequence of steps, and simulated by an algorithm that runs in a finite amount of time.

There are two types of event that may happen during the course of this simulation. When L sweeps across an endpoint of a line segment s, the intersection of L with s is added to or removed from the vertically ordered set of intersection points. These events are easy to predict, as the endpoints are known already from the input to the algorithm. The remaining events occur when L sweeps across a crossing between two line segments s and t. These events may also be predicted from the fact that, just prior to the event, the points of intersection of L with s and t are adjacent in the vertical ordering of the intersection points.

The Bentley–Ottman algorithm itself maintains data structures representing the current vertical ordering of the intersection points of the sweep line with the input line segments, and a collection of potential future events formed by adjacent pairs of intersection points. It processes each event in turn, updating its data structures to represent the new set of intersection points.

Data structures [edit]

In order to efficiently maintain the intersection points of the sweep line L with the input line segments and the sequence of future events, the Bentley–Ottman algorithm maintains two data structures:

- A binary search tree, containing the set of input line segments that cross L, ordered by the y-coordinates of the points where these segments cross L. The crossing points themselves are not represented explicitly in the binary search tree. The Bentley–Ottman algorithm will insert a new segment s into this data structure when the sweep line *L* crosses the left endpoint *p* of this segment; the correct position of *s* in the binary search tree may be determined by a binary search, each step of which tests whether *p* is above or below some other segment that is crossed by *L*. Thus, an insertion may be performed in logarithmic time. The Bentley–Ottmann algorithm will also delete segments from the binary search tree, and use the binary search tree to determine the segments that are immediately above or below other segments; these operations may be performed using only the tree structure itself without reference to the underlying geometry of the segments.

- A priority queue (the "event queue"), used to maintain a sequence of potential future events in the Bentley–Ottmann algorithm. Each event is associated with a point *p* in the plane, either a segment endpoint or a crossing point, and the event happens when line *L* sweeps over *p*. Thus, the events may be prioritized by the *x*-coordinates of the points associated with each event. In the Bentley–Ottmann algorithm, the potential future events consist of line segment endpoints that have not yet been swept over, and the points of intersection of pairs of lines containing pairs of segments that are immediately above or below each other.

The algorithm does not need to maintain explicitly a representation of the sweep line *L* or its position in the plane. Rather, the position of *L* is represented indirectly: it is the vertical line through the point associated with the most recently processed event.

The binary search tree may be any balanced binary search tree data structure, such as a red-black tree; all that is required is that insertions, deletions, and searches take logarithmic time. Similarly, the priority queue may be a binary heap or any other logarithmic-time priority queue; more sophisticated priority queues such as a Fibonacci heap are not necessary.

Detailed algorithm [edit]

The Bentley–Ottmann algorithm performs the following steps.

1. Initialize a priority queue *Q* of potential future events, each associated with a point in the plane and prioritized by the *x*-coordinate of the point. Initially, *Q* contains an event for each of the endpoints of the input segments.
2. Initialize a binary search tree *T* of the line segments that cross the sweep line *L*, ordered by the *y*-coordinates of the crossing points. Initially, *T* is empty.
3. While *Q* is nonempty, find and remove the event from *Q* associated with a point *p* with minimum *x*-coordinate. Determine what type of event this is and process it according to the following case analysis:
   - If *p* is the left endpoint of a line segment *s*, insert *s* into *T*. Find the segments *r* and *t* that are immediately below and above *s* in *T* (if they exist) and if their crossing forms a potential future event in the event queue, remove it. If *s* crosses *r* or *t*, add those crossing points as potential future events in the event queue.
   - If *p* is the right endpoint of a line segment *s*, remove *s* from *T*. Find the segments *r* and *t* that were (prior to the removal of *s*) immediately above and below it in *T* (if they exist). If *r* and *t* cross, add that crossing point as a potential future event in the event queue.
   - If *p* is the crossing point of two segments *s* and *t* (with *s* below *t* to the left of the crossing), swap the positions of *s* and *t* in *T*. Find the segments *r* and *u* (if they exist) that are immediately below and above *t* and *s* respectively (after the swap). Remove any crossing points *rs* and *tu* from the event queue, and, if *r* and *t* cross or *s* and *u* cross, add those crossing points to the event queue.

Analysis [edit]

The algorithm processes one event per segment endpoint or crossing point, in the sorted order of the *x*-coordinates of these points, as may be proven by induction. This follows because, once the *i*th event has been processed, the next event (if it is a crossing point) must be a crossing of two segments that are adjacent in the ordering of the segments represented by *T*, and because the algorithm maintains all crossings between adjacent segments as potential future events in the event queue; therefore, the correct next event will always be present in the event queue. As a consequence, it correctly finds all crossings of input line segments, the problem it was designed to solve.

The Bentley–Ottmann algorithm processes a sequence of $2n + k$ events, where *n* denotes the number of input line segments and *k* denotes the number of crossings. Each event is processed by a constant number of operations in the binary search tree and the event queue, and (because it contains only segment endpoints and crossings between adjacent segments) the event queue never contains more than $3n$ events. Thus, all operations take time $O(\log n)$ and the total time for the algorithm is $O((n + k) \log n)$.

If the crossings found by the algorithm do not need to be stored once they have been found, the space used by the algorithm at any point in time is O(*n*): each of the *n* input line segments corresponds to at most one node of the binary search tree *T*, and as stated above the event queue contains at most 3*n* elements. This space bound is due to Brown (1981); the original version of the algorithm was slightly different (it did not remove crossing events from *Q* when some other event causes the two crossing segments to become non-adjacent) causing it to use more space.[3]

Chen & Chan (2003) described a highly space-efficient version of the Bentley–Ottman algorithm that encodes most of its information in the ordering of the segments in an array representing the input, requiring only O($\log^2 n$) additional memory cells. However, in order to access the encoded information, the algorithm is slowed by a logarithmic factor.

Special position and numerical precision issues [edit]

The algorithm description above assumes that line segments are not vertical, that line segment endpoints do not lie on other line segments, that crossings are formed by only two line segments, and that no two event points have the same *x*-coordinate. However, these general position assumptions are not reasonable for most applications of line segment intersection. Bentley & Ottmann (1979) suggested perturbing the input slightly to avoid these kinds of numerical coincidences, but did not describe in detail how to perform these perturbations. de Berg et al. (2000) describe in more detail the following measures for handling special-position inputs:

- Break ties between event points with the same *x*-coordinate by using the *y*-coordinate. Events with different *y*-coordinates are handled as before. This modification handles both the problem of multiple event points with the same *x*-coordinate, and the problem of vertical line segments: the left endpoint of a vertical segment is defined to be the one with the lower *y*-coordinate, and the steps needed to process such a segment are essentially the same as those needed to process a non-vertical segment with a very high slope.
- Define a line segment to be a closed set, containing its endpoints. Therefore, two line segments that share an endpoint, or a line segment that contains an endpoint of another segment, both count as an intersection of two line segments.
- When multiple line segments intersect at the same point, create and process a single event point for that intersection. The updates to the binary search tree caused by this event may involve removing any line segments for which this is the right endpoint, inserting new line segments for which this is the left endpoint, and reversing the order of the remaining line segments containing this event point. The output from the version of the algorithm described by de Berg et al. (2000) consists of the set of intersection points of line segments, labeled by the segments they belong to, rather than the set of pairs of line segments that intersect.

A similar approach to degeneracies was used in the LEDA implementation of the Bentley–Ottmann algorithm.[4]

For the correctness of the algorithm, it is necessary to determine without approximation the above-below relations between a line segment endpoint and other line segments, and to correctly prioritize different event points. For this reason it is standard to use integer coordinates for the endpoints of the input line segments, and to represent the rational number coordinates of the intersection points of two segments exactly, using arbitrary-precision arithmetic. However, it may be possible to speed up the calculations and comparisons of these coordinates by using floating point calculations and testing whether the values calculated in this way are sufficiently far from zero that they may be used without any possibility of error.[4] The exact arithmetic calculations required by a naïve implementation of the Bentley–Ottmann algorithm may require five times as many bits of precision as the input coordinates, but Boissonat & Preparata (2000) describe modifications to the algorithm that reduce the needed amount of precision to twice the number of bits as the input coordinates.

Faster algorithms [edit]

The O(*n* log *n*) part of the time bound for the Bentley–Ottmann algorithm is necessary, as there are matching lower bounds for the problem of detecting intersecting line segments in algebraic decision tree models of computation.[5] However, the dependence on *k*, the number of crossings, can be improved. Clarkson (1988) and Mulmuley (1988) both provided randomized algorithms for constructing the planar graph whose vertices are endpoints and crossings of line segments, and whose edges are the portions of the segments connecting these vertices, in expected time O(*n* log *n* + *k*), and this problem of arrangement construction was solved deterministically in the same O(*n* log *n* + *k*) time bound by Chazelle & Edelsbrunner (1992). However, constructing this arrangement as a whole requires space O(*n* + *k*), greater than the O(*n*) space bound of the Bentley–Ottmann algorithm; Balaban (1995) described a different algorithm that lists all intersections in time O(*n* log *n* + *k*) and space O(*n*).

If the input line segments and their endpoints form the edges and vertices of a connected graph (possibly with crossings), the O(n log n) part of the time bound for the Bentley–Ottmann algorithm may also be reduced. As Clarkson, Cole & Tarjan (1992) show, in this case there is a randomized algorithm for solving the problem in expected time O(n log* n + k), where log* denotes the iterated logarithm, a function much more slowly growing than the logarithm. A closely related randomized algorithm of Eppstein, Goodrich & Strash (2009) solves the same problem in time O(n + k $\log^{(i)}n$) for any constant i, where $\log^{(i)}$ denotes the function obtained by iterating the logarithm function i times. The first of these algorithms takes linear time whenever k is larger than n by a $\log^{(i)}n$ factor, for any constant i, while the second algorithm takes linear time whenever k is smaller than n by a $\log^{(i)}n$ factor. Both of these algorithms involve applying the Bentley–Ottmann algorithm to small random samples of the input.

Notes [edit]

1. ^ Shamos, M. I.; Hoey, D. (1976). "17th Annual Symposium on Foundations of Computer Science (sfcs 1976)" (PDF). p. 208. doi:10.1109/SFCS.1976.16. |chapter= ignored (help)
2. ^ In the description of the algorithm in de Berg et al. (2000), the sweep line is horizontal and moves vertically; this change entails swapping the use of *x*- and *y*-coordinates consistently throughout the algorithm, but is not otherwise of great significance for the description or analysis of the algorithm.
3. ^ The nonlinear space complexity of the original version of the algorithm was analyzed by Pach & Sharir (1991).
4. ^ [a] [b] Bartuschka, Mehlhorn & Näher (1997).
5. ^ Preparata & Shamos (1985), Theorem 7.6, p. 280.

References [edit]

- Balaban, I. J. (1995), "An optimal algorithm for finding segments intersections", *Proc. 11th ACM Symp. Computational Geometry*, pp. 211–219, doi:10.1145/220279.220302.
- Bartuschka, U.; Mehlhorn, K.; Näher, S. (1997), "A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem", in Italiano, G. F.; Orlando, S., *[http://www.dsi.unive.it/~wae97/proceedings/ Proc. Worksh. Algorithm Engineering]*.
- Bentley, J. L.; Ottmann, T. A. (1979), "Algorithms for reporting and counting geometric intersections", *IEEE Transactions on Computers* **C–28** (9): 643–647, doi:10.1109/TC.1979.1675432.
- de Berg, Mark; van Kreveld, Marc; Overmars, Mark; Schwarzkopf, Otfried (2000), "Chapter 2: Line segment intersection", *Computational Geometry* (2nd ed.), Springer-Verlag, pp. 19–44, ISBN 978-3-540-65620-3.
- Boissonat, J.-D.; Preparata, F. P. (2000), "Robust plane sweep for intersecting segments" (PDF), *SIAM Journal on Computing* **29** (5): 1401–1421, doi:10.1137/S0097539797329373.
- Brown, K. Q. (1981), "Comments on "Algorithms for Reporting and Counting Geometric Intersections"", *IEEE transactions on Computers* **C–30** (2): 147.
- Chazelle, Bernard; Edelsbrunner, Herbert (1992), "An optimal algorithm for intersecting line segments in the plane", *Journal of the ACM* **39** (1): 1–54, doi:10.1145/147508.147511.
- Chen, E. Y.; Chan, T. M. (2003), "A space-efficient algorithm for segment intersection", *Proc. 15th Canadian Conference on Computational Geometry* (PDF).
- Clarkson, K. L. (1988), "Applications of random sampling in computational geometry, II", *Proc. 4th ACM Symp. Computational Geometry*, pp. 1–11, doi:10.1145/73393.73394.
- Clarkson, K. L.; Cole, R.; Tarjan, R. E. (1992), "Randomized parallel algorithms for trapezoidal diagrams", *International Journal of Computational Geometry and Applications* **2** (2): 117–133, doi:10.1142/S0218195992000081. Corrigendum, **2** (3): 341–343.
- Eppstein, D.; Goodrich, M.; Strash, D. (2009), "Linear-time algorithms for geometric graphs with sublinearly many crossings", *Proc. 20th ACM-SIAM Symp. Discrete Algorithms (SODA 2009)*, pp. 150–159, arXiv:0812.0893.
- Mulmuley, K. (1988), "A fast planar partition algorithm, I", *Proc. 29th IEEE Symp. Foundations of Computer Science (FOCS 1988)*, pp. 580–589, doi:10.1109/SFCS.1988.21974.
- O'Rourke, J. (1998), "Section 7.7: Intersection of segments", *Computational Geometry in C* (2nd ed.), Cambridge University Press, pp. 263–265, ISBN 978-0-521-64976-6.
- Preparata, F. P.; Shamos, M. I. (1985), "Section 7.2.3: Intersection of line segments", *Computational Geometry: An Introduction*, Springer-Verlag, pp. 278–287.
- Pach, J.; Sharir, M. (1991), "On vertical visibility in arrangements of segments and the queue size in the Bentley–Ottmann line sweeping algorithm", *SIAM Journal on Computing* **20** (3): 460–470, doi:10.1137/0220029, MR 1094525.
- Shamos, M. I.; Hoey, Dan (1976), "Geometric intersection problems", *17th IEEE Conf. Foundations of Computer Science (FOCS 1976)*, pp. 208–215, doi:10.1109/SFCS.1976.16.

External links [edit]

- Smid, Michiel (2003), *Computing intersections in a set of line segments: the Bentley–Ottmann algorithm* (PDF).

Categories: Computational geometry