Line Sweep Algorithms

Robin Visser

IOI Training Camp
University of Cape Town

23 May 2015

Overview

Background

- A line sweep algorithm is one that uses a conceptual *sweep line* to solve various problems in Euclidean space.
- The basic idea is that one imagines a line swept across the plane, stopping at certain points, whilst doing geometric operations on points in the immediate vicinity of the sweep line.
- Usually, the complete solution is available once the line has passed over all objects.

Background

- A line sweep algorithm is one that uses a conceptual *sweep line* to solve various problems in Euclidean space.
- The basic idea is that one imagines a line swept across the plane, stopping at certain points, whilst doing geometric operations on points in the immediate vicinity of the sweep line.
- Usually, the complete solution is available once the line has passed over all objects.

Background

- A line sweep algorithm is one that uses a conceptual *sweep line* to solve various problems in Euclidean space.
- The basic idea is that one imagines a line swept across the plane, stopping at certain points, whilst doing geometric operations on points in the immediate vicinity of the sweep line.
- Usually, the complete solution is available once the line has passed over all objects.

Closest pair problem

Problem

Given a set of n points, find a pair of points with the smallest distance between them

- Of course, one can do a brute force algorithm in $O(n^2)$ time.
- A line sweep algorithm can reduce this to $O(n \log n)$.

Closest pair problem

Problem

Given a set of n points, find a pair of points with the smallest distance between them

- Of course, one can do a brute force algorithm in $O(n^2)$ time.
- A line sweep algorithm can reduce this to $O(n \log n)$.

Closest pair problem

Problem

Given a set of n points, find a pair of points with the smallest distance between them

- Of course, one can do a brute force algorithm in $O(n^2)$ time.
- A line sweep algorithm can reduce this to $O(n \log n)$.

Algorithm

- Sort all points by their x-coordinate and keep track of best distance so far as h
- Suppose we've processed points 1 to $k-1$:
- We process point k and maintain a set of already processed points whose x coordinates and within h of point k.
- We add the point being processed to the set and remove points from the set when we move one (or when h is decreased)
- The set itself is ordered by y-coordinate.
- We then simply check for points within the range: $y_k - h$ to $y_k + h$ and update h if a new best is found.

Algorithm

- Sort all points by their x-coordinate and keep track of best distance so far as h
- Suppose we've processed points 1 to $k-1$:
- We process point k and maintain a set of already processed points whose x coordinates and within h of point k.
- We add the point being processed to the set and remove points from the set when we move one (or when h is decreased)
- The set itself is ordered by y-coordinate.
- We then simply check for points within the range: $y_k - h$ to $y_k + h$ and update h if a new best is found.

Algorithm

- Sort all points by their x-coordinate and keep track of best distance so far as h
- Suppose we've processed points 1 to $k-1$:
- We process point k and maintain a set of already processed points whose x coordinates and within h of point k.
- We add the point being processed to the set and remove points from the set when we move one (or when h is decreased)
- The set itself is ordered by y-coordinate.
- We then simply check for points within the range: $y_k - h$ to $y_k + h$ and update h if a new best is found.

Algorithm

- Sort all points by their x-coordinate and keep track of best distance so far as h
- Suppose we've processed points 1 to $k-1$:
- We process point k and maintain a set of already processed points whose x coordinates and within h of point k.
- We add the point being processed to the set and remove points from the set when we move one (or when h is decreased)
- The set itself is ordered by y-coordinate.
- We then simply check for points within the range: $y_k - h$ to $y_k + h$ and update h if a new best is found.

Algorithm

- Sort all points by their x-coordinate and keep track of best distance so far as h
- Suppose we've processed points 1 to $k-1$:
- We process point k and maintain a set of already processed points whose x coordinates and within h of point k.
- We add the point being processed to the set and remove points from the set when we move one (or when h is decreased)
- The set itself is ordered by y-coordinate.
- We then simply check for points within the range: $y_k - h$ to $y_k + h$ and update h if a new best is found.

Algorithm

- Sort all points by their x-coordinate and keep track of best distance so far as h
- Suppose we've processed points 1 to $k-1$:
- We process point k and maintain a set of already processed points whose x coordinates and within h of point k.
- We add the point being processed to the set and remove points from the set when we move one (or when h is decreased)
- The set itself is ordered by y-coordinate.
- We then simply check for points within the range: $y_k - h$ to $y_k + h$ and update h if a new best is found.

Code

Pseudocode:

```
sort(p, p+n, xcomp)
set<ycomp> box
box.insert(p[0])
int left = 0
for i in range(1, n):
    while (left < i) and (p[i].x - p[left].x > h):
        box.erase(p[left])
        left++
    for (it = box[p[i].y-h], it < box[p[i].y+h]):
        h = min(h, dist(p[i], it))
    box.insert(p[i])
return h
```

Time Complexity

- For each point, removal from the set is $O(\log n)$. Each point is only removed once, giving $O(n \log n)$ in total for removal.
- We can extract the required range from the set in $O(\log n)$ time (for C++, one can use the lower_bound function on a set)
- Within this range there can only be $O(1)$ elements, since any two points in the set has distance at least h.
- The search for each point is therefore at most $O(\log n)$, giving a total time of $O(n \log n)$

Time Complexity

- For each point, removal from the set is $O(\log n)$. Each point is only removed once, giving $O(n \log n)$ in total for removal.
- We can extract the required range from the set in $O(\log n)$ time (for C++, one can use the lower_bound function on a set)
- Within this range there can only be $O(1)$ elements, since any two points in the set has distance at least h.
- The search for each point is therefore at most $O(\log n)$, giving a total time of $O(n \log n)$

Time Complexity

- For each point, removal from the set is $O(\log n)$. Each point is only removed once, giving $O(n \log n)$ in total for removal.
- We can extract the required range from the set in $O(\log n)$ time (for C++, one can use the lower_bound function on a set)
- Within this range there can only be $O(1)$ elements, since any two points in the set has distance at least h.
- The search for each point is therefore at most $O(\log n)$, giving a total time of $O(n \log n)$

Time Complexity

- For each point, removal from the set is $O(\log n)$. Each point is only removed once, giving $O(n \log n)$ in total for removal.
- We can extract the required range from the set in $O(\log n)$ time (for C++, one can use the lower_bound function on a set)
- Within this range there can only be $O(1)$ elements, since any two points in the set has distance at least h.
- The search for each point is therefore at most $O(\log n)$, giving a total time of $O(n \log n)$

Line segment intersection

Problem

Given a set of n vertical and horizontal line segments, report all intersection points among them

- Again, a trivial brute force algorithm runs in $O(n^2)$ time.
- A line sweep algorithm can reduce this to $O(n \log n)$.

Line segment intersection

Problem

Given a set of n vertical and horizontal line segments, report all intersection points among them

- Again, a trivial brute force algorithm runs in $O(n^2)$ time.
- A line sweep algorithm can reduce this to $O(n \log n)$.

Line segment intersection

Problem

Given a set of n vertical and horizontal line segments, report all intersection points among them

- Again, a trivial brute force algorithm runs in $O(n^2)$ time.
- A line sweep algorithm can reduce this to $O(n \log n)$.

Algorithm

- Make a sorted list of all the x-coordinates at which some event happens (either a vertical line, or start or end of a horizontal line)
- We iterate through the events:
- We keep a set of current horizontal lines (sorted by y-coordinate). We simply add or remove from the set whenever we hit the start or end of a horizontal line.
- When we come across a vertical line, we do a range search in our set to obtain intersections.

Algorithm

- Make a sorted list of all the x-coordinates at which some event happens (either a vertical line, or start or end of a horizontal line)
- We iterate through the events:
- We keep a set of current horizontal lines (sorted by y-coordinate). We simply add or remove from the set whenever we hit the start or end of a horizontal line.
- When we come across a vertical line, we do a range search in our set to obtain intersections.

Algorithm

- Make a sorted list of all the x-coordinates at which some event happens (either a vertical line, or start or end of a horizontal line)
- We iterate through the events:
- We keep a set of current horizontal lines (sorted by y-coordinate). We simply add or remove from the set whenever we hit the start or end of a horizontal line.
- When we come across a vertical line, we do a range search in our set to obtain intersections.

Algorithm

- Make a sorted list of all the x-coordinates at which some event happens (either a vertical line, or start or end of a horizontal line)
- We iterate through the events:
- We keep a set of current horizontal lines (sorted by y-coordinate). We simply add or remove from the set whenever we hit the start or end of a horizontal line.
- When we come across a vertical line, we do a range search in our set to obtain intersections.

Code

Pseudocode:

```
sort(events, events+e)
set s
for i in range(e):
    c = events[i]
    if c is starting point:
        s.insert(c.p1)
    else if end point:
        s.erase(c.p2)
    else:
        for (it = c.p1, it < c.p2):
            #Intersection at c and s[it]
```

Time Complexity

- Using a set, insertion and removal is done in $O(\log n)$ time.
- Finding the required range is $O(\log n)$, plus an additional $O(I)$ to note the I intersections.
- Total time is $O(n \log n + I)$ time for I intersections. Just counting the intersections can be done in $O(n \log n)$ time using an augmented binary tree structure (store number of nodes)
- Algorithm can be generalised for arbitrary line segments (use a priority queue to handle events)

Time Complexity

- Using a set, insertion and removal is done in $O(\log n)$ time.
- Finding the required range is $O(\log n)$, plus an additional $O(I)$ to note the I intersections.
- Total time is $O(n \log n + I)$ time for I intersections. Just counting the intersections can be done in $O(n \log n)$ time using an augmented binary tree structure (store number of nodes)
- Algorithm can be generalised for arbitrary line segments (use a priority queue to handle events)

Time Complexity

- Using a set, insertion and removal is done in $O(\log n)$ time.
- Finding the required range is $O(\log n)$, plus an additional $O(I)$ to note the I intersections.
- Total time is $O(n \log n + I)$ time for I intersections. Just counting the intersections can be done in $O(n \log n)$ time using an augmented binary tree structure (store number of nodes)
- Algorithm can be generalised for arbitrary line segments (use a priority queue to handle events)

Time Complexity

- Using a set, insertion and removal is done in $O(\log n)$ time.
- Finding the required range is $O(\log n)$, plus an additional $O(I)$ to note the I intersections.
- Total time is $O(n \log n + I)$ time for I intersections. Just counting the intersections can be done in $O(n \log n)$ time using an augmented binary tree structure (store number of nodes)
- Algorithm can be generalised for arbitrary line segments (use a priority queue to handle events)

Area of the union of rectangles

Problem

Given a set of rectangles, calculate the area of its union.

- Again, we use a line sweep, keeping track of the important events with an active set.

Area of the union of rectangles

Problem

Given a set of rectangles, calculate the area of its union.

- Again, we use a line sweep, keeping track of the important events with an active set.

Algorithm

- As before, we keep a sorted list of events which we iterate through, namely the left and right edge of our rectangles.
- When we cross a left edge, the rectangle is added to the set, and removed once we cross over the right edge.
- We then do another line sweep running top-down within our active set to determine the total length of the main line sweep which is cut by rectangles.
- Multiplying this by the difference in x-coordinates when we step to the next event, and totalling, gives us the total union area.
- Using a Boolean array to store our active set will result in $O(n^2)$ time. This can be improved to $O(n \log n)$ time using binary tree manipulation tricks within the inner loop.

Algorithm

Line Sweep Algorithms

Robin Visser

Background

Examples

Closest Pair

Line Segments

Union of rectangles

Convex hull

Summary

- As before, we keep a sorted list of events which we iterate through, namely the left and right edge of our rectangles.
- When we cross a left edge, the rectangle is added to the set, and removed once we cross over the right edge.
- We then do another line sweep running top-down within our active set to determine the total length of the main line sweep which is cut by rectangles.
- Multiplying this by the difference in x-coordinates when we step to the next event, and totalling, gives us the total union area.
- Using a Boolean array to store our active set will result in $O(n^2)$ time. This can be improved to $O(n \log n)$ time using binary tree manipulation tricks within the inner loop.

Algorithm

- As before, we keep a sorted list of events which we iterate through, namely the left and right edge of our rectangles.
- When we cross a left edge, the rectangle is added to the set, and removed once we cross over the right edge.
- We then do another line sweep running top-down within our active set to determine the total length of the main line sweep which is cut by rectangles.
- Multiplying this by the difference in x-coordinates when we step to the next event, and totalling, gives us the total union area.
- Using a Boolean array to store our active set will result in $O(n^2)$ time. This can be improved to $O(n \log n)$ time using binary tree manipulation tricks within the inner loop.

Algorithm

- As before, we keep a sorted list of events which we iterate through, namely the left and right edge of our rectangles.
- When we cross a left edge, the rectangle is added to the set, and removed once we cross over the right edge.
- We then do another line sweep running top-down within our active set to determine the total length of the main line sweep which is cut by rectangles.
- Multiplying this by the difference in x-coordinates when we step to the next event, and totalling, gives us the total union area.
- Using a Boolean array to store our active set will result in $O(n^2)$ time. This can be improved to $O(n \log n)$ time using binary tree manipulation tricks within the inner loop.

Algorithm

- As before, we keep a sorted list of events which we iterate through, namely the left and right edge of our rectangles.
- When we cross a left edge, the rectangle is added to the set, and removed once we cross over the right edge.
- We then do another line sweep running top-down within our active set to determine the total length of the main line sweep which is cut by rectangles.
- Multiplying this by the difference in x-coordinates when we step to the next event, and totalling, gives us the total union area.
- Using a Boolean array to store our active set will result in $\mathrm{O}(n^2)$ time. This can be improved to $\mathrm{O}(n \log n)$ time using binary tree manipulation tricks within the inner loop.

Convex hull

Problem

Given a set of n points in the plane, find the convex hull.

- A brute force will run in $O(n^2)$ time.
- A Graham scan is faster. Does sorting by angle and runs in $O(n \log n)$ time.
- Can be expensive to compute angles and may get numeric errors.
- A simpler solution is to simply sort by x-coordinate and sweep line. (Andrew's algorithm)

Convex hull

Problem

Given a set of n points in the plane, find the convex hull.

- A brute force will run in $O(n^2)$ time.
- A Graham scan is faster. Does sorting by angle and runs in $O(n \log n)$ time.
- Can be expensive to compute angles and may get numeric errors.
- A simpler solution is to simply sort by x-coordinate and sweep line. (Andrew's algorithm)

Convex hull

Problem

Given a set of n points in the plane, find the convex hull.

- A brute force will run in $O(n^2)$ time.
- A Graham scan is faster. Does sorting by angle and runs in $O(n \log n)$ time.
- Can be expensive to compute angles and may get numeric errors.
- A simpler solution is to simply sort by x-coordinate and sweep line. (Andrew's algorithm)

Convex hull

Problem

Given a set of n points in the plane, find the convex hull.

- A brute force will run in $O(n^2)$ time.
- A Graham scan is faster. Does sorting by angle and runs in $O(n \log n)$ time.
- Can be expensive to compute angles and may get numeric errors.
- A simpler solution is to simply sort by x-coordinate and sweep line. (Andrew's algorithm)

Convex hull

Problem

Given a set of n points in the plane, find the convex hull.

- A brute force will run in $O(n^2)$ time.
- A Graham scan is faster. Does sorting by angle and runs in $O(n \log n)$ time.
- Can be expensive to compute angles and may get numeric errors.
- A simpler solution is to simply sort by x-coordinate and sweep line. (Andrew's algorithm)

Algorithm

- We compute the convex hull in two parts: the upper hull and the lower hull.
- For the upper hull, we sort all points by their x-coordinate and incrementally add points in sorted order, building up the hull.
- We keep track of the last three points. If it is concave, we discard the second last point (we know it's not on the hull) and repeat the process.
- This is essentially the same procedure as a Graham scan, except sorted by x-coordinate instead of angle.
- The lower hull is done in a similar manner.

Algorithm

- We compute the convex hull in two parts: the upper hull and the lower hull.
- For the upper hull, we sort all points by their x-coordinate and incrementally add points in sorted order, building up the hull.
- We keep track of the last three points. If it is concave, we discard the second last point (we know it's not on the hull) and repeat the process.
- This is essentially the same procedure as a Graham scan, except sorted by x-coordinate instead of angle.
- The lower hull is done in a similar manner.

Algorithm

- We compute the convex hull in two parts: the upper hull and the lower hull.
- For the upper hull, we sort all points by their x-coordinate and incrementally add points in sorted order, building up the hull.
- We keep track of the last three points. If it is concave, we discard the second last point (we know it's not on the hull) and repeat the process.
- This is essentially the same procedure as a Graham scan, except sorted by x-coordinate instead of angle.
- The lower hull is done in a similar manner.

Algorithm

- We compute the convex hull in two parts: the upper hull and the lower hull.
- For the upper hull, we sort all points by their x-coordinate and incrementally add points in sorted order, building up the hull.
- We keep track of the last three points. If it is concave, we discard the second last point (we know it's not on the hull) and repeat the process.
- This is essentially the same procedure as a Graham scan, except sorted by x-coordinate instead of angle.
- The lower hull is done in a similar manner.

Algorithm

- We compute the convex hull in two parts: the upper hull and the lower hull.
- For the upper hull, we sort all points by their x-coordinate and incrementally add points in sorted order, building up the hull.
- We keep track of the last three points. If it is concave, we discard the second last point (we know it's not on the hull) and repeat the process.
- This is essentially the same procedure as a Graham scan, except sorted by x-coordinate instead of angle.
- The lower hull is done in a similar manner.

Code

Pseudocode:

```
sort(points, points+n, xcomp)
u = [], l = []
for i in range(1, n):
    while l[-2], l[-1], points[i] makes CW turn:
        l.pop
    l.append(points[i])

for i in range(n, 1):
    while u[-2], u[-1], points[i] makes CW turn:
        u.pop
    u.append(points[i])

return concat(l, u)
```

Summary

- Line sweep algorithms can be extremely powerful and can be used to solve a variety of problems.
- There are numerous other more advanced problems such as Delaunay triangulations and minimum spanning trees for certain metrics that can be solved using line sweep techniques.

Summary

- Line sweep algorithms can be extremely powerful and can be used to solve a variety of problems.
- There are numerous other more advanced problems such as Delaunay triangulations and minimum spanning trees for certain metrics that can be solved using line sweep techniques.