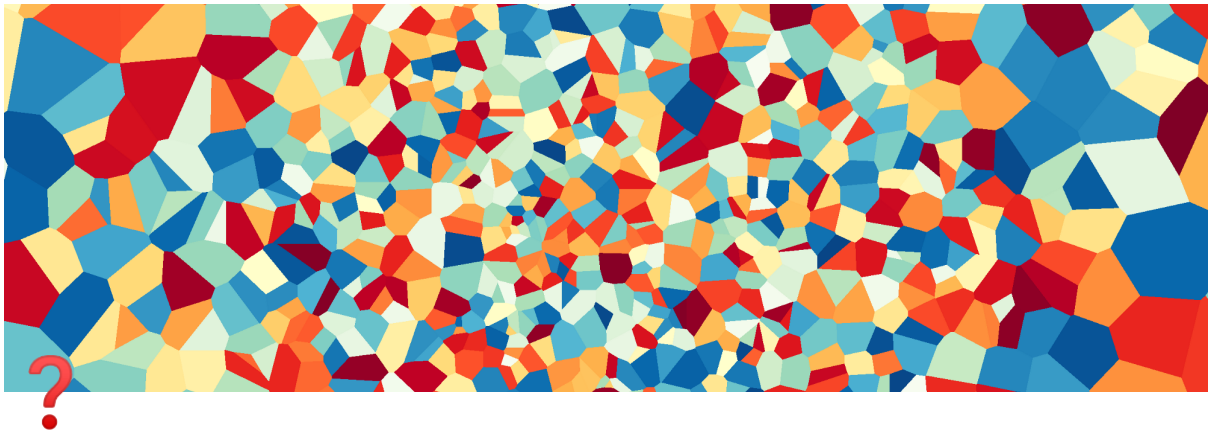


Study Questions

Algorithmic Geometry

Block 1

Convex Hull

▼ What is a convex hull of a set of points?

A region, s.t. every line connecting two points in the set is contained in the region.

▼ How can you naively compute a convex hull?

Iterate over all pairs of points, check if all points lie strictly right of it. Runtime in $O(n^3)$

▼ What better algorithms are there for computing convex hulls?

Graham Scan, Gift Wrapping/Jarvis March, Chans Algorithm

▼ How does Graham Scan work?

Points are ordered by their x-coordinate, the upper and lower convex hull are calculated separately in an iterative fashion. For the upper hull, we start with the first three points and check whether they form a right turn. If they don't, the second to last point is removed. After that, the next point is added, and the check is repeated. This is done until the last point is reached.

▼ What runtime does Grahams Scan have?

$O(n \log n)$

▼ How can you derive this runtime?

Sorting of the points $O(n \log n)$. Right-Turn-Check is constant, performed n times, so the sorting dominates.

▼ How does Gift Wrapping work?

The rightmost point (which is definitely part of the convex hull) $p := p^* = (x, y)$ is determined and a helper point $p' := (x, \infty)$ is added. In an iterative fashion going through all points, the point q maximizing the angle $\measuredangle p'pq$ is found. This point is added to the output. Then we set $p' := p$, $p := q$ and continue. We do this, until we find $q = p^*$, where we terminate.

▼ What runtime does Gift Wrapping have?

$O(nh)$, where h is the number of points of the convex hull (output-sensitive runtime)

▼ How can you derive this runtime?

For each point of the convex hull, we have to check $O(n)$ points to find the one that maximizes the angle.

▼ How do those two algorithms compare?

It depends on the size of the convex hull. In the case that all points are on the convex hull, Grahams Scan with $O(n \log n)$ is better than Gift Wrapping with $O(n^2)$. If the convex hull is just a triangle, Gift Wrapping with $O(3n) = O(n)$ outperforms Grahams Scan.

▼ How does Chans Algorithm work?

Chans Algorithm combines the two approaches. However, we need to know the number of points h in the hull beforehand. The set of points is randomly divided into sets of size h. For each set, Graham Scan is used to compute the hull of this subset. Then, Gift Wrapping is used to compute the full convex hull — however, only points that are in partial hulls have to be considered, thus reducing the number of points to be checked.

▼ What runtime does Chans Algorithm have?

The calculation of the partial hulls takes $O(h \log h)$ each. We have to do this $O(\frac{n}{h})$ times, which gives us a total runtime of $O(n \log h)$ for the first part. The second part iterates over $O(\frac{n}{h})$ convex hulls, each containing at most $O(h)$ points → $O(n)$. For finding the point maximizing the angle, we can make use of the partial hulls. For each partial hull, only the two tangent points are possible. Those can be found in a binary search in $O(\log h)$, as we have the points on the partial hulls in order. This results in a runtime of $O(n \log h)$.

▼ How can we use Chans Algorithm without knowing h?

We try to find h using double exponential search — the algorithm is invoked with $\min\{n, 2^{2^t}\}$ for $t = 0, 1, 2, ...$ and it is modified to return invalid if no correct hull could be found.

▼ Why does this still yield the desired runtime of $O(n \log h)$?

$$\sum_{t=0}^{\lceil \log \log h \rceil} O(n \log 2^{2^t}) = O(n) \cdot \sum_{t=0}^{\lceil \log \log h \rceil} O(2^t) \leq O(n) \cdot O(2^{\log \log h}) = O(n) \cdot O(\log h) = O(n \log h)$$

▼ Is it possible to compute a convex hull faster than $O(n \log n)$ or $O(n \log h)$?

No, because one could use this algorithm to sort numbers, for which we know the lower bound of $\Omega(n \log n)$.

▼ What happens in Graham Scan when the sorting of P is not unique?

Use lexicographic order

▼ How do the algorithms deal with collinear points?

- Graham Scan: do not form a right turn, thus interior point removed
- Gift Wrapping: all collinear points have the same angle, to fulfill the algorithmic definition from the lecture, we have to choose the farthest point

▼ What about the robustness of the algorithms? [TODO]

Line Segment Intersections

▼ How can you naively find all intersections of line segments?

Test all pairs of segments. For n segments this gives us a runtime of $O(n^2)$.

▼ When is this already optimal?

If the number of intersections is in $O(n^2)$.

▼ What technique can you use to compute intersections faster?

We only test pairs that are sufficiently close to each other → sweep line algorithm

▼ What are the main observations regarding line intersections w.r.t to a sweep line?

- Two line segments can only intersect, after they are direct neighbors on the sweep line
- The state of the sweep line changes only at segment endpoints
- A segment locally always only has two neighbors → we only have to test 2 pairs

▼ In the context of sweep line algorithms, what are events?

Points on the y-axis, where the state of the sweep line changes. In our case, those are segment endpoints and intersection points.

▼ How does the proposed algorithm work?

It uses an event queue $\mathcal{Q}$ and a sweep line status $\mathcal{T}$. Initially, all segment endpoints are added to $\mathcal{Q}$. Then, those points are handled consecutively with the following procedure:

handleEvent(p)
  $U(p) \leftarrow$ line segments with p as upper endpoint
  $L(p) \leftarrow$ line segments with p as lower endpoint
  $C(p) \leftarrow$ line segments with p as interior point
  if $|U(p) \cup L(p) \cup C(p)| \geq 2$ **then**
    return p and $U(p) \cup L(p) \cup C(p)$
  remove $L(p) \cup C(p)$ from $\mathcal{T}$
  insert $U(p) \cup C(p)$ to $\mathcal{T}$
  if $U(p) \cup C(p) = \emptyset$ **then** //s_l and s_r neighbors of p in $\mathcal{T}$
    $\mathcal{Q} \leftarrow$ check if s_l and s_r intersect below p
  else //s' and s'' leftmost and rightmost line segment in $U(p) \cup C(p)$
    $\mathcal{Q} \leftarrow$ check if s_l and s' intersect below p
    $\mathcal{Q} \leftarrow$ check if s_r and s'' intersect below p

▼ Which data structures are used?

$\mathcal{Q}$ is a priority queue sorted lexicographically from top to bottom, stored in a balanced binary search tree. Operations `insert`, `delete` and `nextEvent` take $O(\log |\mathcal{Q}|)$ time.

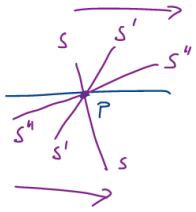
$\mathcal{T}$ is the status of the sweep line, but can also be stored in a balanced binary search tree containing the line segments, that intersect with the sweep line, from left to right.

▼ How can we get $U(p), L(p), C(p)$?

$U(p)$ is stored with the point p in the event queue. $L(p)$ and $C(p)$ are the neighbors in the sweep line status.

▼ Why is $C(p)$ removed and then added?

To reverse its order:



▼ How can you prove that the algorithm finds all intersection points?

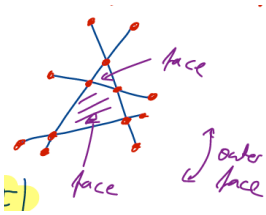
By induction on the number of events processed. Assume, all events up to the current one were correctly computed. If the current event is a segment endpoint, it was inserted into $\mathcal{Q}$ in the beginning. So it suffices to show, that if the current event p is an intersection point, it was added to $\mathcal{Q}$ at some point before. Assume two segments s_i and s_j, that are direct neighbors in p. There has to exist an event q where those segments became neighbors. By induction hypothesis, all previous events were handled correctly, thus p was inserted into $\mathcal{Q}$

▼ What is the running time of the algorithm?

Initially, all $2n$ endpoints have to be added to $\mathcal{Q}$, which takes $O(n \log n)$ time. Then, depending on the number I of intersections, $O(n + I)$ events have to be handled. Each event gets added to the queue once and deleted at most once, therefore the while-loop is iterated $O(n + I)$ times. It remains to determine the runtime of handling a single event. Finding, deleting and adding ≤2 new events to $\mathcal{Q}$ can be done in $O(\log n)$, thus increasing the runtime to $O((n + I) \log n)$. Changes to $\mathcal{T}$ also take $O(\log n)$ time, however it is not clear, how many such changes happen at each event.

Viewing the segments and their intersections as a planar graph, one can show that the number of changes at each event are in $O(n + I)$. Let $m(p)$ be the number of changes at event p. We show that

$$m := \sum_{p \in \mathcal{Q}} m(p) \leq \sum_{v \in V} d(v) = 2 \cdot |E|$$



and from Eulers formula $|V| - |E| + |F| = 2$, $F \leq \frac{2|E|}{3}$ and $|V| \leq 2n + I$ we follow

$$|E| \leq 3|V| - 6 \quad \Longrightarrow \quad m \leq 2|E| \leq 6n + 3I - 6 = O(n + I)$$

Therefore, in the whole course of the algorithm, the number of changes to $\mathcal{T}$ are in $O(n + I)$which gives us again the runtime of $O((n + I) \log n)$.

▼ What edge cases have to be considered?

- Line segments that share the same endpoint
- Segments with endpoints on other segments
- Completely horizontal segments
- Two segments, s.t. two of their endpoints have the same y-coordinate.

▼ Is the algorithm from the lecture always better than the naive one?

No, if $I \in \Omega(n^2)$, then the runtime of the sweep-line algorithm is $O(n^2 \log n)$.

▼ Are there algorithms that have a better runtime?

Yes, in $\Theta(n \log n + I)$ time and $\Theta(n)$ space [Balaban, 1995].

Polygon Triangulation

▼ What is a simple polygon?

No self-intersections, no holes

▼ Is it possible to triangulate every simple polygon with n corners and if yes, how many triangles does a triangulation contain?

Yes, any such triangulation contains exactly $n-2$ triangles.

▼ How can you prove this?

We show this for a polygon P with n corners by induction over n:

$n=3$ — clearly, a simple polygon with 3 corners has $3-2=1$ triangle.

$n>3$ — we show, that there exists a diagonal in the polygon along which we can split the polygon into two simple polygons P_1, P_2. From the induction hypothesis, we know that those smaller polygons have triangulations with n_1-2 and n_2-2 triangles. Also, we know that $n=n_1+n_2-2$. So in total, we have $n_1-2+n_2-2=n-2$ triangles. □

▼ How can you naively find such a triangulation?

The previous proof implies a recursive $O(n^2)$-algorithm.

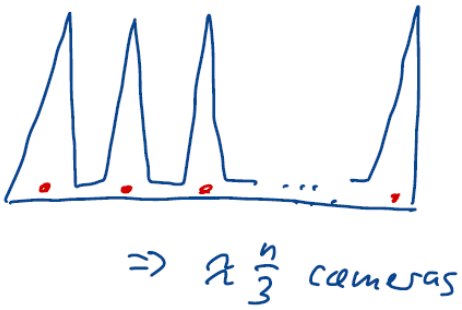
▼ What is the Art-Gallery-Problem?

Finding a minimal amount of points to place cameras to guard an art gallery. Generally, the problem is NP-hard.

▼ How many cameras are sufficient to guard any simple polygon?

$\lfloor n/3 \rfloor$ cameras are sometimes necessary and always sufficient to guard it.

▼ Can you give an example, where this number is necessary?



▼ How can you prove sufficiency?

Triangulate the polygon and find set V', s.t. each triangle is incident to at least one $v \in V$' and $|V'| \leq \lfloor n/3 \rfloor$. This can be proven by 3-coloring the triangulation and choosing the least-selected color as V'. Such a 3-coloring always exists, which can be shown by considering the dual graph of the triangulation, which is a tree.

▼ How does a faster triangulation algorithm work?

Decompose P into y-monotone polygons, then triangulate the resulting polygons

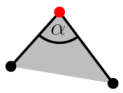
▼ What are y-monotone polygons?

For any horizontal line, the intersection with the polygon has to be connected.

▼ How can you categorize vertices of polygons?

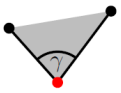
- Turn vertices

  ▼ Start vertices

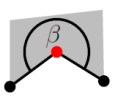
  if $\alpha < 180°$

  ▼ End vertices



if $\gamma < 180°$

▼ Split vertices



if $\beta > 180°$

▼ Merge vertices

if $\delta > 180°$

▼ Regular vertices

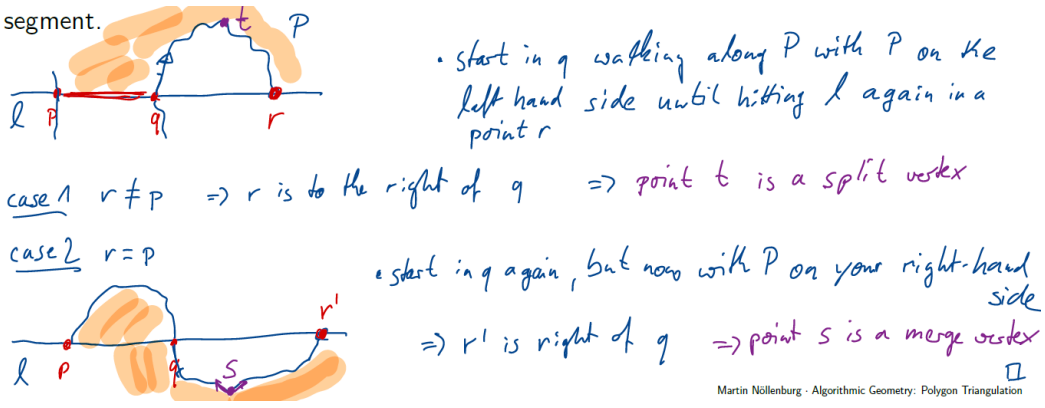
▼ Which types must not y-monotone polygons contain?

Split vertices and merge vertices

▼ How could you prove this?

Assume that P is not y-monotone and show that it must contain a split or a merge vertex. Let ℓ be a horizontal line intersecting P in at least 2 segments and be $\overline{pq}$ the leftmost such segment. When going along the edge of the polygon, there is no way to reach the second segment without encountering a split or a merge vertex.

▼ Proof from the slides



▼ How can you decompose P into y-monotone polygons?

Eliminate all split and merge vertices by adding diagonals

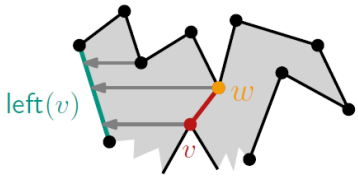
▼ What diagonal do you add for a split vertex v?

Connect v to the lowest vertex w sharing $\text{left}(w) = \text{left}(v)$.

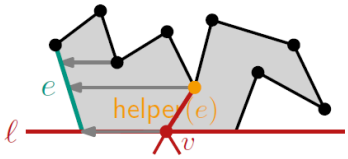
▼ Detailed slide

1) Diagonals for the split vertices

- find for each vertex v its left adjacent edge left(v) wrt. horizontal sweep line ℓ



- connect split vertex v to lowest vertex w above v with left(w) = left(v)
- for each edge e store the lowest vertex w such that left$(w) = e$; notation helper$(e) := w$
- when ℓ reaches a split vertex v, we connect v with helper(left(v))



▼ What diagonal do you add for a merge vertex v?

Connect it to the vertex that replaces it as $\text{helper}(e)$, where $e = \text{left}(v)$ or, if the end of the edge is reached, to the end vertex of this edge.

▼ How does the algorithm work?

A sweep-line algorithm going through the vertices from top to bottom. It uses a special data structure $\mathcal{D}$ (doubly connected edge list), a priority queue $\mathcal{Q}$ and a sweep-line-status $\mathcal{T}$ as known from the line segment intersection algorithm. For each edge $e$intersecting with the sweep line, it stores the lowest vertex over the sweep line directly to the right of the edge as $\text{helper}(e)$.

Handling of a vertex v for different vertex types:

▼ Start vertex

Set $\text{helper}(e) := v$ for its left incident edge e.

Add e to $\mathcal{T}$.

▼ End Vertex

If $u := \text{helper}(e)$ of the left edge e of v is a merge vertex, then add the diagonal uv.

Remove e from $\mathcal{T}$.

▼ Split Vertex

For the edge of the left of v in $\mathcal{T}$, e and $u := \text{helper}(e)$, add the edge uv.

Then set $\text{helper}(e) = v$.

Add left incident edge e' to $\mathcal{T}$.

Set $\text{helper}(e') = v$.

▼ Merge Vertex

Let e be the right edge of v. If $u := \text{helper}(e)$ is a merge vertex, then add the diagonal uv.

Let e' be the edge to the left of v in $\mathcal{T}$. If $w := \text{helper}(e')$ is a merge vertex, then add the diagonal vw.

Set $\text{helper}(e') = v$.

▼ Regular Vertex

If P lies locally to the right of v: Let e be the upper edge of v and e' the lower.

If $u := \text{helper}(e)$ is a merge vertex, then add the diagonal uv.

Remove e from $\mathcal{T}$. Add e' to $\mathcal{T}$. Set $\text{helper}(e') = v$.

Else: Let e be the edge of the left of v in $\mathcal{T}$ If $u := \text{helper}(e)$.

If $u := \text{helper}(e)$ is a merge vertex, then add the diagonal uv.

Set $\text{helper}(e) = v$.

▼ What is a doubly connected edge list?

A data structure that stores faces, (half-)edges and vertices with multiple types of pointers:

- For each face a representative half-edge.
- For each half-edge its origin vertex, its successor edge, its predecessor edge, and its twin edge.
- For each vertex its coordinates and an arbitrary outgoing half-edge.
- If faces have holes, store a list of pointers to a representative boundary edge of each hole.

▼ How can you traverse all edges incident to a vertex quickly?

Start at the half-edge e associated to the vertex, and go to the next edge via $\text{next}(\text{twin}(e))$.

▼ How much space does it need?

As we are storing a planar graph, we have a linear number of edges (and faces). Thus, space is in $O(n)$.

▼ How can you prove correctness of the algorithm?

By construction, we remove all split and merge vertices. It remains to show, that none of the inserted diagonals cross. This can be done by contradiction.

▼ What is the runtime of the algorithm?

$O(n \log n)$ time, $O(n)$ space

- Construct priority queue $\mathcal{Q}$: $O(n)$
- Initialize sweep-line status $\mathcal{T}$: $O(1)$
- Handle a single event: $O(\log n)$
  - $\mathcal{Q}$.deleteMax: $O(\log n)$
  - Find, remove, add element in $\mathcal{T}$: $O(\log n)$
  - Add ≤ 2 diagonals to $\mathcal{D}$: $O(1)$

▼ How can you triangulate y-monotone polygons?

Greedy top-down traversal of both sides.

We get the polygon as a double connected edge list. We maintain a stack containing the not-yet-triangulated vertices. We iterate through the vertices from top to bottom: If we switch sides, we connect the current vertex to all elements of the stack. If we stay on one side, we connect the current vertex to all elements on the stack, which are “visible” from the current vertex.

▼ What is the runtime of this algorithm?

$O(n)$

▼ What is the runtime of this faster algorithm?

$O(n \log n)$, coming from the runtime of splitting in y-monotone polygons.

▼ Can this algorithm be extended to work with polygons with holes?

Yes, but we need $\lfloor (n+h)/3 \rfloor$ cameras, where h is the number of holes.

▼ Can we solve the problem faster (for simple polygons)?

Yes, in linear time, but complicated.

Block 2

Range Queries

▼ What is computed in a range query?

The set of points that lie in a specific range

▼ Which data structure is useful to perform range queries in $\mathbb{R}$?

Balanced binary search tree

▼ What do the internal nodes of the tree store?

The maximum value of the left subtree

▼ How can you perform a range query for the interval $[x, y]$?

Find the split node, where the search paths diverge. For the left path, check if x is smaller than the value of the node. If yes, report the whole right subtree and proceed with the left child. If not, proceed with the right child. The same happens for the right search path.

▼ What is the runtime of a one-dimensional range query?

We have two search paths that go from the root to a leave, which takes $O(\log n)$ in a balanced binary search tree. Additionally, we need to report full subtrees. Reporting a subtree is in $O(k_v)$, where k_v is the number of nodes in the subtree. So the total runtime is $O(\log n + k)$, where $k := \sum_{v \in V} k_v$.

▼ How do you call such a runtime?

Output-sensitive, as it depends on the size of the output set

▼ How can you generalize range queries to $\mathbb{R}^2$?

- kd-trees
- Range Trees

▼ What are kd-trees?

Binary search tree, where even levels subdivide points along the x-axis, and odd levels split them along the y-axis.

▼ How do you query in a kd-tree?

Exactly like in a regular binary search tree, but you compare x and y coordinates alternatingly. Similar to 1d-range trees, we can report full subtrees if the region of a node is contained in the query rectangle. Otherwise, we have to search further.

▼ What is the construction time of kd-trees?

$O(n \log n)$,

▼ How can you show this?

Via the following recurrence relation:

$$Q(x) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2Q(\lceil n/2 \rceil) & \text{otherwise} \end{cases}$$

At each step, we determine the median and divide the list, which takes us $O(n)$. Then we repeat it for the halved sets. The recurrence relation is the same as the one of MergeSort, which is $O(n \log n)$.

▼ What is the query time?

Reporting of the subtree is still linear in $O(k)$, however we don't know how how many other internal nodes are visited. One can show the number is in $O(\sqrt{k})$, which gives us the result $O(\sqrt{n} + k)$.

▼ How can you show this?

Recurrence relation:

$$Q(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2 + 2Q(n/4) & \text{otherwise} \end{cases}$$

This resolves to $O(\sqrt{n})$.

- What are Range Trees?

  Binary search tree for the x-coordinates, where each internal node contains another binary search tree for the canonical subset for its y-coordinates.

  - What is the construction time and space usage of a range tree?

    $O(n \log n)$

    - How can you show this?

      Even though each node stores another binary search tree, we use the property that per level, each point is stored once in some of the secondary trees. As we have $O(\log n)$ levels in the primary tree, this gives us $O(n \log n)$ space for the whole range tree.

      The construction time for the primary tree is $O(n \log n)$. For the secondary trees, we can pre-sort P by y-coordinates in $O(n \log n)$ and build the secondary tree for the root in $O(n)$ time. Then we filter the sorted list into two sorted lists for the subsets of each children and again create their trees in $O(n)$. We can do this recursively until we reach the leaves. For each level, we need $O(n)$ time for the secondary trees, thus we get $O(n \log n)$ for construction.

  - How do you query in a range tree?

    Exactly like in a 1d-range tree, but instead of reporting a subtree, you need to perform a 1-dimensional range query for y.

  - What is the runtime of such a query?

    $O(log^2 n + k)$

    - How can you show this?

      We have two search paths of length $O(\log n)$ which might all cause range queries on the secondary trees taking $O(\log n + k_v)$. In total, this gives us $O(\log^2 n)$ for following a path on the primary tree and each of the secondary trees on the path and additionally $O(k)$ for the ReportSubtree routines.

    - Can this runtime be improved?

      Yes, by fractional cascading we can get a runtime of $O(\log n + k)$.

      - What is fractional cascading?

        The query interval in the y-axis always stays the same. We can use this property to accelerate the 1d queries to $O(1 + k_v)$ time. Instead of secondary trees we store arrays which contain the points sorted by their y-coordinate in an inner node. Each array element a gets a pointer to the array of the left and right child, pointing to the smallest $b \geq a$. This pointer yields the starting point for the secondary search in constant time.

- How do you deal with points that are not in general position? [TODO]

Windowing Queries

- What is an interval tree?

  Each internal node represents the median m of interval endpoints. At an internal node, the set of intervals is stored, which contain this median. This set is stored twice, once as $\mathcal{L}$ sorted from left to right and once as $\mathcal{R}$ from right to left.

  The left child is responsible for all intervals which lie completely to the left of m, the right child for all intervals which lie completely to the right.

  - What is the construction time and space bound for an interval tree?

    Sorting the interval endpoints from left to right to determine their median in $O(n \log n)$. Then we just split the intervals in the three sets in $O(n)$ time per level.

    The tree stores each interval exactly once, so the space is in $O(n)$.

  - How can you query an interval tree?

The query finds all intervals a point lies in. We check, whether the point lies to the left or to the right of the root node. If it lies on the left, we check $\mathcal{L}$ for intervals containing p, and continue in the left child. We do this until we reach a leaf.

- ▼ What is the runtime of such a query?

  The search path has a length of $O(\log n)$, and in each node we need to check the intervals stored there, which takes $O(1 + k_v)$. In sum, we have $O(\log n + k)$.

▼ How can adapt an interval tree to work in two dimensions, outputting the intervals for query segments instead of query lines?

In the nodes, we store a range tree instead of simple lists.

- ▼ How does that affect space and time complexity?

  Construction time remains the same, but querying time increases to $O(\log^2 n + k)$. Also the space consumption increases, as a range tree needs $O(n \log n)$ space. However, all range trees store $O(n)$ endpoints in total, so the space consumption is exactly $O(n \log n)$

▼ What is a segment tree?

Splits intervals up into elementary intervals, s.t. we know for every point, to which intervals this point belongs.

- ▼ What is the space complexity of a segment tree?

  Naively, if you store the elementary intervals in the leaves, you might get quadratic space consumption. We can avoid this by storing an interval as high up in the tree as possible. This way, each interval gets stored at most twice per level, thus resulting in $O(n \log n)$ space.

- ▼ What is the construction time?

  $O(n \log n)$ — first we need to sort the endpoints of the intervals and build our binary search tree structure. Then we need to compute the interval set for each node. As each interval is only stored twice per level, we need $O(\log n)$ for computing this set. So in total, we have $O(n \log n)$ for all intervals.

- ▼ How does a query work?

  We report all intervals in the current node, then depending on whether the query point lies in the left elementary interval or in the right one, we go to the left or right child and repeat.

- ▼ What is the runtime of a query?

  We have one search path in $O(\log n)$, but we need to report all intervals which takes $O(k)$, so we have $O(\log n + k)$ in total.

▼ What is the difference between interval trees and segment trees?

In a segment tree, all intervals stored in a visited node v contain the query point. In an interval tree, you have to continue searching. You pay for this by a higher space requirement of segment trees.

▼ How can you adapt segment trees to work in two dimensions with arbitrarily oriented line segments?

Create segment trees for projected x-intervals. We find segments crossing the query segment by an auxiliary binary search tree, containing the segments of an elementary interval from top to bottom.

- ▼ How does that affect space and time complexity?

  Space requirement stays the same, as we store the same amount of intervals, just in another arrangement. Building auxiliary binary trees takes $O(n \log n)$ per level, which results in $O(n \log^2 n)$ construction time. In a query, we also need to query the auxiliary binary tree, increasing query time to $O(\log^2 n + k)$.

▼ Why can't you use interval trees for arbitrarily oriented line segments? [TODO]

Point Location Queries

▼ What is the goal of a point location query?

To find the face a query point lies in.

▼ How can we efficiently answer point location queries?

Using a trapezoidal map data structure. We shoot a ray to the top and bottom of each endpoint, and therefore yield a partition into trapezoids.

▼ What is the complexity of such a trapezoidal map?

It has at most $6n + 4$ vertices and $3n + 1$ trapezoids.

▼ How could you show this?

$$\#\text{vertices} \leq \underbrace{2n}_{\text{endpoints of } S} + \underbrace{2 \cdot 2n}_{\text{from rays}} + \underbrace{4}_{\text{corners}} = 6n + 4$$

Each left point contributes to at most 2 trapezoid. The right points contribute to 1 trapezoid. The corners of the bounding box contribute to 1 trapezoid. Summing this up, we get $3n + 1$.

▼ How can we compute a trapezoidal map?

We store the trapezoidal map itself as a DCEL. Additionally, we store a DAG for answering point location queries. We construct both iteratively by adding one segment after the other.

▼ What does the DAG consist of?

- x-nodes for testing whether a point is left/right of a segment endpoint
- y-nodes for testing whether a point is above/below a segment
- Leaf nodes indicating the trapezoid which lies at this location

▼ What problem arises with this iterative approach?

Size of the DAG and thus query time depend on the insertion order of the segments.

▼ How can you fix this?

Randomizing the insertion order

▼ How does the algorithm work in detail?

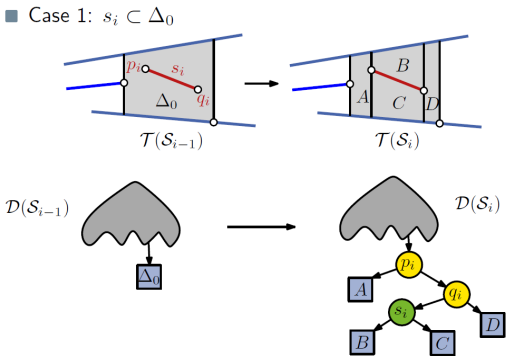
Find the set of trapezoids affected by a new segment. Remove the leaf nodes of those trapezoids and replace them with a new structure including the new segment.

▼ How can we find the set of trapezoids affected?

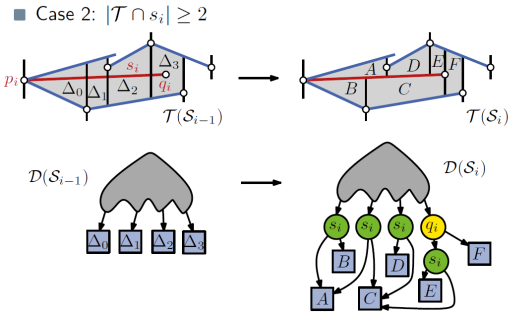
We find the trapezoid of the first endpoint of the segment. depending on whether the rightp() of a trapezoid is above or below the line, we go to the upper or lower right neighbor of the trapezoid. We do this, until we reach the trapezoid, whose rightp() is right of the right endpoint of the segment.

▼ Case 1: segment is fully contained within one trapezoid

We only need to replace this one leaf by a structure consisting of 4 leafs and 3 internal nodes.



▼ Case 2: segment intersects more than one trapezoid



▼ What is the construction time of a trapezoidal map?

Expected construction time is $O(n \log n)$, **expected** space in $O(n)$, **expected** query time in $O(\log n)$. Can be shown by backward analysis.

▼ What is the worst case for the query time?

We can bound the probability of the max search path being longer than some multiple of $O(\log n)$.

Block 3

Voronoi Diagrams

▼ What is a Voronoi diagram?

A set of cells, edges and vertices subdividing the plane. For a Voronoi diagram of a point set P, each point gets its own cell. It holds, that for each point p in the plane, the closest point of P is the one of the cell that p lies in. If p lies on an edge between two cells, both points on the adjacent cells are equally close. If p lies on a vertex, it is equally close to all adjacent cells.

▼ How can you define Voronoi cells, edges and vertices?

$\mathcal{V}(p) = \{q \mid q \in \mathbb{R}^2, |pq| < |qr|, r \in P \setminus \{p\}\}$

$\mathcal{V}(\{p,q\}) = \text{rel-int}(\partial\mathcal{V}(p) \cap \partial\mathcal{V}(q))$, where rel-int is the interior of the edge without its endpoints and δ is the boundary of the Voronoi cell.

$\mathcal{V}(\{p,q,r\}) = \partial\mathcal{V}(p) \cap \partial\mathcal{V}(q) \cap \partial\mathcal{V}(r)$

▼ Are all Voronoi cells convex? [TODO]

▼ How many nodes and edges does a Voronoi diagram at most have?

$|V| \leq 2n - 5$

$|E| \leq 3n - 6$

▼ How can you prove this?

Connect unbounded Voronoi edges to a dummy helper vertex. Use Eulers formula and the fact that each Voronoi vertex has at least degree 3.

▼ What is an alternative characterization of Voronoi vertices/edges?

We can also characterize them w.r.t. empty circles. A point is a Voronoi vertex of points $p_1, ..., p_i$, iff the circle through those points is empty. A point is a voronoi edge of p_1, p_2, iff there exists an empty circle through those two points.

▼ How can you compute a Voronoi diagram?

Sweep-line-ish approach, where we consider point and circle events and everything above a so-called beach line is already fixed.

▼ Why can't we use a basic sweep-line approach to compute a Voronoi diagram?

Because Voronoi cells above the sweep line can be affected by events below the sweep line. Therefore, the classical approach where everything above the sweep line is already fixed, does not work.

▼ Which area above the sweep line is already fixed, then?

All points that are closer to some point above the sweep line than to the sweep line itself. This equation results in a parabola.

- ▼ What is the beach line?

  The lower envelope of all parabolas of the points already visited.

- ▼ What is a point event?

  A point of the point set that results in a new parabola to be added.

- ▼ What is a circle event?

  The point, where the sweep line lies on the circle going through ≥ 3 points. Here, at least one parabola disappears, which results in a Voronoi vertex.

▼ How does the algorithm work in detail?

- ▼ How is the beach line/are the parabolas stored?

  It implicitly stores the beach line as the corresponding points of parabolas in a binary tree $\mathcal{T}$. Each internal node represents the intersection of two parabolas.

- ▼ How are events stored?

  The events are stored in an event queue $\mathcal{Q}$, which is a priority queue, e.g. implemented via a balanced binary search tree.

- ▼ How is the Voronoi diagram stored?

  The resulting Voronoi diagram is stored in a doubly-connected edge list.

- ▼ How are point events handled?

  At a point event p, a new parabola has to be added to $\mathcal{T}$. We locate the parabola lying directly above the the point (binary search), remove the point q representing the parabola from the tree and re-add q, p, q with intersection points $\langle p, q \rangle$ and $\langle q, p \rangle$ to the tree. Also remove circle events of q from $\mathcal{Q}$.

  We also have to take care about circle events introduced by this new point. Potential events are $\langle p_l, q, p \rangle$ and $\langle p, q, p_r \rangle$.

- ▼ How are circle events handled?

  Remove the arc affected by the circle event from $\mathcal{T}$ and any circle points of this arc. Add a Voronoi vertex at the center of the circle and Voronoi edges between the point and its neighbors.

▼ What is the runtime of computing a Voronoi diagram?

Handling an event takes $O(\log n)$ time for performing operations on $\mathcal{T}$ and $\mathcal{Q}$. There are n point events and as a Voronoi diagram has a linear number of Voronoi vertices, also $O(n)$ circle events. False alarm circle events might be added to the queue at some point, but they get deleted before they get processed.

Delaunay Triangulations

▼ How many triangles and edges does a triangulation of a set of n points have?

$t = 2n - h - 2$

$e = 3n - h - 3$

- ▼ How could you prove this?

Proof: Recall Euler: $|V| - |E| + |F| = 2$ $(\star)$

let $t = t(n, h)$ be # triangles

- $|F| = t + 1$ ← outer face
- each triangle has 3 edges, outer face has h edges $\Rightarrow |E| = \frac{3t + h}{2}$
- insert into $(\star)$: $n - \left(\frac{3t+h}{2}\right) + t + 1 = 2 \Leftrightarrow -\frac{t}{2} - \frac{h}{2} + n = 1$

$\Leftrightarrow t = 2n - h - 2$

let $e = e(n, h)$ be # edges

- $e(n, h) = |E| = \frac{3t + h}{2} = \frac{3 \cdot (2n - h - 2) + h}{2} = 3n - 3 - h$

▼ What kind of triangles should be avoided for interpolation purposes?

Skinny Triangles → maximize the smallest angle

▼ When is a triangulation $\mathcal{T}$ called angle-optimal?

If its angle vector is lexicographically larger or equal to the angle vector of all other triangulations.

▼ What is an angle-vector?

Vector containing the angles of the triangles sorted from smallest to biggest

▼ When is an edge of a triangulation called illegal?

If the smallest angle increases after flipping the edge.

▼ What is a legal triangulation?

A triangulation without illegal edges.

▼ Is there always a legal triangulation?

Yes. Consecutively flip every illegal edge until the triangulation has no illegal edges anymore.

while $\mathcal{T}$ has an illegal edge e **do**
  flip$(\mathcal{T}, e)$
return $\mathcal{T}$

terminates, since $A(\mathcal{T})$ increases and #Triangulations is finite ($< 30^n$, [Sharir, Sheffer 2011]), see also exercise sheet 3

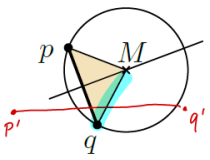
▼ Is a angle-optimal triangulation legal?

Yes, if there was an illegal edge, it would not be angle-optimal.

▼ What is the Delaunay graph?

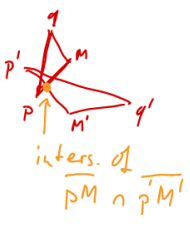
The dual graph of a Voronoi diagram, where neighboring cells are connected via straight lines.

▼ How could you prove that $\mathcal{DG}(P)$ is crossing-free?



- let $\overline{pq}$ be edge in $\mathcal{DG}(P)$
- let $t_{p,q}$ be triangle with midpoint of $C_{p,q}$

$\Rightarrow$ $\overline{pM} \subset \mathcal{V}(p)$ and $\overline{qM} \subset \mathcal{V}(q)$



- assume two edges $\overline{pq}$ and $\overline{p'q'}$ intersect in $\mathcal{DG}(P)$
- p', q' are not inside $C_{p,q}$ and hence not in $t_{p,q}$

$\Rightarrow$ $\overline{p'q'}$ must intersect $\overline{pM}$ or $\overline{qM}$

- for the same reason $\overline{pq}$ must intersect $\overline{p'M'}$ or $\overline{q'M'}$

$\Rightarrow$ w.l.o.g. $\overline{pM}$ intersects $\overline{p'M'}$, but this contradicts $\overline{pM} \subset \mathcal{V}(p)$ and $\overline{p'M'} \subset \mathcal{V}(p')$

▼ What is the Delaunay triangulation?

If P is in general position (meaning no 4 points lie on a circle), then all faces of the Delaunay graph are already triangles. Otherwise, we add diagonals to split all faces into triangles. The result is called the Delaunay triangulation.

▼ How many legal triangulations are there?

If P is in general position, it has only one legal triangulation, which is the Delaunay triangulation.

▼ How can you compute the Delaunay triangulation?

Compute Voronoi diagram in $O(n \log n)$, construct dual graph in linear time.

Point-Line Duality

▼ How is the duality transform defined?

$p = (a, b) \Longrightarrow p^* : y = ax - b$

$l : y = cx + d \Longrightarrow l^* = (c, -d)$

▼ Which properties in the primal plane correspond to which properties in the dual plane?

- $(p^*)^* = p$ and $(l^*)^* = l$

▼ If p lies above/below/on l in the primal plane,

l^* lies below/above/on p^* in the dual plane.

▼ If two lines ℓ_1 and ℓ_2 intersect in a point r,

the line r^* passes through ℓ_1^* and ℓ_2^*.

▼ If points $p_1, ..., p_i$ are collinear in the primal plane,

their dual lines $p_1^*, ..., p_i^*$ intersect in the dual plane, and their intersection point corresponds to the line in the primal going through all the points.

▼ How do line segments look like in the dual space?

Double wedges

▼ How can you compute the lower envelope of a set of lines?

Compute the upper convex hull of the dual points. The upper convex hull in reverse order is the lower envelope.

▼ What is a line arrangement $\mathcal{A}(L)$?

A set of lines, that divides the plane into edges, vertices and (possibly unbounded) cells.

▼ When is a line arrangement called simple?

- Only 2 lines intersect in any point
- No lines are parallel

▼ How many edges, vertices and cells are there in $\mathcal{A}(L)$ for n lines?

n lines have $\binom{n}{2}$ intersections. Each line gets split $n - 1$ times, resulting in n pieces, therefore we get n^2 edges. Adding a dummy vertex and using Eulers formula, we get $|F| = \binom{n}{2} + n + 1$.

▼ What is the Zone Theorem for line arrangements?

A zone of a line arrangement $\mathcal{A}(L)$ and a line ℓ is defined as the set of all cells which ℓ intersects.

The Zone Theorem states, that for any arrangement and any line, the zone consists of at most $6n$ edges.

▼ How could you prove this?

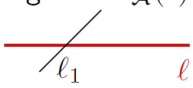
We can show by induction, that there are at most $3n$ left-bounding edges for n lines. The induction step works by showing, that inserting a line splits at most two existing edges and adds one new left bounding edge. Therefore, we add at most 3 new left bounding edges. The same argument can be repeated for right bounding edges.

▼ Full proof

Proof:

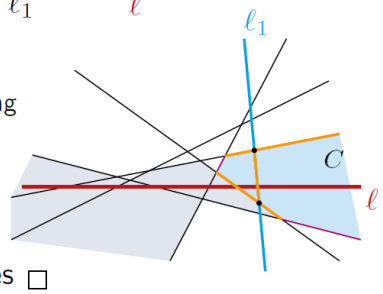
- induction on $i = \#$ lines in L
- rotate $\mathcal{A}(L)$ s.t. ℓ is horizontal and define left/right
- **claim:** there are at most $3i$ left bounding edges in $Z_{\mathcal{A}}(\ell)$

$n = 1$: exactly one left bounding edge



$n - 1 \to n$: (induction step)

- let ℓ_1 be rightmost line of L intersecting ℓ
- for $\mathcal{A}(L \setminus \{\ell_1\})$ we have at most $3(n-1)$ left bounding edges by induction hypothesis
- ℓ_1 and ℓ intersect in rightmost cell C of $Z_{\mathcal{A}(L\setminus\{\ell_1\})}(\ell)$
- ℓ_1 splits ≤ 2 edges of C and contributes one left bounding edge $\Rightarrow \leq 3$ new left bounding edges added



- repeat same argument for right bounding edges $\square$

▼ How can we compute $\mathcal{A}(L)$?

We could use an algorithm we already know: The line segment intersection could calculate the intersections in $O((n+I)\log n) = O(n^2 \log n)$. Alternatively, we could use an iterative approach similar to the trapezoidal maps.

▼ How does the iterative approach work?

We add one line at a time. We determine the cells affected by the new line and update those cells accordingly. From the Zone Theorem we know, that the number of affected cells is linear.

▼ What is the runtime of our algorithm?

$O(n^2)$, dominated by the creation of the bounding box. Insertion of a line is linear, so inserting n points also takes $O(n^2)$.

▼ What other problems do you know that can be solved with duality?

- Minimum Area Triangle
- Line that cuts two sets of points in half simultaneously
- Stabbing Line

▼ How can you compute the minimum area triangle?

For each pair of points pq, the smallest triangle that you can form with them can be one of two candidate triangles — find the two points r_1, r_2, that are closest to the line pq above and below it. Either the triangle pqr_1 or pqr_2 is the smallest triangle containing p and q. This implies a $O(n^2)$ algorithm, if we can find the points r_1, r_2 in short time.

▼ How can you find the points r_1, r_2?

By considering $\mathcal{A}(P^*)$, the line arrangement of the dual lines of the points. The candidate points r_1, r_2 lie vertically above/below the point $(pq)^*$. This is linear in cell complexity.

▼ Can you use duality in higher dimensions?

Yes, you can define incidence- and order-preserving duality transforms between d-dimensional points and hyperplanes.

▼ What about the complexity of higher-dimensional arrangements?

The arrangement of n hyperplanes in $\mathbb{R}^d$ has complexity $\Theta(n^d)$. A generalization of the Zone Theorem yields an $O(n^d)$-time algorithm.

Block 4

Quadtrees & Meshing

▼ What is a quadtree?

A rooted tree, where each internal node has four children. Each node corresponds to a square, and the children form a partition of the root square. It is a data structure that allows fast access to points in the plane.

▼ What is the depth of a quadtree?

$\log(s/c) + 3/2$, where c is the smallest distance in P and s is the side length of the root square Q.

▼ How could you prove this?

The side length at depth i is $s/2^i$. The maximal distance of two points in a square with side length a is $a\sqrt{2}$. Therefore, the maximal distance of two points in a depth-i square is $\frac{s\sqrt{2}}{2^i}$.

$$\frac{s\sqrt{2}}{2^i} \geq c \iff 2^i \leq \frac{s\sqrt{2}}{c} \iff i \leq \log \frac{s}{c} + \frac{1}{2}$$

Lastly, we need to add one level for the leaves.

▼ What is the construction time and size of a quadtree?

$O((d+1)n)$, where d is the depth of the tree

▼ How could you prove this?

- Size: Each level partitions Q ⇒ at most n inner nodes per level and thus $O((d+1)n)$ nodes in total.
- Time: Distribute n points per level to all children takes $O(n)$ time per level, thus $O((d+1)n)$

▼ How can you access the north neighbor of a node in a quadtree?

If the node is a SW/SE-child, it is easy, as we can just return the NW/NE-child of the parent.

Otherwise, it involve stepping out of the squares into their parent squares until we either reach a SW/SE child or the root square. If we reach the root square, we do not have a north neighbor. If we reach a SW//SE child, we step into the NW/NE child as often as we stepped out.

▼ In what runtime can you access any neighbor of a node?

$O(d+1)$

▼ What is a balanced quadtree?

No leaf node can have a neighbor which has a non-leaf child / No two neighboring squares can differ by at most a factor two in their side length.

▼ How can you balance a quadtree?

Further split leaf nodes, that are too large, until the tree is balanced.

▼ How do you know, which leaf nodes are too large?

Check its four neighbors, if one of them has a non-leaf child then the node is too large.

▼ What is the space requirement of a balanced quadtree?

$O(m)$, where m is the number of nodes in the unbalanced tree.

▼ How can you show this?

We can show that we do $\leq 8m$ split operations, by showing that every square has a “old” neighbor that can be charged for splits. As every square has at most 8 neighbors, every existing node gets charged at most 8 splits, which results in at most $8m$ splits in total.

▼ What is the time complexity of balancing a quadtree?

$O((d+1)m)$ — we have $O(m)$ split operations, and each split operation takes $O(d+1)$.

▼ What is adaptive meshing?

Triangulate a square with octilinear, integer-coordinate polygons inside. The following properties have to hold:

- No triangle vertex in interior of triangle edges
- Input edges must be part of the mesh

- Triangle angles between 45° and 90°
- Adaptive (fine at polygon edges, otherwise coarser)

▼ How can we implement adaptive meshing?

Using a quadtree, subdivide until squares are no longer intersected by a polygon or have size 1.

To get a triangular mesh, balance the quadtree and add Steiner points, if necessary.

▼ How fast can you compute the adaptive mesh?

$O(p(S) \log^2 U)$, where $p(S)$ is the total perimeter of all polygons and U the side length of the base square.

▼ How many triangles does this produce?

$O(p(S) \log U)$, where $p(S)$ is the total perimeter of all polygons and U the side length of the base square.

▼ Are there quadtree variants with space linear in n?

Yes, compressed quadtrees, where internal nodes with only one non-empty child get contracted. The skip quadtree with $O(n)$ space can insert, remove and search in $O(\log n)$ time.

▼ How can you generalize a quadtree to higher dimensions?

Divide the base hypercube into 2^d equally sized hypercubes. For 3d, it is called octree.

WSPD

▼ What is a s-well separated pair?

A pair of two point sets A and B, which each can be covered with a disk of radius r, and both disks having at least a distance of sr.

▼ If a pair is s-well separated, what about $s' < s$?

It is also s'-well separated.

▼ What is a s-well separated pair decomposition (s-WSPD) of a point set P?

A set of pairs of sets, for which it holds that:

- Every point is contained in at least one set
- A pair of sets is disjoint
- Each pair is s-well separated

▼ What is the worst-case size of a WSPD?

$O(n^2)$ — just consider each pair of points at its own, singletons are trivially s-well separated for $s > 0$.

▼ How can you construct a linear WSPD?

Using compressed quadtrees. Each quadtree cell is compared with each other.

▼ What are compressed quadtrees?

Each path of a non-separating inner node is contracted into a single edge. Each edge is labeled to reconstruct the path structure.

▼ What does the Packing Lemma for quadtrees state?

Let K be a ball with radius r in $\mathbb{R}^d$. The number $|X|$ of a set X of pairwise disjoint quadtree cells with side length $\geq x$ intersecting K can be upper bound by

$$|X| \leq (1 + \lceil 2r/x \rceil)^d$$

▼ How many pairs does the algorithm create?

$O(s^d n)$

▼ What is the runtime of the algorithm?

$O(n \log n + s^d n)$ from the construction of the quadtree and for the s-WSPD

▼ What is a t-spanner?

A graph, for which it holds that $|xy| \leq \delta_G(x, y) \leq t \cdot |xy|$. In other words, the distance to get from x to y is at most t times the distance of xy.

▼ How can you compute a t-spanner?

Connect the representatives of all ws-pairs. For $s = s(t) \geq 4$, this yields a t-spanner with $O(s^d n)$ edges.

▼ Which s do we have to choose for a given t?

$4 \cdot \frac{t+1}{t-1}$

▼ What is the runtime for calculating a $(1+\varepsilon)$-spanner?

$O(n \log n + n/\varepsilon^d)$

▼ What are further applications of a WSPD?

- Euclidean MST (using a $(1+\varepsilon)$-spanner, we get a $(1+\varepsilon)$-approximation)
- Maximum distance in a point set (for $s = 4/\varepsilon$, we get a $(1+\varepsilon)$-approximation)
- Closest pair of points (for $s > 2$, we get the closest pair)

▼ Can we achieve the same bounds with exact computations?

In $\mathbb{R}^2$, this is often true, but for higher dimensions exact computations get more expensive.

Visibility Graph

▼ How does a shortest path from s to t in $\mathbb{R}^2$ with a set of disjoint open polygons as obstacles look?

It is a polyline whose interior nodes are corners of polygons in S.

▼ What is the visibility graph of a set of disjoint open polygons?

The graph with polygon corners as vertices and edges between vertices, which do not intersect with the polygon.

▼ How can you compute the shortest path using a visibility graph?

Compute visibility graph, perform Dijsktra.

▼ How can you compute a visibility graph?

Rotational sweep method for each vertex.

▼ What is the runtime of the algorithm computing a visibility graph?

The rotational sweep checks n points and for each point makes changes to the sweep line structure in $O(\log n)$, thus for each vertex we need $O(n \log n)$, however, we have to do this rotational sweep for each vertex, which results in $O(n^2 \log n)$.

Summary of complexity bounds