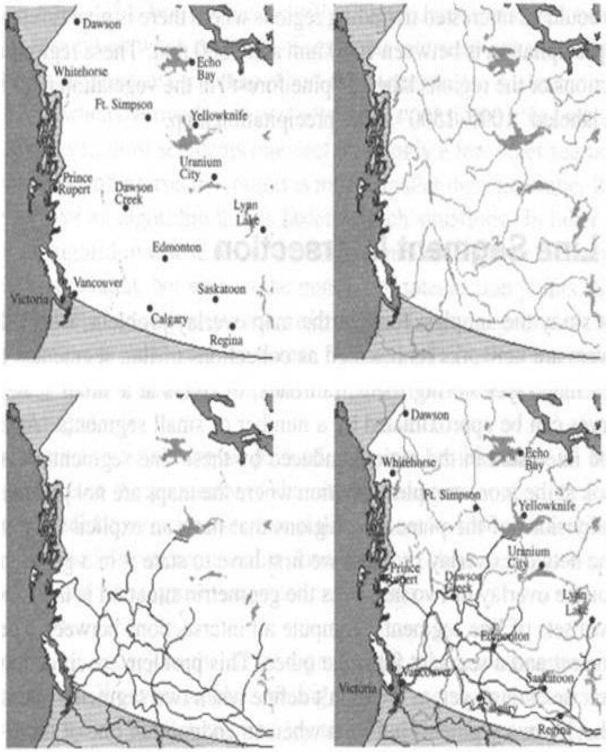
Computational Geometry

2. Line Segment Intersection

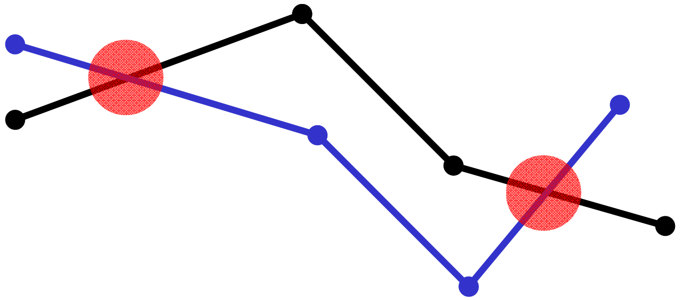
2.1 Motivation

- In a geographical information system (GIS) information is stored in layers of different maps (e.g. streets, rivers, villages,…).
- To mark bridges all intersections of streets with rivers must be computed.
- If streets and rivers are approximated by polygon chains, intersections of line segments must be computed.
- This may exceed the capabilities of a GIS, if the intersections of every street with every river is computed.



2.1 Motivation

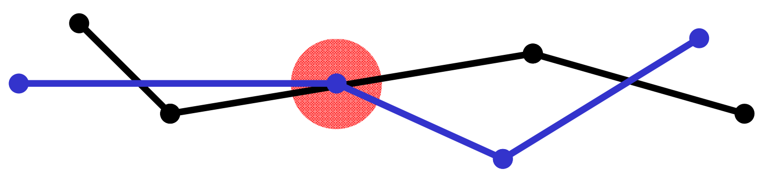
- Example:
  - A polygon chain of layer one,
  - A polygon chain of layer two, etc.



- For simplicity do not consider polygon chains but line segments only.

2.1 Motivation

- Principal approach
  - Add all line segments of all polygon chains to a single set and compute all intersections.
  - Then search for the intersections of individual polygon chains.
  - **Attention:** It might happen that a „real intersection“ coalesces with the start and end point of two adjacent line segments.

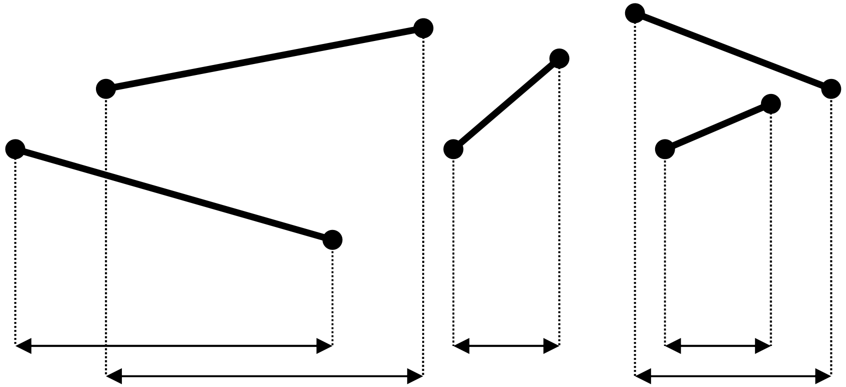


2.2 Naïve approach

- **Input:** Set $S = \{s_1, \dots, s_n\}$ of n line segments in the plane.
- **Goal:** All intersections of line segments from S.
- **Naïve Solution:**
  1. Test all pairs of segments.
  2. If they intersect, return the intersection point.
- **Run time:** $\Theta(n^2)$
- Optimal, if all pairs of line segments intersect, because then there are $\Theta(n^2)$ intersection points!
- Usually the number of intersections is smaller.

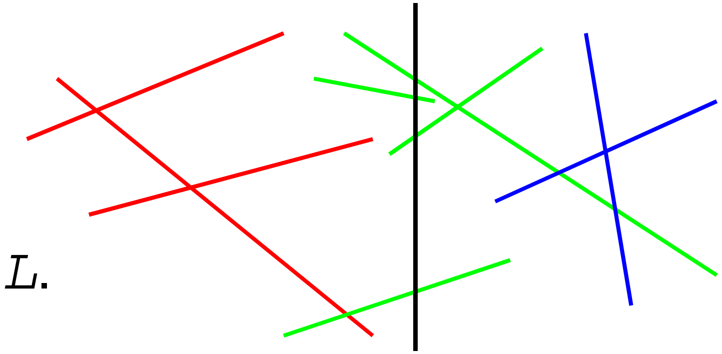
2.3 Scan-Line-Algorithm

- Scan-line principle
  - Scan-line algorithms are more efficient for a relatively small number of intersections (*output-size sensitive*).
- **Input:** Set $S = \{s_1, \dots, s_n\}$ of n line segments in the plane.
- **Idea:** Test only segments that are close to each other, because then it is likely that they intersect.



2.3 Scan-Line-Algorithm

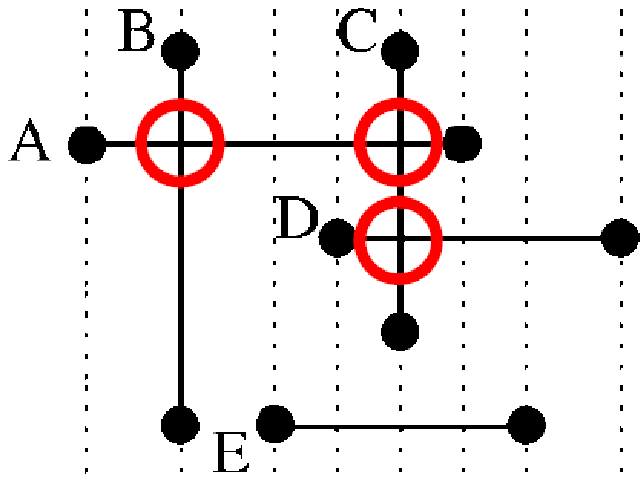
- Approach
  - Move a vertical line L (scan-line or sweep-line) from left to right over the set of line segments.
- The scan-line L clusters the segments in three classes:
  - ***dead objects***: lie completely left of L,
  - ***active objects***: intersect L,
  - ***inactive objects***: lie completely right of L.



- The state of L is the set of active objects.
- The state of L is only changed by end points of segments.
  - These points a called *event points*.

2.3.1 Iso-oriented segments

- For simplicity first the simpler case of ***iso-oriented line segments***.
- **Input:** Set $S = \{s_1, \dots, s_n\}$ of n vertical and horizontal line segments in the plane.
- **Goal:** All pairs of intersecting line segments in S.

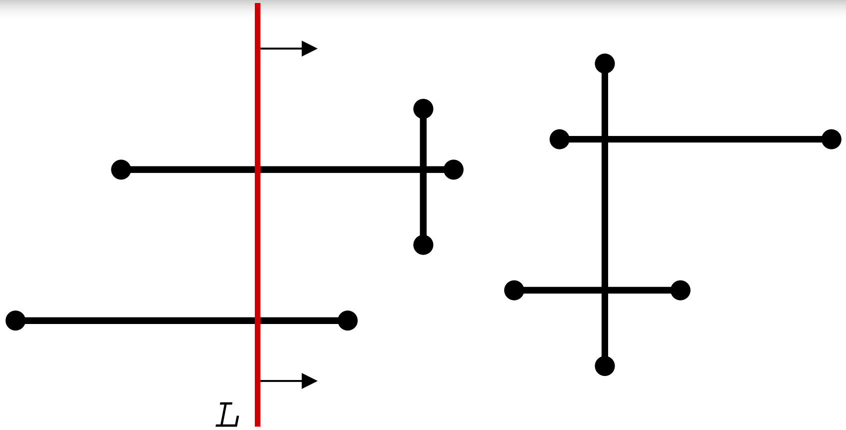


Intersecting pairs: 3
→ (B,A), (C,A), (C,D)

2.3.1 Iso-oriented segments

- What is the maximal number of intersections for n line segments in this case?
  - 50% of the segments are vertical and 50% are horizontal.
  - All possible intersections: $\left(\frac{n}{2}\right)^2$
- Simplifying assumption:
  - All start and end points of horizontal segments und all vertical segments have mutually distinct x-coordinates.
  - This eliminates some special cases in the algorithm.
  - For our example these assumptions are satisfied.

2.3.1 Iso-oriented segments



- Observations
  - It is not necessary to move the scan line L continuously, since only the event points are important.
  - Thus, there are three kinds of event points with respect to the x-coordinates:
    - start of a horizontal segment,
    - end of a horizontal segment,
    - vertical segment.
  - If L hits a vertical segment, this must be an active vertical segment.

2.3.1 Iso-oriented segments

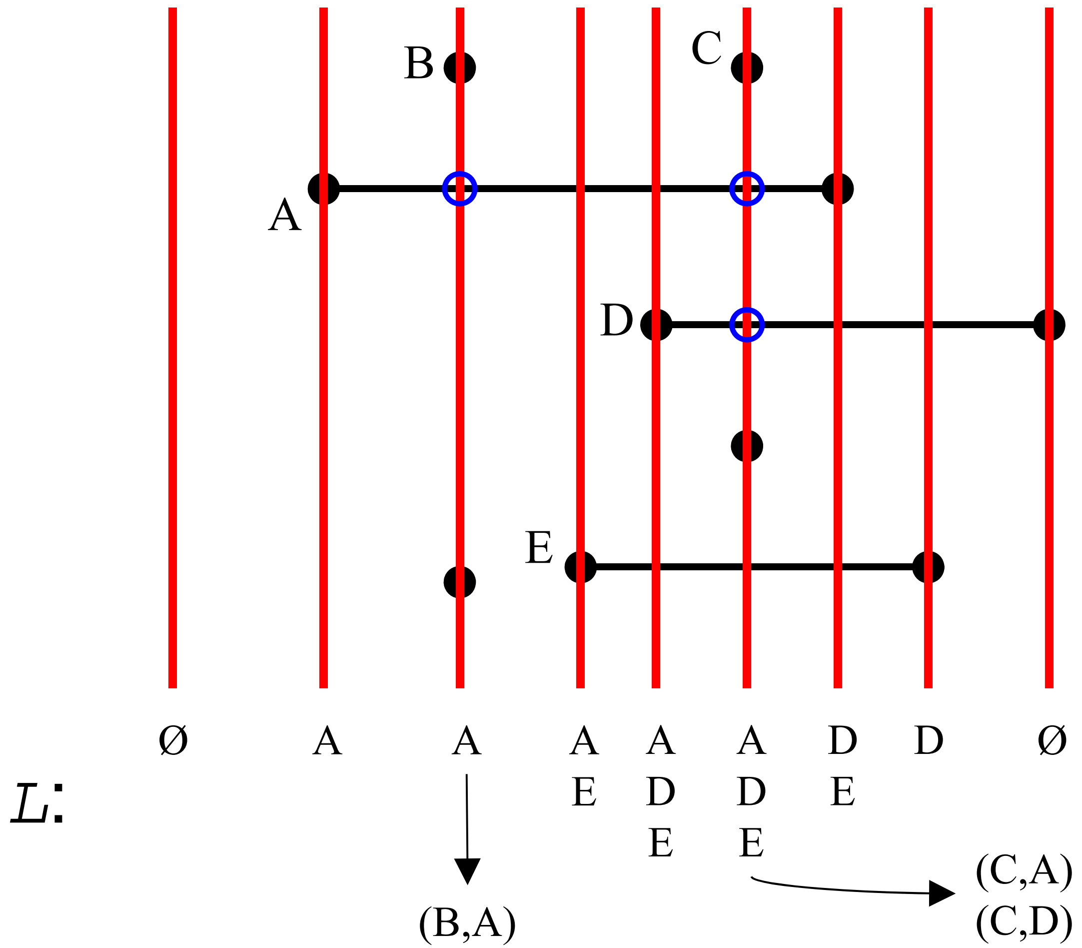
Algorithm 1: Intersect iso-oriented line segments

Input: n iso-oriented line segments.
Output: All intersections.

```
Initialize set Q with all events sorted by x-coordinate;
L = ∅; // List of active segments, sorted by y-coordinate
while (Q is not empty) {
  Take next event p from Q and remove it from Q;
  if (p is left end point of horizontal segment s) then {
    Add s to list L;
  } else if (p is right end point of segment s) then {
    Remove s from list L;
  } else { // p.key is the x-coordinate of a vertical
           // segment s with lower end point (p.key,y_l)
           // and  upper end point (p.key,y_u)
    Determine all horizontal segments t in L, whose
    y-coordinate t.y is in [y_l,y_u] and report the
    intersecting pair (t,s);
  }
}
```

2.3.1 Iso-oriented segments

- Example



2.3.1 Iso-oriented segments

- Requirements on the data structure for L (*event structure*)
  - Horizontal elements are stored in L with their y-coordinate as key.
  - Efficient operations for:
    - `insertion` of a new element,
    - `deletion` of an element,
    - output of all elements in L, whose key lies within a certain range (*range search*).
  - Possible solution: balanced binary search trees, e.g. AVL-trees, see [1] and [2].

2.3.1 Iso-oriented segments

- Operations on the data structure for L
  - `Insertion` and `deletion` have run time $O(\log |L|)$.
  - Line 11: `Range search` for horizontal line segments p with y-coordinate in $[y_l, y_u]$
    - Find element p with smallest key $\geq y_l$.
    - Return p, if key $\leq y_u$.
    - Traverse tree from p in in-order-sequence, i.e.
      - left sub-tree – root – right sub-tree
    - and return all visited knots, as long as key $\leq y_u$.
      - This is assumed to be known and will not be detailed here.
    - Auxiliary function `Successor(knot q)`: Successors of knot q in in-order-sequence.

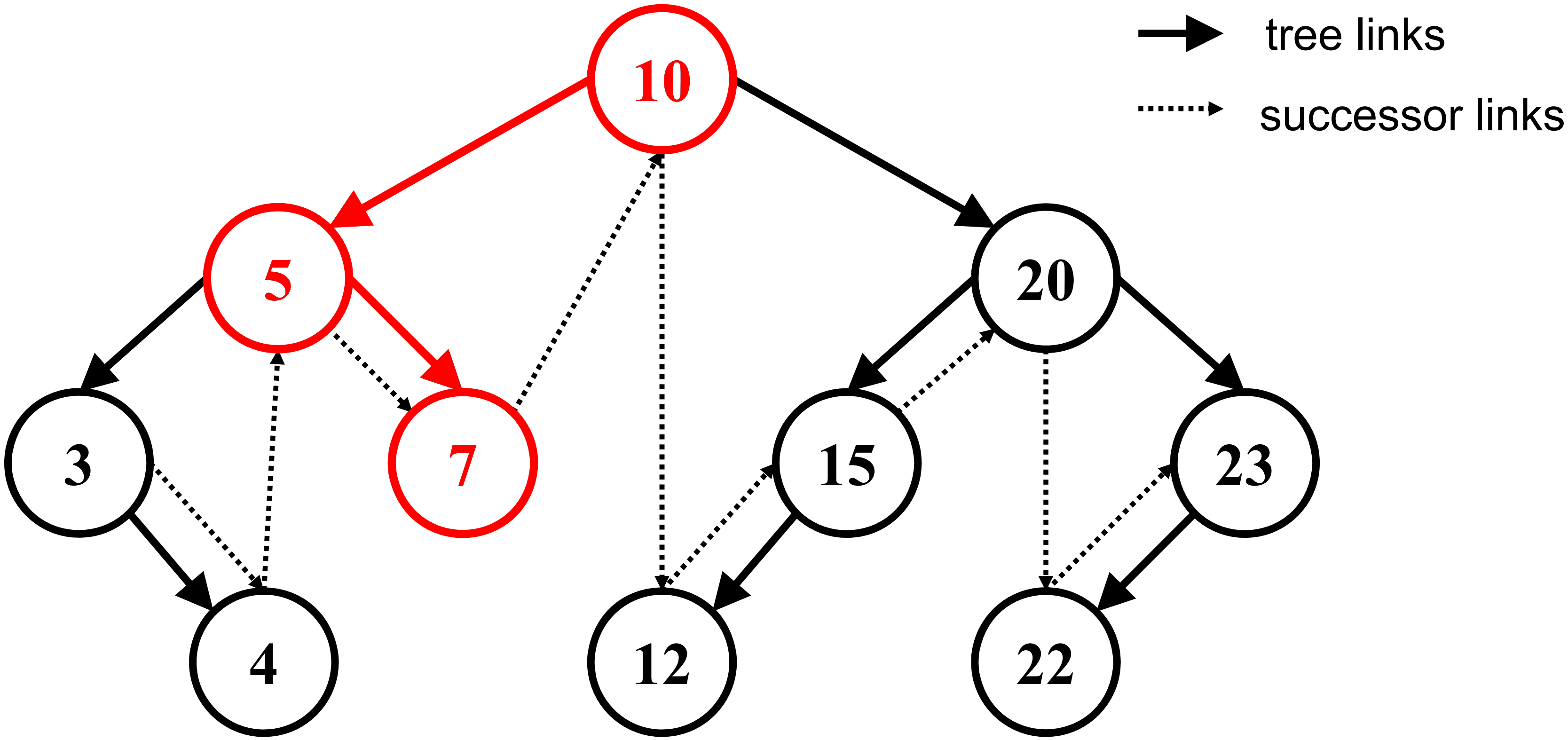
2.3.1 Iso-oriented segments

Algorithm 2: Find knot *p* with smallest key ≥ *x*

```
p = root; q = NULL;
while ((p != NULL) && (p.key != x)) {
   q = p;
   if (x < p.key) then p = p.leftson;
   else                p = p.rightson;
}
if (p    != NULL) then return p;     // key==x found
if (root == NULL) then return NULL;  // empty tree
if (q    == root) then       // tree has only one knot
   return NULL or root depending on root.key;
if (x < q.key) then return q;
else                return Successor(q);
```

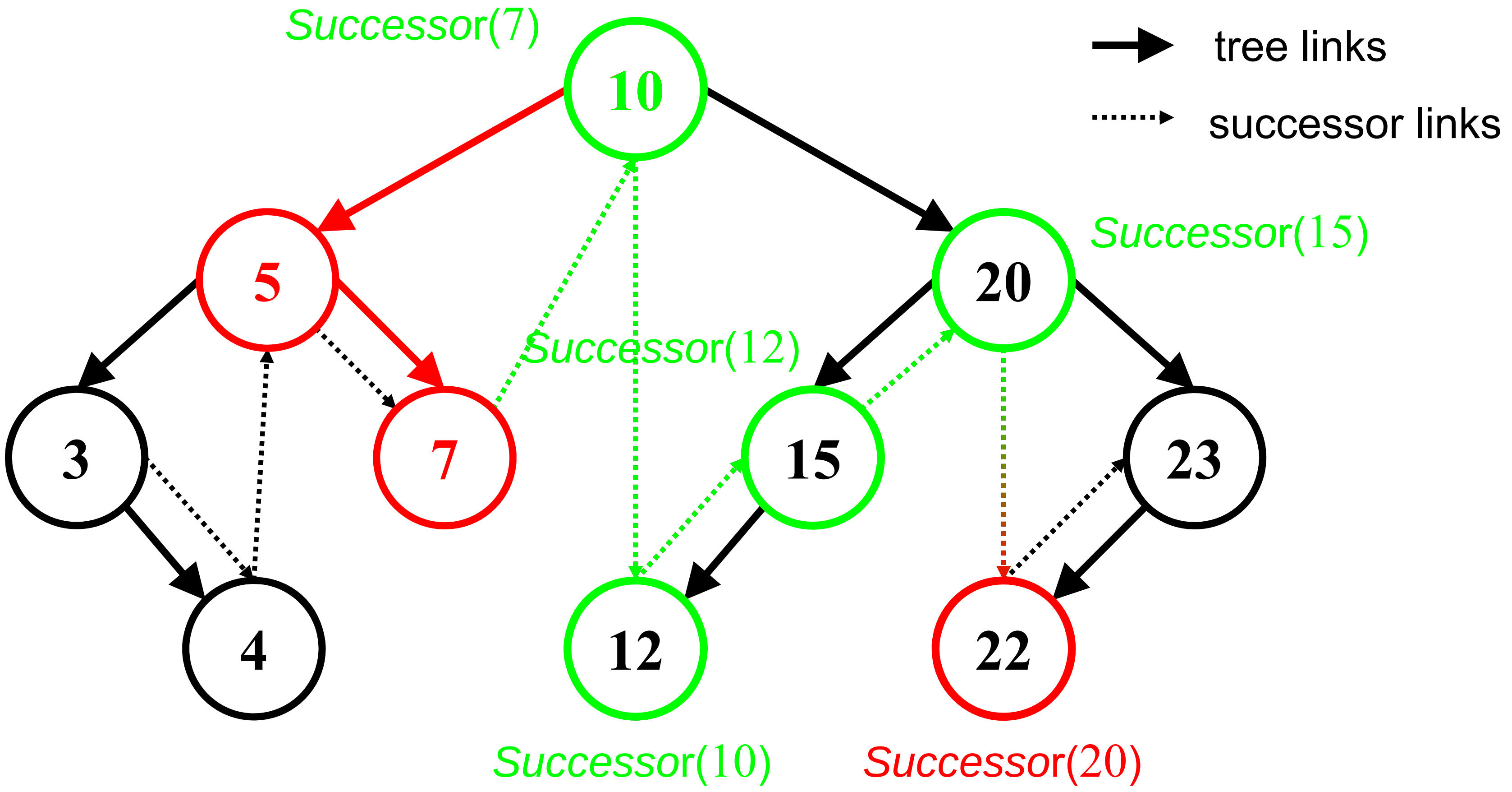
2.3.1 Iso-oriented segments

- Example: Search knots with keys in $[8,21]$



2.3.1 Iso-oriented segments

- <u>Example:</u> Search knots with keys in $[8,21]$



2.3.1 Iso-oriented segments

- Run time
  - Range search
    - Let r be the number of elements in $[y_l, y_u]$
    - **Search** for y_l: $O(\log n)$
    - **Traverse** the tree to knot $> y_u$: $O(r)$
    - **Total:** $O(\log n + r)$
  - Total scan-line run time
    - **Sorting** of the event points: $O(n \log n)$.
    - **Scanning:** For every event point p_i we need $O(\log n + r_i)$, where r_i is the number of intersections at p_i.
    - **Total:** $O(n \log n + R)$, where R is the total number of intersections.

2.3.1 Iso-oriented segments

Remarks

- If R grows slower than quadratically, the scan-line algorithm is superior to the naive approach!
- One can show, that in the worst case $\Omega(n \log n + R)$ steps are necessary to return all intersections.

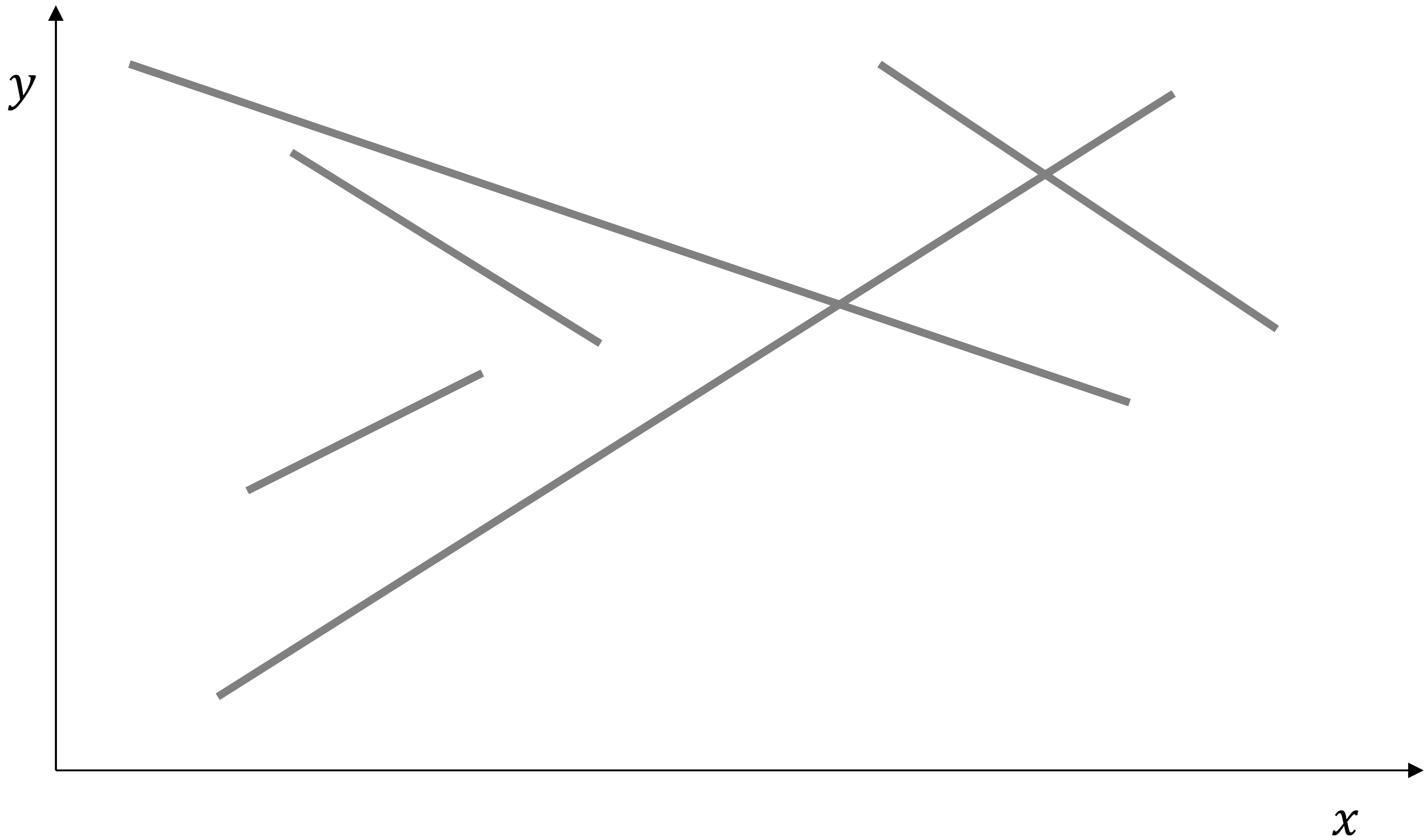
➡ Scan-line-algorithms for iso-oriented line segments are *worst-case-optimal*.

2.3.2 Arbitrary segments

- Now, arbitrary line segments in general position.
- **Input:** Set $S = \{s_1, \ldots, s_n\}$ of n line segments in 2D.
- **Goal:** Set of all *true intersections*, i.e. intersections, that are not end points.
- Assumption: *General position of the segments*
  - No segment is vertical.
  - The intersection of two segments is either empty or exactly one point.
  - In one point no more than two segments intersect.
  - All end- and intersection-points have mutually distinct x-coordinates.

2.3.2 Arbitrary segments

- Example



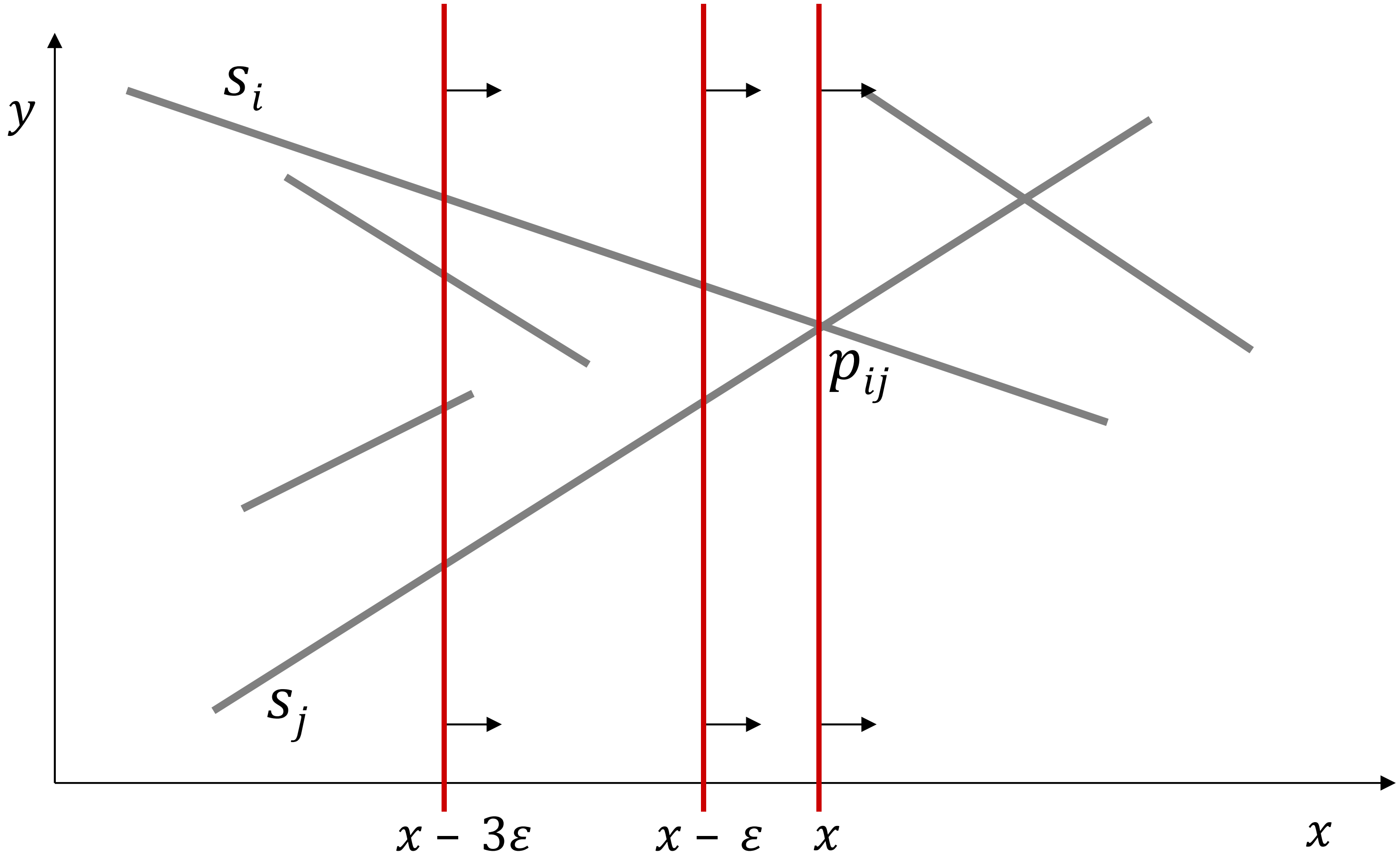
2.3.2 Arbitrary segments

- Approach
  - The principle is the same as for iso-oriented line segments.
  - Scan-line from left to right.

- Observation 1
  - Let s_i and s_j be two line segments, intersecting in a point $p_{ij} = (x, y)$.
  - Then s_i and s_j are adjacent on the scan-line, if the scan-line is at $x - \varepsilon$ for a sufficiently small and positive ε.

2.3.2 Arbitrary segments

- Example

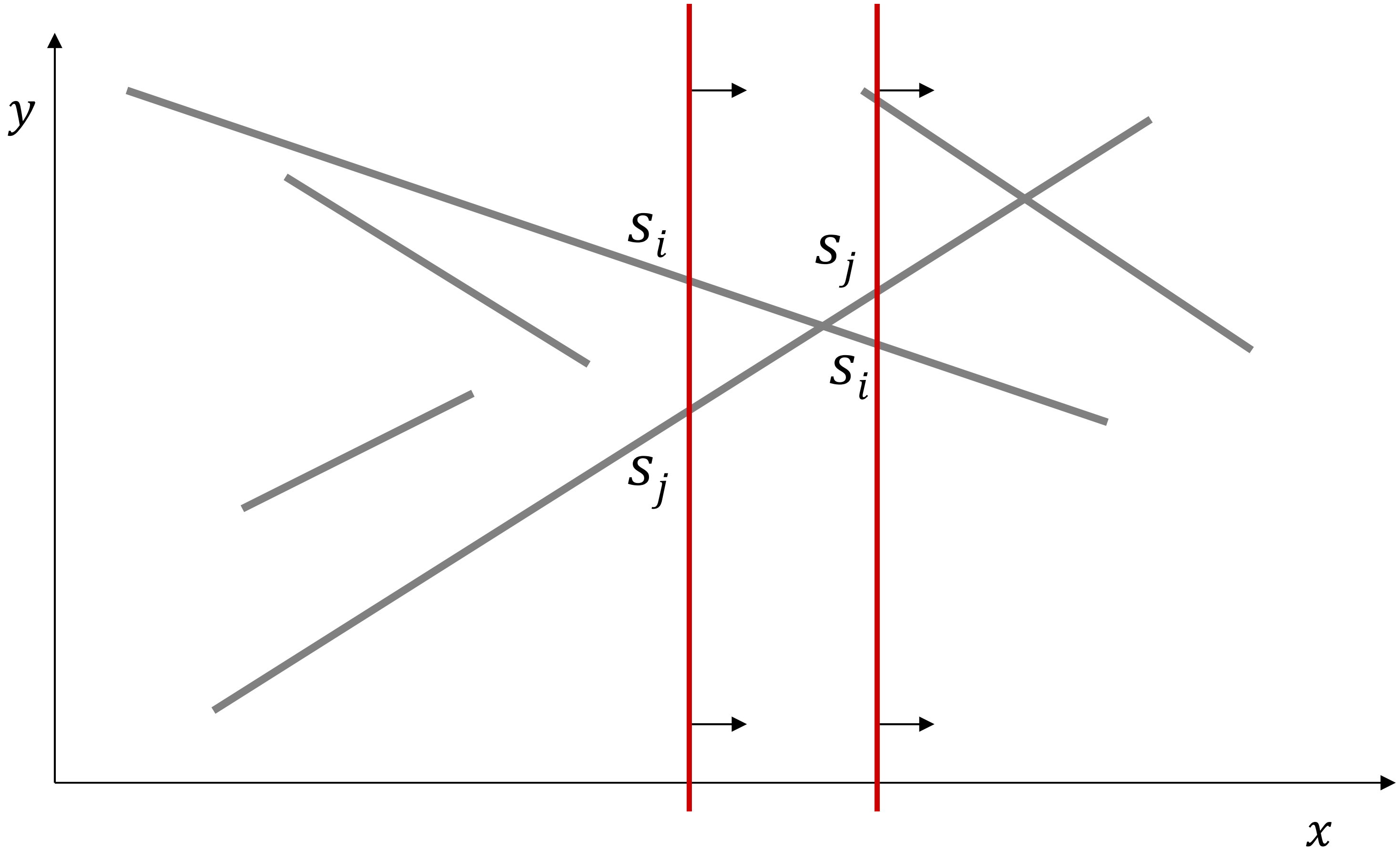


2.3.2 Arbitrary segments

- Observation 2
  - At an intersection of two line segments the order of these two segments is reversed on the scan-line.
- Consequence
  - When an intersection is reached, the event structure needs to be updated to correct the sequence of segments on the scan-line.
    - ➡ Also intersections are event points.
    - ➡ Intersections must be added to the event structure on runtime.

2.3.2 Arbitrary segments

- Example



2.3.2 Arbitrary segments

- There are three types of events:
  1. Start point of a segment: known a priori,
  2. End point of a segment: known a priori,
  3. Intersection point of two segments: computed on runtime.
- The data structure event E represents one event and contains:
  - x-coordinates,
  - identifiers of the relevant segments:
    - one segment ID for start and end points,
    - two segment ID's for intersection points.

2.3.2 Arbitrary segments

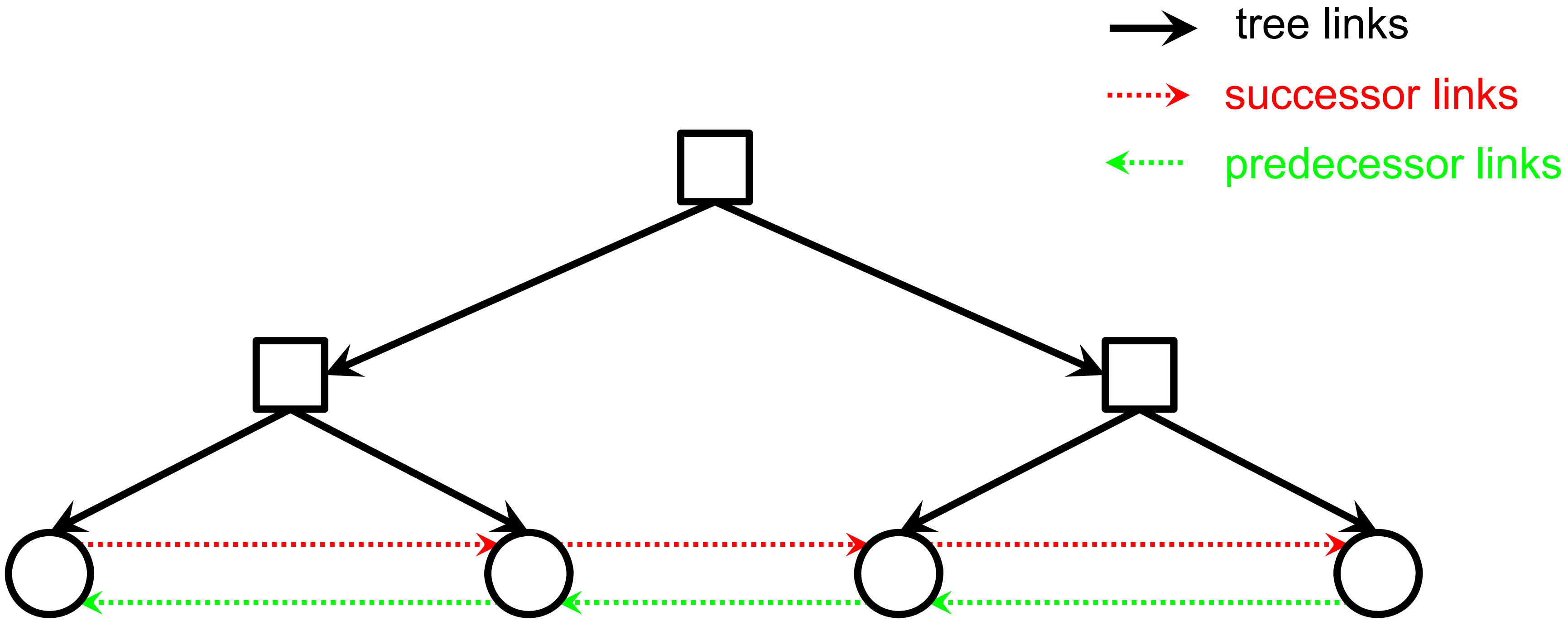
- Operations for the event data structure ES, that maintains the events:
  - `NextEvent()`: Returns event with smallest x-coordinate from ES and removes it.
  - `AddEvent(Event E)`: Inserts a new event to ES according to its x-coordinate.
- Implementation of ES:
  - Heap or balanced tree.
  - Run time of both operations: $O(\log |ES|)$.

2.3.2 Arbitrary segments

- *Scan-Line-State-Structure* SSS
  - Stores segments ordered by y-coordinate of the actual intersection of the active segments with the scan-line L.
  - Operations
    - `AddSegment(Segment` s`):` Adds segment s to SSS.
    - `RemoveSegment(Segment` s`):` Removes segment s from SSS.
    - `Pred(Segment` s`):` Returns predecessor of s on L.
    - `Succ(Segment` s`):` Returns successor of s on L.
    - `Exchange(Segment` s_1`,Segment` s_2`):` Exchanges the sequence of two segments s_1 and s_2 on L.

2.3.2 Arbitrary segments

- Implementation of SSS
  - Leaf-oriented, balanced, binary search tree with additional links for successors and predecessors.



2.3.2 Arbitrary segments

- Auxiliary function for intersection-events

  `TestIntersectionGenerateEvent(`s_1, s_2`):`

  - Takes two adjacent segments s_1 and s_2 from L.
  - Tests, if they intersect.
    - If yes, a new event is generated with x-coordinate of the intersection and ID's of the intersecting segments.
    - Otherwise, the empty event is returned.

2.3.2 Arbitrary segments

Algorithm 3: Intersect arbitrary line segments (Part 1)

Input: n line segments in general position.
Output: All intersections.

```
Initialize ES and SSS;
Sort the 2n end point by y-coordinate;
Store the resulting events in ES;
while (ES is not empty) {
    E = ES.NextEvent();
    if (E is start of a segment) then {
        SSS.AddSegment(E.Segment);
        VS = SSS.Pred(E.Segment);
        E' = TestIntersectionGenerateEvent(E,VS);
        if (E' ≠ ∅) then ES.AddEvent(E');
        NS = SSS.Succ(E.Segment);
        E' = TestIntersectionGenerateEvent(E,NS);
        if (E' ≠ ∅) then ES.AddEvent(E');
    }
⋮
}
```

2.3.2 Arbitrary segments

Algorithm 3: Intersect arbitrary line segments (Part 2)

Input: n line segments in general position.
Output: All intersections.

```
⋮
⋮
while (ES is not empty) {
   E = ES.NextEvent();
⋮
   if (E is end of a segment) then {
      VS = SSS.Pred(E.Segment);
      NS = SSS.Succ(E.Segment);
      SSS.Remove(E.Segment);
      E' = TestIntersectionGenerateEvent(VS,NS);
      if (E' ≠ Ø) then ES.AddEvent(E');
   }
⋮
⋮
⋮
}
```

2.3.2 Arbitrary segments

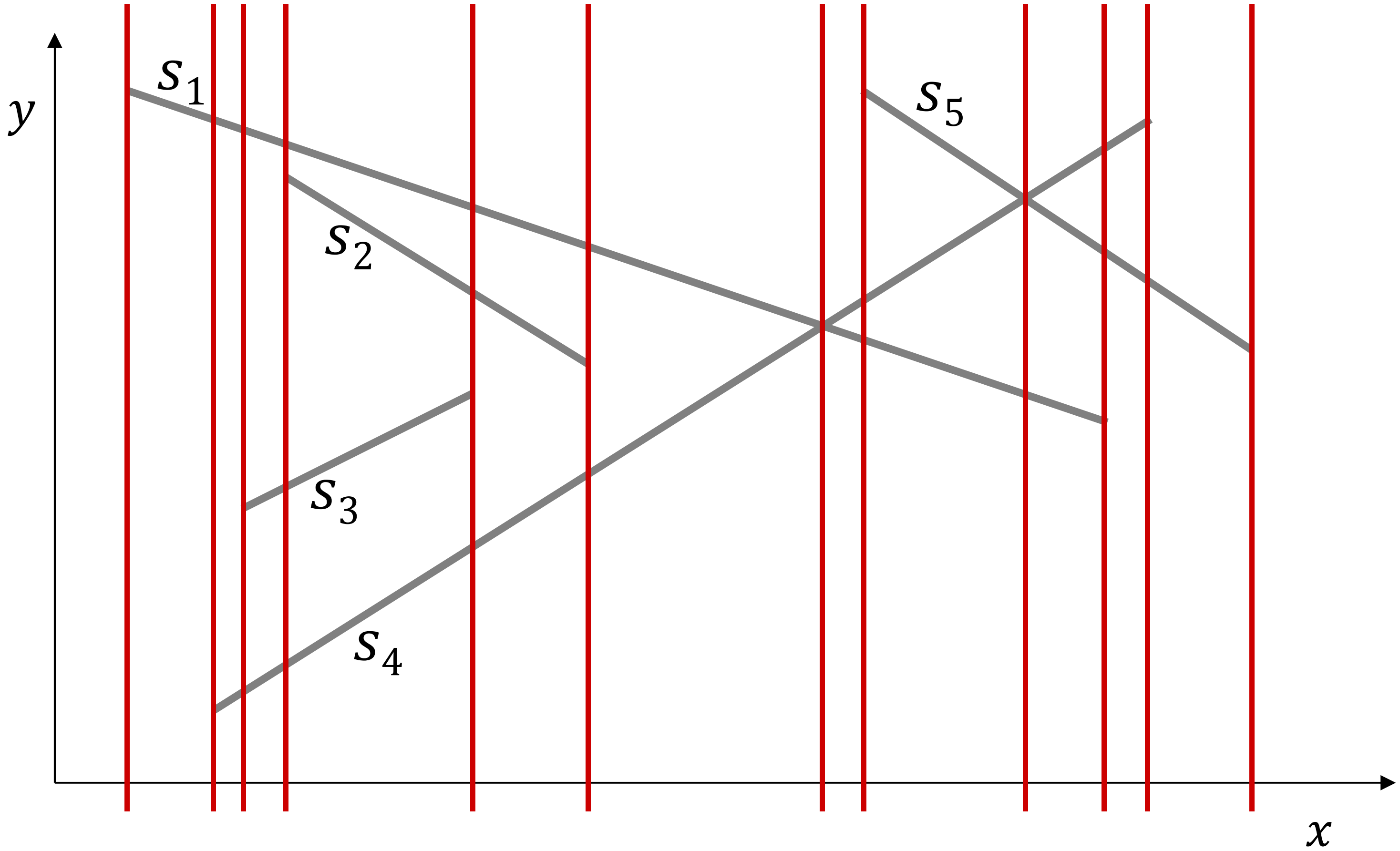
Algorithm 3: Intersect arbitrary line segments (Part 3)

Input: *n* line segments in general position.
Output: All intersections.

```
⋮
⋮
while (ES is not empty) {
   E = ES.NextEvent();
⋮
   if (E is intersection) then {
      Report Intersection of E.SegmentU and E.SegmentL;
      SSS.Exchange(E.SegmentU,E.SegmentL);
      VS = SSS.Pred(E.SegmentU);
      E' = TestIntersectionGenerateEvent(E.SegmentU,VS);
      if (E' ≠ ∅) then ES.AddEvent(E');
      NS = SSS.Succ(E.SegmentL);
      E' = TestIntersectionGenerateEvent(E.SegmentL,NS);
      if (E' ≠ ∅) then ES.AddEvent(E');
   }
}
```

2.3.2 Arbitrary segments

- Example



2.3.2 Arbitrary segments

- Run time
  - Sorting of the $2n$ end points by their x-coordinate: $O(n \log n)$.
  - Total number of events: $2n + k$ (k is the number of intersections).
    - ➡ The outer loop is repeated at most $O(2n + k)$ times.
    - Per iteration there are at most six operations on ES and SSS.
  - **Runtime of *ES*-Operations:** There are at most $O(n^2)$ events at the same time in ES, because there are at most $O(n^2)$ intersections.
    - ➡ Every operation on ES takes $O(\log n^2) = O(\log n)$.
  - **Runtime of *SSS*-Operations:** There are at most n elements in SSS.
    - ➡ Every operation on SSS takes $O(\log n)$.

2.3.2 Arbitrary segments

- Total run time
  - $O((n+k)\log n)$
- Memory
  - $O(n^2)$, because there are at most quadratically many intersection in ES.

2.4 Literature

[1] Aho, Hopcroft, Ullmann: *Data structures and algorithms*, Addison-Wesley, 1982.

[2] Ottmann, Widmayer: *Algorithmen und Datenstrukturen*, Spektrum akademischer Verlag, 2. Auflage, 1996.

[3] Bentley, Ottmann: *Algorithms for reporting and counting geometric intersections*. IEEE Trans. Comput., C-28: 643-647, 1979.