Convex Hull Algorithms

Claudio Mirolo

Dip. di Scienze Matematiche, Informatiche e Fisiche
Università di Udine, via delle Scienze 206 – Udine
claudio.mirolo@uniud.it

Computational Geometry
www.dimi.uniud.it/claudio
Outline

1. Incremental algorithm
   - degeneracies
   - correctness
   - computational costs

2. Divide-et-impera algorithm
   - recursive approach
   - correctness
   - computational costs

3. Randomized algorithm
   - conflict graph
   - correctness
   - computational costs
Given a set $P$ of $n$ points in the plane (space)

“Smaller” convex region containing all points in $P$

Region = convex polygon (polyhedron)
Given a set $P$ of $n$ points in the plane (space)

“Smaller” convex region containing all points in $P$

Region = convex polygon (polyhedron)
Given a set $P$ of $n$ points in the plane (space)

“Smaller” convex region containing all points in $P$

Region = convex polygon (polyhedron)
Outline

1. Incremental algorithm
   - degeneracies
   - correctness
   - computational costs

2. Divide-et-impera algorithm
   - recursive approach
   - correctness
   - computational costs

3. Randomized algorithm
   - conflict graph
   - correctness
   - computational costs
Graham’s scan

- Lower hull first

- Adding points one by one, sorted left-to-right

- Lower hull is updated after each addition
Graham’s scan

- Lower hull first

- Adding points one by one, *sorted left-to-right*

- Lower hull is updated after each addition
Graham’s scan

- Lower hull first
- Adding points one by one, *sorted left-to-right*
- Lower hull is updated after each addition
Scan step

- From \( \text{lower}_\text{hull}( \{ p_1, p_2, p_3, \ldots, p_{i-1} \} ) \) \ldots

- \ldots to \( \text{lower}_\text{hull}( \{ p_1, p_2, p_3, \ldots, p_{i-1}, p_i \} ) \)

- Basic idea: Animation
Scan step

- From $\text{lower\_hull}(\{p_1, p_2, p_3, \ldots, p_{i-1}\})$ ...

- ... to $\text{lower\_hull}(\{p_1, p_2, p_3, \ldots, p_{i-1}, p_i\})$

- Basic idea: Animation
Scan step

- From $lower\_hull( \{ p_1, p_2, p_3, \ldots, p_{i-1} \} )$ ...
- ... to $lower\_hull( \{ p_1, p_2, p_3, \ldots, p_{i-1}, p_i \} )$

- Basic idea: Animation
Lower hull

- After the $i$-th step, $p_1$ and $p_i$ belong to the lower hull
- $p_1$ and $p_n$ belong to both the lower and upper hull
- Lower hull construction: Code
Lower hull

- After the $i$-th step, $p_1$ and $p_i$ belong to the lower hull
- $p_1$ and $p_n$ belong to both the lower and upper hull

Lower hull construction: Code
After the $i$-th step, $p_1$ and $p_i$ belong to the lower hull

$p_1$ and $p_n$ belong to both the lower and upper hull

Lower hull construction: Code
Upper hull: similar processing, by symmetry

E.g., adding points in backward order

Upper hull construction: Code
Upper hull

- Upper hull: similar processing, by symmetry
- E.g., adding points in backward order

Upper hull construction: Code
Upper hull

- Upper hull: similar processing, by symmetry
- E.g., adding points in backward order
- Upper hull construction: Code
Degeneracies: Vertically aligned points

- **Lexicographic order**
  - Geometric interpretation of lexicographic order
  - “Symbolic perturbation”
Degeneracies: Vertically aligned points

- **Lexicographic order**

- Geometric interpretation of lexicographic order

- “Symbolic perturbation”
Degeneracies: Vertically aligned points

- Lexicographic order
- Geometric interpretation of lexicographic order
- “Symbolic perturbation”
Geometric interpretation

Convex Hull
Geometric interpretation
Degeneracies: Collinear hull vertices

- **Strict** left turn
- Which is the result?
- Code details...
Degeneracies: Collinear hull vertices

- \textit{Strict} left turn

- Which is the result?

- Code details...
Degeneracies: Collinear hull vertices

- *Strict* left turn
- Which is the result?
- *Code* details...
Robustness?

- What about inaccurate floating-point calculations?
  - *Structural integrity* is preserved!
  - ... As opposed to the "brute force" algorithm
Robustness?

- What about inaccurate floating-point calculations?

- *Structural integrity* is preserved!

- ... As opposed to the “brute force” algorithm
Robustness?

- What about inaccurate floating-point calculations?

- *Structural integrity* is preserved!

- ... As opposed to the “brute force” algorithm
Iteration invariants

Lower hull construction invariant:

- After $i$-th step: $P_i = \{ p_1, p_2, p_3, \ldots, p_{i-1}, p_i \}$
- No point of $P_i$ lies below (to the right of) $\text{lower}_\text{hull}(P_i)$
- $\text{lower}_\text{hull}(P_i)$ is convex (left turn at each vertex)
Lower hull construction invariant:

- After $i$-th step: $P_i = \{ p_1, p_2, p_3, \ldots, p_{i-1}, p_i \}$
- No point of $P_i$ lies below (to the right of) $\text{lower}_\text{hull}(P_i)$
- $\text{lower}_\text{hull}(P_i)$ is convex (left turn at each vertex)
Iteration invariants

Lower hull construction invariant:

- After \( i \)-th step: \( P_i = \{ p_1, p_2, p_3, \ldots, p_{i-1}, p_i \} \)
- No point of \( P_i \) lies below (to the right of) \( \text{lower}_\text{hull}( P_i ) \)
- \( \text{lower}_\text{hull}( P_i ) \) is convex (left turn at each vertex)
Iteration invariants

Lower hull construction invariant:

- After $i$-th step: $P_i = \{ p_1, p_2, p_3, \ldots, p_{i-1}, p_i \}$
- No point of $P_i$ lies below (to the right of) $\text{lower}_hull(P_i)$
- $\text{lower}_hull(P_i)$ is convex (left turn at each vertex)
Proof: Induction step

lower_hull(P_{i-1}) convex

no point of P_{i-1} here

no point in vertical slab between p_{i-1} and p_i
Proof: Induction step

\[ \text{Proof: Induction step} \]

**lower hull** \( P_{i-1} \) convex

no point of \( P_{i-1} \) here

no point in vertical slab between \( p_{i-1} \) and \( p_i \)

\( \text{Proof: Induction step} \)
Computational costs

- Sorting points in lexicographic order: $O(n \log n)$
- Adding points incrementally: $O(n)$ for iterations
- Updating lower/upper hull: $O(n)$ while iterations...
- ... over all for iterations (at each further iteration a hull vertex is removed!)
Computational costs

- Sorting points in lexicographic order: $O(n \log n)$
- Adding points incrementally: $O(n)$ for iterations
- Updating lower/upper hull: $O(n)$ while iterations...
- ... over all for iterations
  (at each further iteration a hull vertex is removed!)
Computational costs

- Sorting points in lexicographic order: $O(n \log n)$
- Adding points incrementally: $O(n)$ for iterations
- Updating lower/upper hull: $O(n)$ while iterations...

... over all for iterations
(at each further iteration a hull vertex is removed!)
Computational costs

- Sorting points in lexicographic order: $O(n \log n)$

- Adding points incrementally: $O(n)$ for iterations

- Updating lower/upper hull: $O(n)$ while iterations

- ... over all for iterations
  (at each further iteration a hull vertex is removed!)
May any better algorithm be conceived?

- Sorting $x_1, x_2, \ldots x_n$ can be reduced to convex hull.
- Consider point set $P = \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$
- Convex hull (sorted vertices): $n \log n$ lower bound
May any better algorithm be conceived?

**Sorting** $x_1, x_2, \ldots, x_n$ can be reduced to *convex hull*...

Consider point set $P = \{ (x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2) \}$

Convex hull (sorted vertices): $n \log n$ lower bound
Optimality of $n \log n$

- May any better algorithm be conceived?
- **Sorting** $x_1, x_2, \ldots, x_n$ can be reduced to convex hull...
- Consider point set $P = \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$
- Convex hull (sorted vertices): $n \log n$ lower bound
May any better algorithm be conceived?

*Sort* $x_1, x_2, \ldots, x_n$ can be reduced to convex hull.

Consider point set $P = \{ (x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2) \}$

Convex hull (sorted vertices): $n \log n$ lower bound
Outline

1. Incremental algorithm
   - degeneracies
   - correctness
   - computational costs

2. Divide-et-impera algorithm
   - recursive approach
   - correctness
   - computational costs

3. Randomized algorithm
   - conflict graph
   - correctness
   - computational costs
Preliminarily, points are sorted lexicographically

- Balanced bipartition through a vertical line
- Convex hull of the left half (recursively)
- Convex hull of the right half (recursively)
Preparata & Hong’s recursive approach

- Preliminarily, points are sorted lexicographically
- Balanced bipartition through a vertical line
  - Convex hull of the left half (recursively)
  - Convex hull of the right half (recursively)
Preparata & Hong’s recursive approach

- Preliminarily, points are sorted lexicographically
- Balanced bipartition through a vertical line
- Convex hull of the left half (recursively)
- Convex hull of the right half (recursively)
Preparata & Hong’s recursive approach

- Preliminarily, points are sorted lexicographically
- Balanced bipartition through a vertical line
- Convex hull of the left half (recursively)
- Convex hull of the right half (recursively)
Preparata & Hong’s recursive approach

- Boundary walks to draw the connecting edges (common tangent lines above and below)
- Cut & sew appropriate (half)chains of points and connecting edges
- Animation / Code
Preparata & Hong’s recursive approach

- Boundary walks to draw the connecting edges (common tangent lines above and below)
- Cut & sew appropriate (half)chains of points and connecting edges
- Animation / Code
Preparata & Hong’s recursive approach

- Boundary walks to draw the connecting edges (common tangent lines above and below)
- Cut & sew appropriate (half) chains of points and connecting edges
- Animation / Code
Correctness

- Recursive constructions are assumed to be correct (and base cases are)

- Walk(s) to determine connecting edges must come to an end and the resulting chain will be convex
Correctness

- Recursive constructions are assumed to be correct (and base cases are)

- Walk(s) to determine connecting edges must come to an end and the resulting chain will be convex
Computational costs

- **Walks + cut & sew:** $O(n)$

- **Well known equation:** $T(n) = 2T(n/2) + O(n)$

- **Solution:** $T(n) = O(n \log n)$
Computational costs

- Walks + cut & sew: $O(n)$

- Well known equation: $T(n) = 2T(n/2) + O(n)$

- Solution: $T(n) = O(n \log n)$
Computational costs

- Walks + cut & sew: \( O(n) \)
- Well known equation: \( T(n) = 2 T(n/2) + O(n) \)
- Solution: \( T(n) = O(n \log n) \)
Outline

1. Incremental algorithm
   - degeneracies
   - correctness
   - computational costs

2. Divide-et-impera algorithm
   - recursive approach
   - correctness
   - computational costs

3. Randomized algorithm
   - conflict graph
   - correctness
   - computational costs
Randomization and conflict graph

Conflict-graph framework:

- **Objects**: problem input data — set $S$
- **Regions**: identified by $O(1)$ objects
- **Conflicts**: relationship between regions and objects

Result: Set of regions defined by the objects in $S$ without conflicts with these objects
Randomization and conflict graph

Conflict-graph framework:

- **Objects**: problem input data — set $S$
- **Regions**: identified by $O(1)$ objects
- **Conflicts**: relationship between regions and objects

Result: Set of regions defined by the objects in $S$ without conflicts with these objects
Randomization and conflict graph

Conflict-graph framework:

- **Objects**: problem input data — set $S$
- **Regions**: identified by $O(1)$ objects
- **Conflicts**: relationship between regions and objects

Result: Set of regions defined by the objects in $S$ without conflicts with these objects
Randomization and conflict graph

Conflict-graph framework:

- **Objects**: problem input data — set $S$
- **Regions**: identified by $O(1)$ objects
- **Conflicts**: relationship between regions and objects

Result: Set of regions defined by the objects in $S$ without conflicts with these objects
Randomization and conflict graph

Conflict-graph framework:

- **Objects**: problem input data — set $S$
- **Regions**: identified by $O(1)$ objects
- **Conflicts**: relationship between regions and objects

Result: Set of regions defined by the objects in $S$ *without* conflicts with these objects
Assumptions (*static* approach):

- Links in both directions between regions and objects *in conflict*
- Direct link from object to conflict region
- Direct link from region to entry point ("iterator") to list of conflicting objects
Conflict-graph

Assumptions (*static* approach):

- Links in both directions between regions and objects *in conflict*
- Direct link from object to conflict region
- Direct link from region to entry point ("iterator") to list of conflicting objects
Conflict-graph

Assumptions (*static* approach):

- Links in both directions between regions and objects *in conflict*
- Direct link from object to conflict region
- Direct link from region to entry point ("iterator") to list of conflicting objects
Conflict-graph as a general framework

Simple example: *Sorting* numbers

- **Objects**: real numbers $x_i$ from finite set $X$
- **Regions**: intervals $[x_i, x_j]$ between (two) numbers from $X$
- **Conflicts**: $[x_i, x_j]$ and $x$ are in conflict if $x \in ]x_i, x_j[$

Result: pairs of numbers identifying intervals without conflicts are consecutive in the sorted sequence of $X$’s elements
Simple example: Sorting numbers

- **Objects**: real numbers \( x_i \) from finite set \( X \)
- **Regions**: intervals \([x_i, x_j]\) between (two) numbers from \( X \)
- **Conflicts**: \([x_i, x_j]\) and \( x \) are in conflict if \( x \in ]x_i, x_j[ \)

Result: pairs of numbers identifying intervals without conflicts are consecutive in the sorted sequence of \( X \)'s elements
Conflict-graph as a general framework

Simple example: Sorting numbers

- **Objects**: real numbers $x_i$ from finite set $X$
- **Regions**: intervals $[x_i, x_j]$ between (two) numbers from $X$
- **Conflicts**: $[x_i, x_j]$ and $x$ are in conflict if $x \in ]x_i, x_j[$

Result: pairs of numbers identifying intervals without conflicts are consecutive in the sorted sequence of $X$’s elements
Conflict-graph as a general framework

Simple example: *Sorting* numbers

- **Objects**: real numbers $x_i$ from finite set $X$
- **Regions**: intervals $[x_i, x_j]$ between (two) numbers from $X$
- **Conflicts**: $[x_i, x_j]$ and $x$ are in conflict if $x \in ]x_i, x_j[$

Result: pairs of numbers identifying intervals without conflicts are consecutive in the sorted sequence of $X$’s elements
Simple example: *Sorting* numbers

- **Objects:** real numbers $x_i$ from finite set $X$
- **Regions:** intervals $[x_i, x_j]$ between (two) numbers from $X$
- **Conflicts:** $[x_i, x_j]$ and $x$ are in conflict if $x \in ]x_i, x_j[$

Result: pairs of numbers identifying intervals without conflicts are consecutive in the sorted sequence of $X$’s elements
Conflict-graph as a general framework

Simple example: A different point of view on quick-sort?

- **Objects**: real numbers \( x_i \) from finite set \( X \)
- **Regions**: intervals \([x_i, x_j]\) between (two) numbers from \( X \)
- **Conflicts**: \([x_i, x_j]\) and \( x \) are in conflict if \( x \in ]x_i, x_j[\)

Result: pairs of numbers identifying intervals without conflicts are consecutive in the sorted sequence of \( X \)’s elements
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**: points $p_i$ from finite set $P$
- **Regions**: outer “sector” of (current) convex hull edge $p_ip_j$
- **Conflicts**: when $p \in P$ falls in one such outer sector

Result: edge chain $H$ such that no point $p \in P$ lies outside $H$

… Animation … incremental approach, indeed
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**: points $p_i$ from finite set $P$
- **Regions**: outer “sector” of (current) convex hull edge $p_ip_j$
- **Conflicts**: when $p \in P$ falls in one such outer sector

Result: edge chain $H$ such that no point $p \in P$ lies outside $H$

... Animation ... incremental approach, indeed
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**: points \( p_i \) from finite set \( P \)
- **Regions**: outer “sector” of (current) convex hull edge \( p_ip_j \)
- **Conflicts**: when \( p \in P \) falls in one such outer sector

Result: edge chain \( H \) such that no point \( p \in P \) lies outside \( H \)
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**: points \( p_i \) from finite set \( P \)
- **Regions**: outer “sector” of (current) convex hull edge \( p_ip_j \)
- **Conflicts**: when \( p \in P \) falls in one such outer sector

Result: edge chain \( H \) such that no point \( p \in P \) lies outside \( H \)
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**: points $p_i$ from finite set $P$
- **Regions**: outer “sector” of (current) convex hull edge $p_ip_j$
- **Conflicts**: when $p \in P$ falls in one such outer sector

Result: edge chain $H$ such that no point $p \in P$ lies outside $H$

... Animation ... incremental approach, indeed
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**: points $p_i$ from finite set $P$
- **Regions**: outer “sector” of (current) convex hull edge $p_ip_j$
- **Conflicts**: when $p \in P$ falls in one such outer sector

Result: edge chain $H$ such that no point $p \in P$ lies outside $H$

... Animation ... incremental approach, indeed
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**: points $p_i$ from finite set $P$
- **Regions**: outer “sector” of (current) convex hull edge $p_ip_j$
- **Conflicts**: when $p \in P$ falls in one such outer sector

Result: edge chain $H$ such that no point $p \in P$ lies outside $H$

... Animation ... incremental approach, indeed
Randomized incremental convex hull algorithm
Randomized incremental convex hull algorithm
Randomized incremental convex hull algorithm
Randomized incremental convex hull algorithm
Randomized incremental convex hull algorithm
Randomized incremental convex hull algorithm
Randomized incremental convex hull algorithm
Conflict-graph approach to the convex hull

Conflict-graph framework:

- **Objects**
- **Regions**
- **Conflicts**

Result: no point \( p \in P \) lies outside \( H_n \)
Points $p_i$ from finite set $P$

Conflict-graph framework:

- **Objects**
- **Regions**
- **Conflicts**

Result: no point $p \in P$ lies outside $H_n$
Outer sector of hull edge $p_ip_j$

Conflict-graph framework:
- **Objects**
- **Regions**
- **Conflicts**

Result: no point $p \in P$ lies outside $H_n$
\( p \in P \) falls in one such outer sector

Conflict-graph framework:
- **Objects**
- **Regions**
- **Conflicts**

Result: no point \( p \in P \) lies outside \( H_n \)
$p \in P$ falls in one such outer sector

Conflict-graph framework:
- **Objects**
- **Regions**
- **Conflicts**

Result: no point $p \in P$ lies outside $H_n$
Correctness

- Straightforward

- Walk(s) to get rid of non-convex vertices must come to an end and the updated hull will be convex
Correctness

- Straightforward

- Walk(s) to get rid of non-convex vertices must come to an end and the updated hull will be convex
At each stage, to update the convex hull from $H_k$ to $H_{k+1}$, the algorithm accomplishes three main tasks:

- It removes a few “old” edges from $H_k$
- It adds two “new” edges to build $H_{k+1}$
- It re-arranges the links of the conflict graph between the outer regions of the new edges and all the points in the conflict lists of the removed edges
At each stage, to update the convex hull from $H_k$ to $H_{k+1}$, the algorithm accomplishes three main tasks:

- It removes a few “old” edges from $H_k$
- It adds two “new” edges to build $H_{k+1}$
- It re-arranges the links of the conflict graph between the outer regions of the new edges and all the points in the conflict lists of the removed edges
At each stage, to update the convex hull from $H_k$ to $H_{k+1}$, the algorithm accomplishes three main tasks:

- It removes a few “old” edges from $H_k$
- It adds two “new” edges to build $H_{k+1}$
- It re-arranges the links of the conflict graph between the outer regions of the new edges and all the points in the conflict lists of the removed edges
Removing and creating edges

- Only edges that have been created can be removed
- Two edges are created at each of the $O(n)$ stages
- Hence, the overall costs of both removing and creating edges are bound by $O(n)$
Removing and creating edges

- Only edges that have been created can be removed.
- Two edges are created at each of the $O(n)$ stages.
- Hence, the overall costs of both removing and creating edges are bound by $O(n)$. 
Removing and creating edges

- Only edges that have been created can be removed
- Two edges are created at each of the $O(n)$ stages
- Hence, the overall costs of both removing and creating edges are bound by $O(n)$
Updating the conflict graph

- Expected cost
  - Approach: backward analysis
  - Points which do not fall outside $H_k$ at some stage will no longer be taken into account
  - Focus on a set $P_k \subset P$ of $k$ points ($k$-th stage) ...
Updating the conflict graph

- Expected cost
- Approach: backward analysis
  - Points which do not fall outside $H_k$ at some stage will no longer be taken into account
  - Focus on a set $P_k \subset P$ of $k$ points ($k$-th stage) . . .
Updating the conflict graph

- Expected cost

- Approach: backward analysis

- Points which do not fall outside $H_k$ at some stage will no longer be taken into account

- Focus on a set $P_k \subset P$ of $k$ points ($k$-th stage) ...
Expected cost

Approach: backward analysis

Points which do not fall outside $H_k$ at some stage will no longer be taken into account

Focus on a set $P_k \subset P$ of $k$ points ($k$-th stage) . . .
Because of the randomization, each of the $k$ points in $P_k$ may be the last added with equal probability $1/k$.

Let $p$ be this last added point.

Then, either $p$ is interior to $H_k$ (no related processing) or $p$ is a vertex of $H_k$ with incident edges $e'$ and $e''$.

Graph links re-arranged at the $k$-th stage (only) for points in the conflict lists of $e'$ and $e''$ . . .
Because of the randomization, each of the $k$ points in $P_k$ may be the last added with equal probability $1/k$.

Let $p$ be this last added point.

Then, either $p$ is interior to $H_k$ (no related processing) or $p$ is a vertex of $H_k$ with incident edges $e'$ and $e''$.

Graph links re-arranged at the $k$-th stage (only) for points in the conflict lists of $e'$ and $e''$ ...
Backward analysis

- Because of the randomization, each of the $k$ points in $P_k$ may be the last added with equal probability $1/k$.

- Let $p$ be this last added point.

- Then, either $p$ is interior to $H_k$ (no related processing) or $p$ is a vertex of $H_k$ with incident edges $e'$ and $e''$.

- Graph links re-arranged at the $k$-th stage (only) for points in the conflict lists of $e'$ and $e''$ . . .
Backward analysis

Because of the randomization, each of the $k$ points in $P_k$ may be the last added with equal probability $1/k$.

Let $p$ be this last added point.

Then, either $p$ is interior to $H_k$ (no related processing) or $p$ is a vertex of $H_k$ with incident edges $e'$ and $e''$.

Graph links re-arranged at the $k$-th stage (only) for points in the conflict lists of $e'$ and $e''$ . . .
Backward analysis

- ... the cost of re-arranging links at stage $k$ is $O(l' + l'')$, where $l'$, $l''$ are the sizes of the conflict lists of $e', e''$

- Then, the expected cost of the $k$-th stage is

$$
\sum_{p \in P_k} \frac{1}{k} O(l' + l'') = \frac{1}{k} \sum_{e \in H_k} 2 O(l) \leq \frac{2}{k} O(n)
$$

since the conflict lists contain $n - k \leq n$ points overall\(^1\)

- Notice that $2O(n)/k$ does not depend on the specific $P_k$, but the result would be the same for any $P_k \subset P$ of size $k$

\(^1\)actually $l' + l''$ may underestimate the costs at stage $k$, but $O(n)$ recovers anything lost

C. Mirolo  Convex Hull
Backward analysis

- The cost of re-arranging links at stage $k$ is $O(l' + l'')$, where $l'$, $l''$ are the sizes of the conflict lists of $e'$, $e''$.

Then, the expected cost of the $k$-th stage is

$$
\sum_{p \in P_k} \frac{1}{k} O(l' + l'') = \frac{1}{k} \sum_{e \in H_k} 2 O(l) \leq \frac{2}{k} O(n)
$$

since the conflict lists contain $n - k \leq n$ points overall.$^1$

- Notice that $2O(n)/k$ does not depend on the specific $P_k$, but the result would be the same for any $P_k \subset P$ of size $k$.

---

$^1$actually $l' + l''$ may underestimate the costs at stage $k$, but $O(n)$ recovers anything lost
Backward analysis

- The cost of re-arranging links at stage $k$ is $O(l' + l'')$, where $l'$, $l''$ are the sizes of the conflict lists of $e'$, $e''$

- Then, the expected cost of the $k$-th stage is

$$
\sum_{p \in P_k} \frac{1}{k} O(l' + l'') = \frac{1}{k} \sum_{e \in H_k} 2 O(l) \leq \frac{2}{k} O(n)
$$

since the conflict lists contain $n - k \leq n$ points overall

- Notice that $2O(n)/k$ does not depend on the specific $P_k$, but the result would be the same for any $P_k \subset P$ of size $k$

---

1 actually $l' + l''$ may underestimate the costs at stage $k$, but $O(n)$ recovers anything lost
Backward analysis

... the cost of re-arranging links at stage $k$ is $O(l' + l'')$, where $l'$, $l''$ are the sizes of the conflict lists of $e'$, $e''$

Then, the expected cost of the $k$-th stage is

$$\sum_{p \in P_k} \frac{1}{k} O(l' + l'') = \frac{1}{k} \sum_{e \in H_k} 2 O(l) \leq \frac{2}{k} O(n)$$

since the conflict lists contain $n - k \leq n$ points overall\(^1\)

Notice that $2O(n)/k$ does not depend on the specific $P_k$, but the result would be the same for any $P_k \subset P$ of size $k$

---

\(^1\) actually $l' + l''$ may underestimate the costs at stage $k$, but $O(n)$ recovers anything lost
Backward analysis

- $2O(n)/k$ is the expected cost of the $k$-th stage

- Then, the expected overall cost of re-arranging links is

$$\sum_{k=4}^{n} \frac{O(n)}{k} = O(n) \sum_{k=4}^{n} \frac{1}{k}$$

i.e. $O(n \log n)$

... which dominates the running time of the algorithm
Backward analysis

- $2O(n)/k$ is the expected cost of the $k$-th stage

- Then, the expected overall cost of re-arranging links is

$$\sum_{k=4}^{n} \frac{O(n)}{k} = O(n) \sum_{k=4}^{n} \frac{1}{k}$$

i.e. $O(n \log n)$

[ $1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{k}, \ldots$ harmonic series ]

... which dominates the running time of the algorithm
Backward analysis

- \(2O(n)/k\) is the expected cost of the \(k\)-th stage
- Then, the expected overall cost of re-arranging links is

\[
\sum_{k=4}^{n} \frac{O(n)}{k} = O(n) \sum_{k=4}^{n} \frac{1}{k}
\]

i.e. \(O(n \log n)\) \([\frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{k}, \ldots\]\) harmonic series

...which dominates the running time of the algorithm
Backward analysis

- $2\Theta(n)/k$ is the expected cost of the $k$-th stage

Then, the expected overall cost of re-arranging links is

$$\sum_{k=4}^{n} \frac{O(n)}{k} = O(n) \sum_{k=4}^{n} \frac{1}{k}$$

i.e. $O(n \log n)$ [ $1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{k}, \ldots$ harmonic series ]

...which dominates the running time of the algorithm
Outline

4 Semi-dynamic algorithms

5 Related results
   • Convex hull in 3D
   • Miscellaneous results

6 References
Static vs. semi-dynamic algorithms

- Conflict graph \(\rightarrow\) influence graph

- *Influence graph*: tree-like, incrementally updated structure

- *Static algorithm* \(\rightarrow\) *semi-dynamic algorithm*

- Same computational trend, for *random* input data, provided the cost of each graph update is \(O(\log n)\)
Static vs. semi-dynamic algorithms

- Conflict graph  $\rightarrow$ influence graph

- *Influence graph*: tree-like, incrementally updated structure

- *static algorithm*  $\rightarrow$ *semi-dynamic algorithm*

- Same computational trend, for *random* input data, provided the cost of each graph update is $O(\log n)$
Static vs. semi-dynamic algorithms

- Conflict graph $\rightarrow$ influence graph

- *Influence graph*: tree-like, incrementally updated structure

- *static* algorithm $\rightarrow$ *semi-dynamic* algorithm

- Same computational trend, for *random* input data, provided the cost of each graph update is $O(\log n)$
Static vs. semi-dynamic algorithms

- Conflict graph $\rightarrow$ influence graph

- *Influence graph*: tree-like, incrementally updated structure

- *static* algorithm $\rightarrow$ *semi-dynamic* algorithm

- Same computational trend, for *random* input data, provided the cost of each graph update is $O(\log n)$
Semi-dynamic algorithms

Related results

References

Devillers (1996)

Figure 3: The influence graph

2.2 The historical approach

In fact, the conflict graph can be replaced by another structure which, instead of storing the conflicts of non yet inserted objects, locates the regions in conflict with the new object. This approach yields semi-dynamic algorithms, objects are not known in advance but only when they are inserted. The basic idea of the influence graph [BDS+92] consists in remembering the history of the construction. When the insertion of a new object makes the conflicting regions disappear, they are not deleted but just marked inactive. The regions created are linked to existing regions in the influence graph in order to locate further conflicts. This idea of using the history appeared in computational geometry with the Delaunay tree [BT86, BT93] and was used in various other works for example [Tei93, Sei91, GKS92].

We now detail the case of sorting. The influence graph is in this case a binary tree whose nodes are intervals, the two sons of a node correspond to the splitting of that interval into two sub-intervals. When a new number \( x_k + 1 \) is inserted, it is located in the binary tree, the leaf containing it becomes an internal node, its interval \([x_i, x_j]\) is split into two new intervals. Thus for sorting, the influence graph is nothing else than an usual binary search tree (without balancing scheme). In fact, the comparisons done in the two algorithms are exactly the same. If \( x_i \) and \( x_j, i < j \) must be compared, they are compared during the insertion of \( x_i \) in the conflict graph and during the insertion of \( x_j \) in the influence graph. This likeness between the conflict and influence graphs is general, the conflict tests computed are the same, they are only delayed to achieve semi-dynamic algorithms.

2.3 Complexity

The algorithms above, as they are presented, are not randomized. They are incremental algorithms, updating a result (the set of regions without conflict) each time a new object is inserted. If a classical complexity analysis (in the worst case) is done, results are very bad, because the insertion of a new object may change a lot of things in the current result.

Now, we will randomize the algorithm, that is introduce some randomness,
Semi-dynamic algorithms

Related results
- Convex hull in 3D
- Miscellaneous results

References
No straightforward 3D generalization of Graham’s scan

- Divide-et-impera approach:
  Preparata & Hong (1977), $O(n \log n)$

- Randomized incremental approach:
  E.g. see survey in Devillers (1996), $O(n \log n)$
No straightforward 3D generalization of Graham’s scan

Divide-et-impera approach:
Preparata & Hong (1977), \( O(n \log n) \)

Randomized incremental approach:
E.g. see survey in Devillers (1996), \( O(n \log n) \)
Convex hull in 3D

- No straightforward 3D generalization of Graham’s scan

- Divide-et-impera approach:
  Preparata & Hong (1977), \( O( n \log n ) \)

- Randomized incremental approach:
  E.g. see survey in Devillers (1996), \( O( n \log n ) \)
Fig. 4. Merging two convex hulls. Construction of $\mathcal{J}$.
More about convex hull algorithms

- Jarvis’ march (2D) / gift wrapping (3D)
  - simple to code
  - running time: $O(nh)$
  - $h$ = hull vertices: may it be convenient?

- Optimal output sensitive algorithms
  - E.g., Chan (1996)
  - running time: $O(n \log h)$
  - more complex structure
More about convex hull algorithms

- Jarvis’ march (2D) / gift wrapping (3D)
  - simple to code
  - running time: $O(nh)$
  - $h =$ hull vertices: may it be convenient?

- Optimal output sensitive algorithms
  - E.g., Chan (1996)
  - running time: $O(n \log h)$
  - more complex structure
More about convex hull algorithms

- Jarvis’ march (2D) / gift wrapping (3D)
  - simple to code
  - running time: $O(nh)$
  - $h = \text{hull vertices}$: may it be convenient?

- Optimal output sensitive algorithms
  - E.g., Chan (1996)
  - running time: $O(n \log h)$
  - more complex structure
More about convex hull algorithms

- Jarvis’ march (2D) / gift wrapping (3D)
  - simple to code
  - running time: $O(nh)$
  - $h = $ hull vertices: may it be convenient?

- Optimal output sensitive algorithms
  - E.g., Chan (1996)
  - running time: $O(n \log h)$
  - more complex structure
Outline

4. Semi-dynamic algorithms

5. Related results
   - Convex hull in 3D
   - Miscellaneous results

6. References
“Convex hull is the favorite paradigm of computational geometers. Although the description of the problem is fairly simple, its solution takes into account all aspects of computational geometry.”

Olivier Devillers (1996)
References

F.P. Preparata & S.J. Hong (1977)
Convex hulls of finite sets of points in two and three dimensions
*Communications of the ACM, 157*

O. Devillers (1996)
An introduction to randomization in computational geometry
*Theoretical Computer Science, 20(2)*

T.M. Chan (1996)
Optimal output-sensitive convex hull algorithms in two and three dimensions
*Discrete & Computational Geometry, 16*