A Faster Convex-Hull Algorithm via Bucketing

Ask Neve Gamby[1] Jyrki Katajainen[2,3]

[1]National Space Institute
Technical University of Denmark

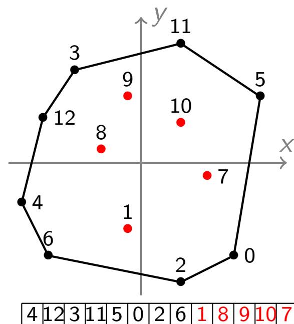
[2]Department of Computer Science
University of Copenhagen

[3]Jyrki Katajainen and Company

28th of June, 2019

Convex-hull problem in two dimensions

- Number of input points: n
- Number of output points: h
- The convex hull of a multiset S is the smallest polygon containing all the points in S
- Worst-case time: $\Omega(n + sort(h))$ and $O(n \lg(h))$ or $O(n \lg(n))$
- Average-case time: $O(n)$
- **Research question:** What is the fastest average-case algorithm in $\mathbb{Z}^2$?



Contributions

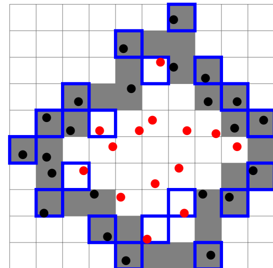
- We describe a space-efficient bucketing algorithm for the convex-hull problem
- We perform micro-benchmarks to show which operations are expensive and which are not
- We provide a few enhancements to most algorithms to speed up their straightforward implementations
- We perform experiments to investigate the state of the art of convex-hull algorithms
- We report the lessons learned while doing this study

Contributions

- We describe a space-efficient bucketing algorithm for the convex-hull problem
- We perform micro-benchmarks to show which operations are expensive and which are not
- We provide a few enhancements to most algorithms to speed up their straightforward implementations
- We perform experiments to investigate the state of the art of convex-hull algorithms
- We report the lessons learned while doing this study

Bucketing

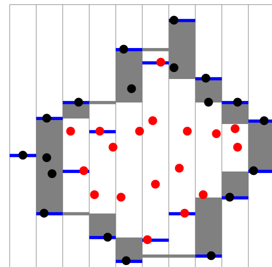
1. Determine a bounding rectangle of the n input points
2. Divide the bounding rectangle into a grid of size $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$, and distribute all the points into it
3. Mark all outer-layer cells that may contain extreme points
4. Collect the points in the marked cells and remove the rest
5. Use some other algorithm to compute the convex hull of the remaining points



Work space: $\Theta(n)$

Our improvement

1. Determine the west and east poles
2. Divide the x-axis into $\lceil \sqrt{n} \rceil$ slabs, and find the minimum and maximum y-value of each slab
3. Adjust the minimum and maximum y-values such that they form staircases
4. Collect the points outside the adjusted minimum and maximum y-values in their slab and remove the rest
5. Use some other algorithm to compute the convex hull of the remaining points



Work space: $\Theta(\sqrt{n})$

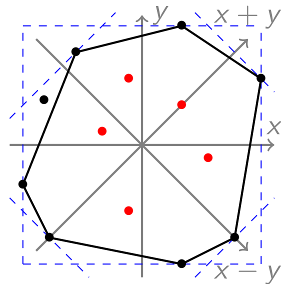
Throw-away elimination

Algorithm:

1. Find the extreme points in a few predetermined directions
2. Form a polygon of these extreme points
3. Remove all points inside the polygon
4. Use some other algorithm to compute the convex hull of the remaining points

Improvements:

- Used two left-turn tests per point
- Implemented in place
- Explored fast left-turn test, etc.

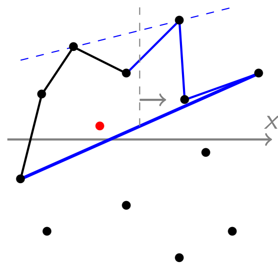


Work space: $\Theta(1)$

Plane sweep

Algorithm:

1. Find the west and east poles
2. Separate into two candidate collections
3. Sort the upper-hull candidates
4. Walk through the upper-hull candidates and eliminate left turns (and no turns)
5. Repeat 3–4 for the lower-hull candidates



Time: $O(n + sort(n))$
Work space: $O(\lg(n))$

Improvements:

- Implemented in situ
- Avoided in-place stable partitioning
- Explored fast left-turn test, sorting etc.

Experimental set-up

Input data:

- Coordinates of type **signed int** (32 bits)
- Each test is repeated $\left\lceil \frac{2^{27}}{n} \right\rceil$ times

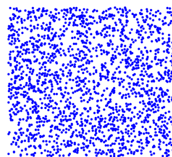
Platform:

- Hardware: 2.6 GHz CPU; 8 GB RAM
- Compiler: g++ 8.2.0
- Options: -O3 -std=c++2a -Wall -Wextra -fconcepts -DNDEBUG

Source code:

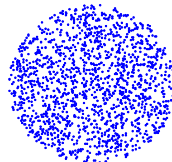
- `http://www.cphstl.dk/downloads.html`

Square data set:



$\mathbf{E}(h) = O(\lg n)$

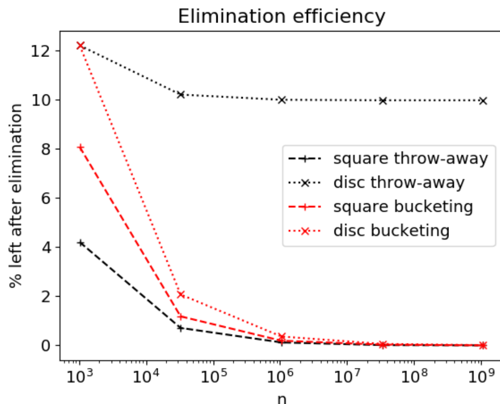
Disc data set:



$\mathbf{E}(h) = O(\sqrt[3]{n})$

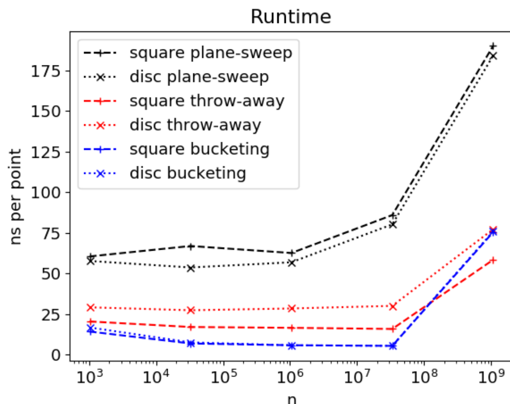
Elimination efficiency

- It is easy to eliminate most of the points for reasonably large *n*
- Throw-away struggles with curvature

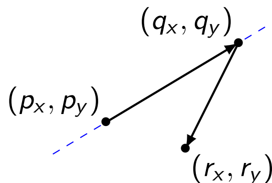


Execution time

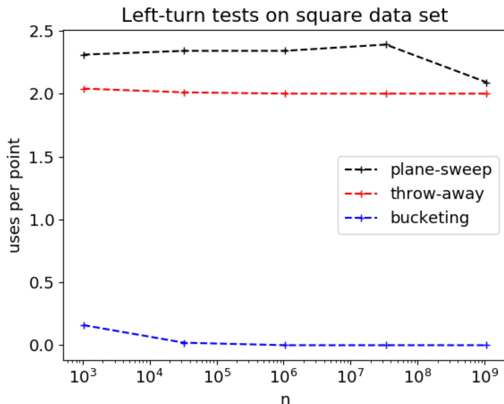
- Both preprocessing algorithms improve upon plane-sweep
- Bucketing is consistently the fastest



Left-turn test



- Integer calculations need slightly more than double precision: $\approx$ 15 ns per test
- Floating-point filter: $\approx$ 8 ns per test

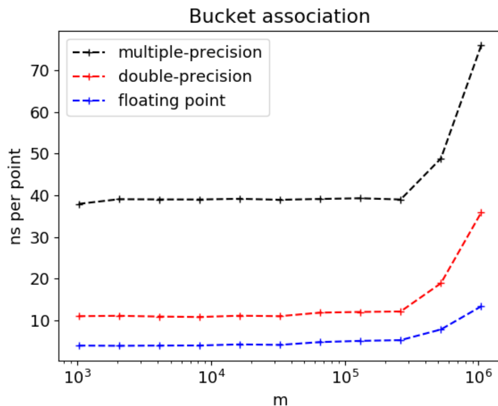


$$(q_x - p_x) \cdot (r_y - p_y) > (r_x - p_x) \cdot (q_y - p_y)$$

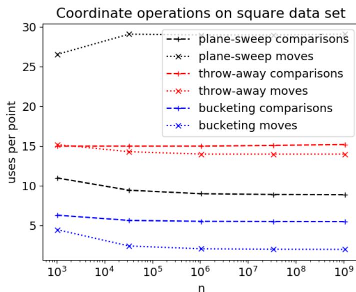
Bucket association

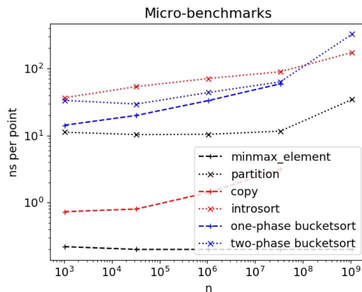
- Heavily sped up by floating-point numbers
- Floating-point numbers free us from specific ordering of multiplication and division
- $\frac{m-1}{x_{\max}-x_{\min}}$ precomputed

$$i = \left\lfloor \frac{(x - x_{\min}) \cdot (m-1)}{x_{\max} - x_{\min}} \right\rfloor$$



Other performance indicators





- Bucketing also does less coordinate comparisons and moves
- Based on micro-benchmarks, partitioning and sorting are the other expensive operations

Reflections

Compiler options: Make sure you use full optimization

Library facilities: They are usually good; no need to reinvent them

Techniques: Keep bucketing in your toolbox

Robustness: Implement your (geometric) primitives in a robust manner; it is easy with a multiple-precision library

Floating-point acceleration: Use floating-point numbers wisely; often they speed up things on modern processors

Space efficiency: Do not waste lots of space; $O(\sqrt{n})$ work space is acceptable

Correctness: Use an automated test framework and a verifier; they catch lots of bugs

Quality assurance: Try several alternatives for the same task to be sure about the quality of the chosen alternative