Beyond Theory

Geometric Algorithms

Lecture 6: Beyond Theory – Computing Delaunay Triangulations
Delaunay Triangulations

- Empty-circle property
Computing Delaunay Triangulations

- Since Lecture ϱ we know how to compute Delaunay Triangulations
- So in $O(n \log n)$ time we get ...
Computing Delaunay Triangulations

• Since Lecture γ we know how to compute Delaunay Triangulations

• So in $O(n \log n)$ time we get ... or ...
Why Engineering Algorithms?

Theory

In theory, theory and practice are the same.
Why Engineering Algorithms?

The real world out there...

In practice, theory and practice may be quite different...
Why Engineering Algorithms?

<table>
<thead>
<tr>
<th>Theory</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number types: IN, IR</td>
<td><strong>int, float, double</strong></td>
</tr>
<tr>
<td>Only asymptotics matter</td>
<td>Seconds do matter</td>
</tr>
<tr>
<td>Abstract algorithm description</td>
<td>Non-trivial implementation decisions, error-prone</td>
</tr>
<tr>
<td>Unbounded memory, unit access cost</td>
<td>Memory hierarchy / bandwidth</td>
</tr>
<tr>
<td>Elementary operations take constant time</td>
<td>Instruction pipelining, ...</td>
</tr>
</tbody>
</table>
Algorithm Engineering Cycle

- Algorithm design
- Theoretical analysis
- Algorithm implementation
- Experimental analysis

Deeper insights
Bottlenecks, Heuristics
More realistic models
Hints to refine analysis
Geometric Robustness

What went wrong?

Due to inexact arithmetic:
InCircle test reported a in circle of bcd
Geometric Predicates

Programs need to test relative positions of points based on their coordinates.

Examples:

**Orientation test** (in convex hull)
does c lie left/right/on the line ab?

**Incircle test** (in Delaunay triangulation)
does d lie in/out/on the circle abc?
Orientation(a, b, c) = sign of

\[
\begin{vmatrix}
  a_x & a_y & 1 \\
  b_x & b_y & 1 \\
  c_x & c_y & 1 \\
\end{vmatrix} = \begin{vmatrix}
  a_x - c_x & a_y - c_y \\
  b_x - c_x & b_y - c_y \\
\end{vmatrix}
\]

We only consider the **sign** of an expression involving +, -, and *

**Differences** reduce error, but we still need to deal with it.
InCircle Test = Orientation Test after lifting points \((a_x, a_y) \rightarrow (a_x, a_y, a_x^2 + a_y^2)\)
InCircle Test

InCircle(a, b, c, d) = sign of

\[
\begin{vmatrix}
 a_x & a_y & a_x^2 + a_y^2 & 1 \\
 b_x & b_y & b_x^2 + b_y^2 & 1 \\
 c_x & c_y & c_x^2 + c_y^2 & 1 \\
 d_x & d_y & d_x^2 + d_y^2 & 1 \\
\end{vmatrix}
= \begin{vmatrix}
 a_x - d_x & a_y - d_y & (a_x - d_x)^2 + (a_y - d_y)^2 \\
 b_x - d_x & b_y - d_y & (b_x - d_x)^2 + (b_y - d_y)^2 \\
 c_x - d_x & c_y - d_y & (c_x - d_x)^2 + (c_y - d_y)^2 \\
\end{vmatrix}
\]

We only consider the **sign** of an expression involving +, - and *
Floating-point Filters

Get **correct sign** (-1, 0 or 1) of an exact expression $E$ using floating-point!

```
let F = E (X) in floating point
if F > error bound then 1 else
  if –F > error bound then –1 else
    increase precision and repeat
  or switch to exact arithmetic
```

“filters out” the easy cases

If the correct result is 0, must go to exact phase
Robustness Wrap-Up

- trade-off: exact arithmetic vs speed
- often sufficient: answer predicates exactly, but construction may be inexact

- robustness is difficult to achieve
- good news: robust implementations exist
<table>
<thead>
<tr>
<th>Program</th>
<th>F</th>
<th>point location</th>
<th>E</th>
<th>C</th>
<th>degeneracy</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGAL</td>
<td>N</td>
<td>Delaunay hierarchy</td>
<td>Y</td>
<td>N</td>
<td>Perturbing $E^3$</td>
<td>C++</td>
</tr>
<tr>
<td>Hull</td>
<td>N</td>
<td>history dag</td>
<td>Y</td>
<td>Y</td>
<td>Perturb points into hull in $E^4$</td>
<td>C</td>
</tr>
<tr>
<td>Pyramid</td>
<td>Y</td>
<td>jump-and-walk</td>
<td>Y</td>
<td>N</td>
<td>Perturb points into hull in $E^4$. Remove flat tetrahedra by post-processing</td>
<td>C</td>
</tr>
<tr>
<td>QHull</td>
<td>N</td>
<td>outside set</td>
<td>N</td>
<td>N</td>
<td>Perturb points into hull in $E^4$. Remove flat tetrahedra by post-processing</td>
<td>C</td>
</tr>
<tr>
<td>Tess3</td>
<td>N</td>
<td>Hilbert ordering, zig-zag walk</td>
<td>N</td>
<td>Y</td>
<td>Perturbation in $E^4$ with no flat tetrahedra.</td>
<td>C</td>
</tr>
</tbody>
</table>

**Table 1.** Program comparison summary. Column abbreviations: F = uses flips; E = exact; C = uses caching spheres; L = programming language. Versions and dates: CGAL version 2.4; hull and pyramid obtained in March 2004; qhull version 2003.1; Tess3 last revised in 9/2003.
Let’s suppose we have a robust randomized incremental construction of Delaunay triangulations …
Standard RIC in Experiments

- What went wrong?
- Thrashing due to random memory access
Partial Randomization

- Random access to large data: bad idea
- Don’t randomize? Really bad in theory and also causes overhead in experiments

- **Partially** randomized insertion order
  - increase locality of reference, especially as data structure gets large
  - retain enough randomness to guarantee optimality
Partial Randomization

- Choose each point with prob = ½.
- Insert chosen points recursively
- Insert the remaining points in order of choice (local, not random!).
Partial Randomization

- Choose each point with prob = $\frac{1}{2}$.
- Insert chosen points recursively.
- Insert the remaining points in order of choice (local, not random!).
Partial Randomization

**Theorem:** The Delaunay triangulation of $n$ points in the plane can be constructed in $O(n \log n)$ expected time using partial randomization.

**Proof on blackboard**

- We know: full randomization is asymptotically optimal
- Partial randomization: probability for a triangle to occur at most a constant factor larger than for full randomization
Interim evaluation

Algorithm design

Theoretical analysis

Algorithm implementation

Experimental analysis

Deeper insights

Bottlenecks, Heuristics

More realistic models

Hints to refine analysis

Geometric Algorithms Lecture 6: Beyond Theory
Interim evaluation

– Implementations, experiments and theory go well together
– robust implementations are challenging
– strong experimental analysis is crucial. Guidelines?!
David Johnson’s guidelines

[A Theoretician’s Guide to the Experimental Analysis of Algorithms]

• 3 types of paper describe the implementation of an algorithm
  – application paper
    “Here’s a good algorithm for this problem”
  – Horse race paper
    “Here’s a fast algorithm (compared to previous)”
  – experimental paper
    “Here’s how this algorithm behaves in practice”

• These lessons apply to all 3
David Johnson’s guidelines

- Perform “newsworthy” experiments
  - standards higher than for theoretical papers!
  - run experiments on real problems
    - theoreticians can get away with idealized distributions
      but experimentalists have no excuse!
  - don’t use algorithms that theory can already dismiss
  - look for generality and relevance
    - don’t just report algorithm A dominates algorithm B,
      identify why it does!
David Johnson’s guidelines

• Place work in context
  – compare against previous work in literature
  – ideally, obtain their code and test sets
    verify their results, and compare with your new algorithm
  – less ideally, re-implement their code
    report any differences in performance
  – least ideally, simply report their old results
    try to make some ball-park comparisons of machine speeds
David Johnson’s guidelines

- Use reasonably efficient implementations
  - “somewhat” controversial
  - efficient implementation supports claims of practicality
    - tells us what is achievable in practice
  - can run more experiments on larger instances
    - can do our research quicker!
  - don’t have to go over-board on this
  - exceptions can also be made
    - e.g. not studying CPU time, comparing against a previously newsworthy algorithm, programming time more valuable than processing time, ...
David Johnson’s guidelines

• Use testbeds that support general conclusions
  – ideally one (or more) random class, & real world instances
    \textit{predict performance on real world problems based on random class, evaluate quality of predictions}
  – structured random generators
    \textit{parameters to control structure as well as size}
  – don’t just study real world instances
    \textit{hard to justify generality unless you have a very broad class of real world problems!}
David Johnson’s guidelines

• Provide explanations and back them up with experiment
  – adds to credibility of experimental results
  – improves our understanding of algorithms
    leading to better theory and algorithms
  – can “weed” out bugs in your implementation!
David Johnson’s guidelines

• Ensure reproducibility
  – easily achieved via the Web
  – adds support to a paper if others can (and do) reproduce the results
  – requires you to use large samples and wide range of problems

  otherwise results will not be reproducible!
David Johnson’s guidelines

• Ensure comparability (and give the full picture)
  – make it easy for those who come after to reproduce your results
  – provide meaningful summaries
    
    *give sample sizes, report standard deviations, plot graphs but report data in tables in the appendix*

  – do not hide anomalous results
  – report running times even if this is not the main focus
    
    *readers may want to know before studying your results in detail*
David Johnson’s pitfalls

- Failing to report key implementation details
- Extrapolating from tiny samples
- Using irreproducible benchmarks
- Using running time as a stopping criterion
- Ignoring hidden costs (e.g. preprocessing)
- Misusing statistical tools
- Failing to use graphs
David Johnson’s pitfalls

• Obscuring raw data by using hard-to-read charts
• Comparing apples and oranges
• Drawing conclusions not supported by the data
• Leaving obvious anomalies unnoted/unexplained
• Failing to back up explanations with further experiments
• Ignoring the literature

*the self-referential study!*