The Role of Science and Mathematics in Software Development

Robert Sedgewick
Princeton University

The scientific method is essential in applications of computation

A personal opinion formed on the basis of decades of experience as a

- CS educator
- author
- algorithm designer
- software engineer
- Silicon Valley contributor
- CS researcher

Personal opinion . . . or unspoken consensus?

Unfortunate facts

Many scientists lack basic knowledge of computer science
Many computer scientists lack back knowledge of science

1970s: Want to use the computer? Take intro CS.

2000s: Intro CS course relevant only to future cubicle-dwellers

One way to address the situation

- identify fundamentals
- teach them to all students who need to know them
- as early as possible

One way to address the situation

Teach the same course to all science/engineering students

All students learn the importance of
- modern programming models
- the scientific method in understanding program behavior
- fundamental precepts of computer science
- computation in a broad variety of applications
- preparing for a lifetime of engaging with computation
Science/engineering students at Princeton take the same intro CS course, most in the first year

**modern programming model**
- Basic control structures
- Standard input and output streams
- Drawings, images and sound
- Data abstraction
- Use any computer, and the web

**relevant CS concepts**
- Applications programming
- Understanding of the costs
- Fundamental data types
- Computer architecture
- Computability and Intractability

**Goals**
- demystify computer systems
- empower students to exploit computation
- build awareness of intellectual underpinnings of CS

**Underlying message: performance matters**

in a large number of interesting applications

Simple fact: quadratic algorithms are useless in modern applications
- millions or billions of inputs
- $10^{12}$ nanoseconds is 15+ minutes
- $10^{18}$ nanoseconds is 31+ years

Simple test: Doubling hypothesis
- Perform experiments, measure $T(N)$ and $T(2N)$
- if $T(2N)/T(N) \sim 4$, need another algorithm

**Lessons:**
1. Efficient algorithms enable solution of problems that could not otherwise be addressed.
2. Scientific method is essential in understanding program performance

**Examples and assignments**

use familiar easy-to-motivate applications

**Ideal programming example/assignment**
- teaches a basic CS concept
- solves an important problem
- appeals to students' intellectual interest
- illustrates modular programming

**The scientific method**

is essential in understanding program performance

**Scientific method**
- create a model describing natural world
- use model to develop hypotheses
- run experiments to validate hypotheses
- refine model and repeat

1950s: uses scientific method
- **Algorithm designer** who does not experiment gets lost in abstraction

2000s: uses scientific method
- **Software developer** who ignores cost risks catastrophic consequences
Modern software development requires huge amounts of code

but

performance-critical code implements relatively few fundamental algorithms

Warmup: random number generation

Problem: write a program to generate random numbers

model: classical probability and statistics

hypothesis: frequency values should be uniform

weak experiment:

• generate random numbers
• check for uniform frequencies

better experiment:

• generate random numbers
• use $\chi^2$ test to check frequency values against uniform distribution

better hypotheses/experiments still needed

• many documented disasters
• active area of scientific research
• applications: simulation, cryptography
• connects to core issues in theory of computation

Q. Is a given sequence of numbers random?
A. No.

Q. Does a given sequence exhibit some property that random sequence exhibits?

Birthday paradox
Average count of random numbers generated until a duplicate happens is about $\sqrt{n}/2$.

Example of a better experiment:

• generate numbers until duplicate
• check that count is close to $\sqrt{n}/2$
• even better: repeat many times, check against distribution
• still better: run many similar tests for other properties

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin” — John von Neumann
**Finding Paths in Graphs**

Robert Sedgewick
Princeton University

**Motivating example: maxflow**

Ford-Fulkerson maxflow scheme
- find any s-t path in a (residual) graph
- augment flow along path (may create or delete edges)
- iterate until no path exists

Goal: compare performance of two basic implementations
- shortest augmenting path
- maximum capacity augmenting path

Key steps in analysis
- How many augmenting paths?
- What is the cost of finding each path?

**Finding an s-t path in a graph**

is a fundamental operation that demands understanding

**Ground rules for this talk**
- work in progress (more questions than answers)
- basic research
- save “deep dive” for the right problem

**Applications**
- graph-based optimization models
- networks
- percolation
- computer vision
- social networks
- (many more)

**Basic research**
- fundamental abstract operation with numerous applications
- worth doing even if no immediate application
- resist temptation to prematurely study impact

**Motivating example: max flow**

Compare performance of Ford-Fulkerson implementations
- shortest augmenting path
- maximum-capacity augmenting path

**Graph parameters**
- number of vertices $V$
- number of edges $E$
- maximum capacity $C$

**How many augmenting paths?**

<table>
<thead>
<tr>
<th>shortest</th>
<th>worst case upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE/2</td>
<td>VC</td>
</tr>
</tbody>
</table>

**How many steps to find each path?** $E$ (worst-case upper bound)
Motivating example: max flow

Compare performance of Ford-Fulkerson implementations
- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters for example graph
- number of vertices $V = 177$
- number of edges $E = 2000$
- maximum capacity $C = 100$

How many augmenting paths?

<table>
<thead>
<tr>
<th>worst case upper bound</th>
<th>for example</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortest</td>
<td>VE/2</td>
</tr>
<tr>
<td></td>
<td>VC</td>
</tr>
<tr>
<td>max capacity</td>
<td>2E lg C</td>
</tr>
<tr>
<td></td>
<td>26,575</td>
</tr>
</tbody>
</table>

How many steps to find each? 2000 (worst-case upper bound)

Motivating example: max flow

Compare performance of Ford-Fulkerson implementations
- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters for example graph
- number of vertices $V = 177$
- number of edges $E = 2000$
- maximum capacity $C = 100$

How many augmenting paths?

<table>
<thead>
<tr>
<th>worst case upper bound</th>
<th>for example</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortest</td>
<td>VE/2</td>
<td>177,000</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>17,700</td>
</tr>
<tr>
<td>max capacity</td>
<td>2E lg C</td>
<td>26,575</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How many steps to find each? < 20, on average

Motivating example: lessons

Goals of algorithm analysis
- predict performance (running time)
- guarantee that cost is below specified bounds

Common wisdom
- random graph models are unrealistic
- average-case analysis of algorithms is too difficult
- worst-case performance bounds are the standard

Unfortunate truth about worst-case bounds
- often useless for prediction (fictional)
- often useless for guarantee (too high)
- often misused to compare algorithms

Bounds are useful in some applications:

Open problem: Do better!
**Surely, we can do better**

An actual exchange with a theoretical computer scientist:

**TCS (in a talk):** Algorithm A is bad. Google should be interested in my new Algorithm B.

**RS:** What’s the matter with Algorithm A?

**TCS:** It is not optimal. It has an extra $O(\log \log N)$ factor.

**RS:** But Algorithm B is very complicated, $\lg \lg N$ is less than 6 in this universe, and that is just an upper bound. Algorithm A is certainly going to run 10 to 100 times faster in any conceivable real-world situation. Why should Google care about Algorithm B?

**TCS:** Well, I like it. I don’t care about Google.

---

**Finding an st-path in a graph**

is a basic operation in a great many applications

**Q.** What is the best way to find an st-path in a graph?

**A.** Several well-studied textbook algorithms are known

- Breadth-first search (BFS) finds the shortest path
- Depth-first search (DFS) is easy to implement
- Union-Find (UF) needs two passes

**BUT**

- all three process all $E$ edges in the worst case
- diverse kinds of graphs are encountered in practice

Worst-case analysis is useless for predicting performance

**Which basic algorithm should a practitioner use?**

---

**Grid graphs**

Algorithm performance depends on the graph model

- complete
- random
- grid
- neighbor
- small-world

Initial choice: grid graphs

- sufficiently challenging to be interesting
- found in practice (or similar to graphs found in practice)
- scalable
- potential for analysis

Ground rules

- algorithms should work for all graphs
- algorithms should not use any special properties of the model

---

**Applications of grid graphs**

- conductivity
- concrete
- granular materials
- porous media
- polymers
- forest fires
- epidemics
- Internet
- resistor networks
- evolution
- social influence
- Fermi paradox
- fractal geometry
- stereo vision
- image restoration
- object segmentation
- scene reconstruction

**Example 1: Percolation**

- widely-studied model
- few answers from analysis
- arbitrarily huge graphs

**Example 2: Image processing**

- model pixels in images
- DFS, maxflow/mincut, and other alg
- huge graphs
Finding an s-t-path in a grid graph

**M by M grid** of vertices
undirected edges connecting each vertex to its HV neighbors
source vertex s at center of top boundary
destination vertex t at center of bottom boundary

Find any path connecting s to t

Cost measure: number of graph edges examined

Data abstraction

A modern tool to separate clients from implementations

A **data type** is a set of values and the operations performed on them

An abstract data type (ADT) is a data type whose representation is hidden

An applications programming interface (API) is a specification

Implementation should not be tailored to particular client

Implementations that work properly for all clients
Study their performance for the client at hand

Implementing a GRAPH data type

is an exercise in **software engineering**

Sample “design pattern” (for this talk)

**GRAPH API**

<table>
<thead>
<tr>
<th>METHOD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAPH()</td>
<td>create a graph</td>
</tr>
<tr>
<td>void findPath(int s, int t)</td>
<td>conduct a search from s to t</td>
</tr>
<tr>
<td>int predecessor(int v)</td>
<td>return predecessor of v on path found</td>
</tr>
</tbody>
</table>

Client code for grid graphs

```java
int e = 0;
for (int i = 0; i < V; i++)
  if (i < M) a[i][i] = new Edge(i, i+1);
  if (i+1 < V) a[i][i+1] = new Edge(i, i+1);
  if (i < M) a[i][i+M] = new Edge(i, i+1);
  if (i+1 < V) a[i][i+M+1] = new Edge(i, i+1);
```

**GRAPH** G = new **GRAPH**();
G.FindPath(V-M/2, M/2);
for (int k = t; k != s; k = G.predecessor(k))
  System.out.print(k + " -> ");
```
Three standard ways to find a path

Depth-first search (DFS): recursive (stack-based) search
Breadth-first search (BFS): queue-based shortest-path search
Union-find (UF): use classic set-equivalence algorithms

First step: Implement GRAPH using each algorithm

Depth-first search: a standard implementation

GRAPH constructor code

```java
for (int k = 0; k < E; k++)
{ int v = a[k].v, w = a[k].w;
  adj[v] = new Node(w, adj[v]);
  adj[w] = new Node(v, adj[w]);
}
```

dfs implementation (code to save path omitted)

```java
void findPath(int s, int t)
{ if (s == t) return;
  visited[s] = true;
  for (Node x = adj[s]; x != null; x = x.next)
    if (!visited[x.v]) findPath(x.v, t);
}
```

Addressing the basic flaw

Advise the client to randomize the edges?
- no, very poor software engineering
- leads to nonrandom edge lists

Randomize each edge list before use?
- no, may not need the whole list

Solution: Use a randomized iterator

Standard iterator

```java
int N = adj[x].length;
for(int i = 0; i < N; i++)
{ process vertex adj[x][i];
}
```

Randomized iterator

```java
int N = adj[x].length;
for(int i = 0; i < N; i++)
{ int r = (int) Math.random() * (N-1);
  exchange random vertex from adj[x][r] with adj[x][i];
  process vertex adj[x][i];
}
```
Use of randomized iterators
turns every graph algorithm into a randomized algorithm

Important practical effect: stabilizes algorithm performance

cost depends on problem not its representation

Yields well-defined and fundamental analytic problems
- Average-case analysis of algorithm X for graph family Y(N)?
- Distributions?
- Full employment for algorithm analysts

(Revised) standard DFS implementation

```
graph ADT constructor code
for (int k = 0; k < E; k++)
{
    int v = a[k].v, w = a[k].w;
    adj[v][deg[v]+1] = w;
    adj[w][deg[w]+1] = v;
}
```

DFS implementation (code to save path omitted)

```
void findPathR(int s, int t)
{
    int N = adj[s].length;
    if (s == t) return;
    visited[s] = true;
    for(int i = 0; i < N; i++)
    {
        int v = exch(adj[s], i, i+(int) Math.random()*(N-i));
        if (!visited[v]) searchR(v, t);
    }
}
```

BFS: standard implementation

Use a queue to hold fringe vertices

```
void findPath(int s, int t)
{
    Queue Q = new Queue();
    Q.put(s); visited[s] = true;
    while (!Q.empty())
    {
        int x = Q.get(); int N = adj[x].length;
        if (x == t) return;
        for (int i = 0; i < N; i++)
        {
            int v = exch(adj[x], i, i+(int) Math.random()*(N-i));
            if (!visited[v])
            {
                Q.put(v); visited[v] = true;
            }
        }
    }
}
```

Animations
give intuition on performance and suggest hypotheses to verify with experimentation

Aside: Are you using animations like this regularly? Why not?
Experimental results

show that DFS is faster than BFS and UF \textit{on the average}

<table>
<thead>
<tr>
<th>M</th>
<th>V</th>
<th>E</th>
<th>BFS</th>
<th>DFS</th>
<th>UF</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>49</td>
<td>168</td>
<td>0.75</td>
<td>0.51</td>
<td>1.05</td>
</tr>
<tr>
<td>15</td>
<td>225</td>
<td>840</td>
<td>0.75</td>
<td>0.45</td>
<td>1.02</td>
</tr>
<tr>
<td>31</td>
<td>961</td>
<td>3720</td>
<td>0.75</td>
<td>0.36</td>
<td>1.14</td>
</tr>
<tr>
<td>63</td>
<td>3969</td>
<td>15624</td>
<td>0.75</td>
<td>0.32</td>
<td>1.05</td>
</tr>
<tr>
<td>127</td>
<td>16129</td>
<td>64008</td>
<td>0.75</td>
<td>0.40</td>
<td>0.98</td>
</tr>
<tr>
<td>255</td>
<td>65025</td>
<td>259080</td>
<td>0.75</td>
<td>0.42</td>
<td>1.08</td>
</tr>
</tbody>
</table>

**Analytic proof?**

**Faster algorithms available?**

A faster algorithm

for finding an \textit{st}-path in a graph

Use two depth-first searches
- one from the source
- one from the destination
- interleave the two

Examines 13\% of the edges
3·8 times faster than standard implementations

Not bad (but still apparently linear)

Are other approaches faster?

Other search algorithms
- randomized?
- farthest-first?

Multiple searches?
- interleaving strategy?
- merge strategy?
- how many?
- which algorithm?

Hybrid algorithms
- which combination?
- probabilistic restart?
- merge strategy?
- randomized choice?

Better than constant-factor improvement possible? Proof?

Experiments with other approaches

Randomized search
- use random queue in BFS
- easy to implement
Result: not much different from BFS

Multiple searchers
- use N searchers
- one from the source
- one from the destination
- \(N-2\) from random vertices
- Additional factor of 2 for \(N>2\)
Result: not much help anyway

Best method found (by far): DFS with 2 searchers
Small-world graphs

are a widely studied graph model with many applications

A small-world graph has
- large number of vertices
- low average vertex degree (sparse)
- low average path length
- local clustering

Examples:
- Add random edges to grid graph
- Add random edges to any sparse graph with local clustering
- Many scientific models

Q. How do we find an st-path in a small-world graph?

Finding a path in a small-world graph

is a heavily studied problem

Milgram experiment (1960)

Small-world graph models
- Random (many variants)
- Watts-Strogatz
- Kleinberg

How does 2-way DFS do in this model?

Experiment:
- add M ~ E^{1/2} random edges to an M-by-M grid graph
- use 2-way DFS to find path

Surprising result: Finds short paths in ~ E^{1/2} steps!

Finding a path in a small-world graph

is much easier than finding a path in a grid graph

Conjecture: Two-way DFS finds a short st-path in sublinear time in any small-world graph

Evidence in favor
1. Experiments on many graphs
2. Proof sketch for grid graphs with V shortcuts
   - step 1: 2 E^{1/2} steps ~ 2 V^{1/2} random vertices
   - step 2: like birthday paradox

Path length?
Multiple searchers revisited?

Next steps: refine model, more experiments, detailed proofs

Applications of small-world graphs

- social networks
- airlines
- roads
- neurobiology
- evolution
- social influence
- protein interaction
- percolation
- internet
- electric power grids
- political trends

Example 1: Social networks
- infectious diseases
- extensive simulations
- some analytic results
- huge graphs

Example 2: Protein interaction
- small-world model
- natural process
- experimental validation
Introduction

Motivating example

Grid graphs

Search methods

Small world graphs

Analytic combinatorics

Detailed example: paths in graphs

End of "lecture-within-a-lecture"

More questions than answers

Answers:
- Randomization makes cost depend on graph, not representation.
- DFS is slower than BFS or TF for finding paths in grid graphs.
- Two DFSs are faster than 1 DFS — or 9 of them — in grid graphs.
- We can find shortest paths quickly in small-world graphs.

Questions:
- What are the BFS, DFS, and UCS constraints in grid graphs?
- Is there a sublinear algorithm for grid graphs?
- Which methods adapt to directed graphs?
- Can we predict parallel and sequential costs for grid graphs?
- What is the cost distribution for DFSs for interesting graph models?
- Are effective any these methods for other graph families?
- Do these methods lead to faster-matrix algorithms?
- Are effective these methods in practice?

Lessons

• We know much less about graph algorithms than you might think
• The scientific method is essential in understanding performance

Appropriate mathematical models

are essential for scientific studies of program behavior

Pioneering work by Don Knuth

Large and active "analysis of algorithms" research community is actively studying models and methods.

Caution: Not all mathematical models are appropriate!

Example (from beginning of talk): O-notation in the theory of algorithms

• hides details of implementation
• takes input out by doing worst-case
• useful for classifying algorithms and complexity classes
• not at all useful for predicting or comparing performance

Concluding remarks

on the role of mathematics in understanding performance

Worrisome point

• Complicated mathematics seems to be needed for models
• Do all programmers need to know the math?

Good news

• Many people are working on the problem
• Simple universal underlying models are emerging

Analytic Combinatorics

is a modern basis for studying discrete structures

Developed by

Philippe Flajolet and many coauthors

based on

classical combinatorics and analysis

Generating functions (GFs) encapsulate sequences

Symbolic methods treat GFs as formal objects

• formal definition of combinatorial constructions
• direct association with generating functions

Complex asymptotics treat GFs as functions in the complex plane

• Study them with singularity analysis and other techniques
• Accurately approximate original sequence

Analytic Combinatorics

Philippe Flajolet Bobet Sedgewick

Cambridge University Press

Coming in 2008, now available on the web
Analysis of algorithms: classic example

A binary tree is a node connected to two binary trees.

How many binary trees with $N$ nodes?

Given a recurrence relation
introduce a generating function
multiply both sides by $z^N$ and sum to get an equation
that we can solve algebraically
and expand to get coefficients
that we can approximate

Basic challenge: need a new derivation for each problem

\[
B_n = B_0 B_{n-1} + \ldots + B_4 B_{n-4} + \ldots + B_{k-1} B_k
\]

\[
B(z) = B_0 z^0 + B_1 z^1 + B_2 z^2 + B_3 z^3 + \ldots
\]

\[
B(z) = \frac{1 - \sqrt{1 - 4z}}{2z}
\]

\[
B_n = \frac{1}{N+1} \binom{2N}{N}
\]

\[
B_n \approx \frac{4^n}{N! \pi N}
\]

Analytic combinatorics: classic example

A tree is a node connected to a sequence of trees

How many trees with $N$ nodes?

Combinatorial constructions

\[
G(z) = 1 + G(z) + G(z)^2 + G(z)^3 + \ldots
\]

directly map to CFs

\[
G(z) = \frac{1 - \sqrt{1 - 4z}}{2z}
\]

by quadratic equation

\[
G(z) = \frac{1}{1 - G(z)}
\]

so $G(z)^2 - G(z) + z = 0$

and treat as a complex function to approximate growth

First principle: location of singularity determines exponential growth

Second principle: nature of singularity determines subexponential factor

Analytic combinatorics: universal laws

of sweeping generality derive from the same technology

Ex. Context free constructions

\[
< G_0 > = 0P( < G_0 >, < G_1 >, \ldots, < G_n > )
\]

\[
< G_1 > = 0P( < G_0 >, < G_1 >, \ldots, < G_n > )
\]

\[
< G_i > = 0P( < G_0 >, < G_1 >, \ldots, < G_n > )
\]

Combinatorial constructions

\[
G_0(z) = F_0( G_0(z), G_0(z), \ldots, G_0(z) )
\]

\[
G_1(z) = F_1( G_0(z), G_0(z), \ldots, G_0(z) )
\]

\[
G_i(z) = F_i( G_0(z), G_0(z), \ldots, G_0(z) )
\]

Grothendieck-basis elimination

Drmota-Lalley-Woods

that is amenable to singularity analysis

\[
C_n \sim b^N N^c
\]

for any context-free construction!

Good news: Several such laws have been discovered

Better news: Distributions also available (typically normal, small sigma)


**A general hypothesis from analytic combinatorics**

The running time of your program is \( \sim a \ b^c \ (\lg N)^d \)

- the constant \( a \) depends on both complex functions and properties of machine and implementation
- the exponential growth factor \( b \) should be 1
- the exponent \( c \) depends on singularities
- the log factor \( d \) is reconciled in detailed studies

Why?

- data structures evolve from combinatorial constructions
- universal laws from analytic combinatorics have this form

To compute values:

- \( \lg(T(2N))/T(N) \to c \) \( \text{the doubling test that we teach to beginners} \)
- \( T(N)/b^c \ N^{c} \to a \)

Plenty of caveats, but provides a basis for studying program performance

---

**Final remarks**

Writing a program without understanding performance is like

not knowing where a rocket will go

We need to

- teach the scientific method throughout the curriculum
- use the scientific method whenever developing software
- do the research necessary to develop underlying models

---

**The Role of Science and Mathematics in Software Development**

Robert Sedgewick
Princeton University