The Role of the Scientific Method in Software Development

Robert Sedgewick
Princeton University
The scientific method is necessary in algorithm design and software development.

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

Algorithm designer who does not experiment gets lost in abstraction

Software developer who ignores cost risks catastrophic consequences
First hypothesis (needs checking)

Modern software development requires huge amounts of code
First hypothesis (needs checking)

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
Warmup (continued)

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

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

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

Example of a better experiment:
• generate numbers until duplicate
• check that count is close to $\sqrt{\pi V/2}$
• 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
Detailed example: paths in graphs

A lecture within a lecture

Finding Paths in Graphs

Robert Sedgewick
Princeton University
Finding an \textit{st}-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: 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?

research literature
this talk
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>VE/2</th>
<th>VC</th>
</tr>
</thead>
<tbody>
<tr>
<td>max capacity</td>
<td>2E lg C</td>
<td></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 VC</td>
</tr>
<tr>
<td>max capacity</td>
<td>$2E \log C$</td>
</tr>
</tbody>
</table>

How many steps to find each path? 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>Shortest</th>
<th>For example</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VE/2$</td>
<td>177,000</td>
<td>37</td>
</tr>
<tr>
<td>$VC$</td>
<td>17,700</td>
<td></td>
</tr>
<tr>
<td>Max capacity</td>
<td>2E lg C</td>
<td>26,575</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

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

Total number of steps?

<table>
<thead>
<tr>
<th>worst case upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortest</td>
</tr>
<tr>
<td>VE^2/2</td>
</tr>
<tr>
<td>VEC</td>
</tr>
<tr>
<td>max capacity</td>
</tr>
<tr>
<td>2E^2 lg C</td>
</tr>
</tbody>
</table>

WARNING: The Algorithm General has determined that using such results to predict performance or to compare algorithms may be hazardous.
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 many applications:

Open problem: Do better!
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

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

if vertices have positions we can find short paths quickly with A* (stay tuned)
Applications of grid graphs

Example 1: Statistical physics
- percolation model
- extensive simulations
- some analytic results
- arbitrarily huge graphs

Example 2: Image processing
- model pixels in images
- maxflow/mincut
- energy minimization
- huge graphs
Finding an $st$-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 $t$

<table>
<thead>
<tr>
<th>$M$</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>49</td>
<td>84</td>
</tr>
<tr>
<td>15</td>
<td>225</td>
<td>420</td>
</tr>
<tr>
<td>31</td>
<td>961</td>
<td>1860</td>
</tr>
<tr>
<td>63</td>
<td>3969</td>
<td>7812</td>
</tr>
<tr>
<td>127</td>
<td>16129</td>
<td>32004</td>
</tr>
<tr>
<td>255</td>
<td>65025</td>
<td>129540</td>
</tr>
<tr>
<td>511</td>
<td>261121</td>
<td>521220</td>
</tr>
</tbody>
</table>

Cost measure: number of graph edges examined
Finding an $st$-path in a grid graph

Similar problems are covered extensively in the literature

- Percolation
- Random walk
- Nonselfintersecting paths in grids
- Graph covering

Which basic algorithm should a practitioner use to find a path in a grid-like graph?
Finding an $st$-path in a grid graph

Elementary algorithms are found in textbooks

• Depth-first search (DFS)
  
• Breadth-first search (BFS)
  
• Union-find

Which basic algorithm should a practitioner use to find a path in a grid-like graph?
Abstract data types

separate clients from implementations

A data type is a set of values and the operations performed on them. An abstract data type is a data type whose representation is hidden.

Implementation should not be tailored to particular client.

Develop implementations that work properly for all clients. Study their performance for the client at hand.
Graph abstract data type

**Vertices** are integers between 0 and V–1

**Edges** are vertex pairs

**Graph ADT** implements

- `Graph(Edge[])` to construct graph from array of edges
- `findPath(int, int)` to conduct search from s to t
- `st(int)` to return predecessor of v on path found

*Example: client code for grid graphs*

```java
int e = 0;
Edge[] a = new Edge[E];
for (int i = 0; i < V; i++)
    {
        if (i < V-M) a[e++] = new Edge(i, i+M);
        if (i >= M) a[e++] = new Edge(i, i-M);
        if ((i+1) % M != 0) a[e++] = new Edge(i, i+1);
        if (i % M != 0) a[e++] = new Edge(i, i-1);
    }
GRAPH G = new GRAPH(a);
G.findPath(V-1-M/2, M/2);
for (int k = t; k != s; k = G.st(k))
    System.out.println(s + "-" + t);
```
DFS: standard implementation

**graph ADT 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 findPathR(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]) searchR(x.v, t);
    }

void findPath(int s, int t)
    {
        visited = new boolean[V];
        searchR(s, t);
    }
```

**graph representation**

- vertex-indexed array of linked lists
- two nodes per edge
- grid graph representation
- DFS: standard implementation
Basic flaw in standard DFS scheme

cost strongly depends on arbitrary decision in client code!

```java
for (int i = 0; i < V; i++)
{
    if ((i+1) % M != 0) a[e++] = new Edge(i, i+1);
    if (i % M != 0) a[e++] = new Edge(i, i-1);
    if (i < V-M) a[e++] = new Edge(i, i+M);
    if (i >= M) a[e++] = new Edge(i, i-M);
}
```

... 

*west, east, north, south*

*south, north, east, west*

~$E/2$  ~$E^{1/2}$
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**

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

**randomized iterator**

```
int N = adj[x].length;
for(int i = 0; i < N; i++)
    { exch(adj[x], i, i + (int) Math.random()*(N-i));
        process vertex adj[x][i];
    }
```

represent graph with arrays, not lists

exchange random vertex from adj[x][i..N-1] with 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**

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

**DFS implementation (code to save path omitted)**

```java
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);
          }
    }

void findPath(int s, int t)
    { visited = new boolean[V];
      findpathR(s, t);
    }
```

**graph representation**

- vertex-indexed
- array of variable-length arrays
BFS: standard implementation

Use a queue to hold *fringe* vertices

```java
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;
            }
        }
    }
}
```

**Generalized graph search:** other queues yield A* and other graph-search algorithms
Union-Find implementation

1. Run union-find to find component containing $s$ and $t$

2. Build subgraph with edges from that component

3. Use DFS to find $st$-path in that subgraph
Animations give intuition on performance

- **BFS**
- **DFS**
- **UF**

and suggest hypotheses to verify with experimentation
Experimental results

for basic algorithms

DFS is substantially faster than BFS and UF 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>.75</td>
<td>.32</td>
<td>1.05</td>
</tr>
<tr>
<td>15</td>
<td>225</td>
<td>840</td>
<td>.75</td>
<td>.45</td>
<td>1.02</td>
</tr>
<tr>
<td>31</td>
<td>961</td>
<td>3720</td>
<td>.75</td>
<td>.36</td>
<td>1.14</td>
</tr>
<tr>
<td>63</td>
<td>3969</td>
<td>15624</td>
<td>.75</td>
<td>.32</td>
<td>1.05</td>
</tr>
<tr>
<td>127</td>
<td>16129</td>
<td>64008</td>
<td>.75</td>
<td>.40</td>
<td>.99</td>
</tr>
<tr>
<td>255</td>
<td>65025</td>
<td>259080</td>
<td>.75</td>
<td>.42</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Analytic proof?

Faster algorithms available?
A faster algorithm

for finding an st-path in a graph

Use two depth-first searches
• one from the source \( s \)
• one from the destination \( t \)
• interleave the two

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

Not loglog \( E \), but not bad!
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?
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?
Small-world graphs model the six degrees of separation phenomenon.

Example: Kevin Bacon number
Applications of small-world graphs

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

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

social networks
airlines
roads
neurobiology
evolution
social influence
protein interaction
percolation
internet
electric power grids
political trends
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 $M \sim E^{1/2}$ random edges to an $M$-by-$M$ grid graph
• use 2-way DFS to find path

Surprising result: Finds short paths in $\sim E^{1/2}$ 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: $2E^{1/2}$ steps $\sim 2V^{1/2}$ random vertices
   - step 2: like birthday paradox

Path length?
Multiple searchers revisited?

Next steps: refine model, more experiments, detailed proofs
Answers

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

Questions

- What are the BFS, UF, and DFS constants in grid graphs?
- Is there a sublinear algorithm for grid graphs?
- Which methods adapt to directed graphs?
- Can we precisely analyze and quantify costs for small-world graphs?
- What is the cost distribution for DFS for any interesting graph family?
- How effective are these methods for other graph families?
- Do these methods lead to faster maxflow algorithms?
- How effective are these methods in practice?
- ...
Lessons

We know much less than you might think about most of the algorithms that we use.

The scientific method is necessary in algorithm design and software development.
The scientific method

The scientific method is necessary in algorithm design and software development.

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

**1950s**
![Algorithm designer](image)

**2000s**
![Software developer](image)

**Algorithm designer** who does not experiment gets lost in abstraction

**Software developer** who ignores cost risks catastrophic consequences