Algorithm Engineering the Theory

Gerth Stølting Brodal
Aarhus University

ADYN Summer School on Algorithm Engineering for Network Problems, Hasso Plattner Institute, Potsdam, Germany, September 19-22, 2023
Gerth Stølting Brodal

Research
Data structures 1993 –

Teaching
Algorithms and Data Structures 2002 –
Introduction to Programming (Python) 2018 –
Bachelor project advising
Creating new theory is cool

Filling in proof details less exciting

Uncertainty – have all cases been addressed? [you prove the algorithm is correct but rarely that your proof is correct]

Frustrating when errors make their way into published papers 😞

Coding is healthy

Coding is fun

Debugging less so...

Procrastinating from writing the theory?

Document relevance of theory

Study theory vs real world

Identify shortcomings of theory

Inspire new theory
Goal

- Have more celebrations
- Make progress on bugs more frequently
- Not necessarily fewer bugs!
Certifying algorithms

R.M. McConnell\textsuperscript{a}, K. Mehlhorn\textsuperscript{b,\ast}, S. Náher\textsuperscript{c}, P. Schweitzer\textsuperscript{d}

\textsuperscript{a}Computer Science Department, Colorado State University Fort Collins, USA
\textsuperscript{b}Max Planck Institute for Informatics and Saarland University, Saarbrücken, Germany
\textsuperscript{c}Fachbereich Informatik, Universität Trier, Trier, Germany
\textsuperscript{d}College of Engineering and Computer Science, Australian National University, Canberra, Australia

ABSTRACT

A certifying algorithm is an algorithm that produces, with each output, a certificate or witness (easy-to-verify proof) that the particular output has not been compromised by a bug. A user of a certifying algorithm inputs \( x \), receives the output \( y \) and the certificate \( w \), and then checks, either manually or by use of a program, that \( w \) proves that \( y \) is a correct output for input \( x \). In this way, he/she can be sure of the correctness of the output without having to trust the algorithm.

We put forward the thesis that certifying algorithms are much superior to non-certifying algorithms, and that for complex algorithmic tasks, only certifying algorithms are satisfactory. Acceptance of this thesis would lead to a change of how algorithms are taught and how algorithms are researched. The widespread use of certifying algorithms would greatly enhance the reliability of algorithmic software.

We survey the state of the art in certifying algorithms and add to it. In particular, we start a theory of certifying algorithms and prove that the concept is universal.

Example:

Bipartite graph?

Yes

No

Certificate = two coloring

Certificate = odd cycle
Automatic testing of algorithm implementation

```python
while True:
    x = generate_random_input()
    answer, certificate = algorithm(x)
    assert verify(x, answer, certificate)
    print('.')
```

- Happy when sequence of dots grow
- Program crashes – debugger can perhaps help you find the bug
- Verification fails – a bug somewhere in the program/algorithm
Simplifying failed input (Greedy DFS)

```python
def simplify_bug(failed_input):
    for x in simplifications(failed_input):
        try:
            algorithm(x)
        except Bug:
            print('YES! - failed on', x)
        return simplify_bug(x)
    return failed_input
```

- `simplifications` could e.g. report all ways of removing a single vertex or edge from a graph
- Complex input triggering the bug can sometimes be simplified to an input of manageable size
Invariants

- **Invariants** are a fundamental tool when *designing* and *analyzing* algorithms and data structures

- Capture state of algorithm

- Example: AVL tree invariant
  1) Search tree
  2) $\forall v : |v\.left\.height - v\.right\.height | \leq 1$

- Invariants can be made **assertions** in code $\Rightarrow$ ensure code integrity
```python
def validate(tree, min_value=None, max_value=None):
    '''Validate AVL-tree invariants.''

    if not is_empty(tree):
        assert min_value == None or min_value <= tree.root
        assert max_value == None or tree.root <= max_value
        assert tree.height == 1 + max(tree.left.height, tree.right.height)
        assert abs(tree.left.height - tree.right.height) <= 1
        validate(tree.left, min_value, tree.root)
        validate(tree.right, tree.root, max_value)

def inorder(tree):
    '''Generator that yields values in tree in sorted order.''

    if not is_empty(tree):
        yield from inorder(tree.left)
        yield tree.root
        yield from inorder(tree.right)

def test_insertions(n):
    data = random.choices(range(10 * n), k=n)
    tree = empty_tree
    for i, x in enumerate(data):
        tree = insert(tree, x)
    validate(tree)
    assert sorted(data[:i + 1]) == list(inorder(tree))
```

- Write validate and test methods before implementing insert
“Test driven algorithm design”

- Formulate **invariants** as the driving tool for algorithm design
- Implement invariants in code as **assertions** and **verifier methods**
- Automate **stress tests**
- Develop algorithm through **failed tests**
  - ⇒ likely good coverage of special cases
  - ⇒ little redundant code

- Note: **verification methods might slow down code significantly (asymptotic slower!), but the focus is on developing correct theory**
def tikz(tree):
    def recurse(tree):
        if tree is empty_tree:
            return '{}'
        else:
            return f'[{tree.root} {recurse(tree.left)} {recurse(tree.right)}]'
    return r'\Tree ' + recurse(tree)
An unexpected journey

- Bachelor project = shortest paths on Open Street Map graphs
- Students have trouble implementing Dijkstra's algorithm in Java™
Dijkstra's algorithm (1956)

- Non-negative edge weights
- Visits nodes in increasing distance from source

\[
\begin{align*}
Q = \langle 0, A \rangle \\
\langle 2, B \rangle, \langle 4, C \rangle \\
\langle 3, C \rangle, \langle 6, D \rangle \\
\langle 4, D \rangle \\
\langle 6, E \rangle
\end{align*}
\]

```
proc Dijkstra₁(V, E, δ, s)
  dist[v] = +∞ for all v ∈ V \ {s}
  dist[s] = 0
  Insert(Q, (dist[s], s))
  while Q ≠ ∅ do
    (d, u) = ExtractMin(Q)
    for (u, v) ∈ E ∩ (\{u\} × V) do
      if dist[u] + δ(u, v) < dist[v] then
        dist[v] = dist[u] + δ(u, v)
        if v ∈ Q then
          DecreaseKey(Q, v, dist[v])
        else
          Insert(Q, (v, dist[v]))
    return dist
```

```
proc Dijkstra₂(V, E, δ, s)
  dist[v] = +∞ for all v ∈ V \ {s}
  dist[s] = 0
  Insert(Q, (dist[s], s))
  while Q ≠ ∅ do
    (d, u) = ExtractMin(Q)
    for (u, v) ∈ E ∩ (\{u\} × V) do
      if dist[u] + δ(u, v) < dist[v] then
        dist[v] = dist[u] + δ(u, v)
        if v ∈ Q then
          Remove(Q, v)
          Insert(Q, (v, dist[v]))
    return dist
```

- Relax
- Fibonacci heaps (Fredman, Tarjan 1984) ⇒ O(m + n · log n)
- O(log n) Remove ⇒ O(m · log n)
The challenge - Java's builtin binary heap

- no decreasekey
- remove $O(n)$ time
  $\Rightarrow$ Dijkstra $O(m \cdot n)$
- comparator function
Repeated insertions

- **Relax** inserts new copies of item
- **Skip outdated** items

```plaintext
proc Dijkstra_3(V, E, δ, s)
  \(dist[v] = +\infty\) for all \(v \in V \setminus \{s\}\)
  \(dist[s] = 0\)
  Insert\((Q, \langle dist[s], s \rangle)\)
  while \(Q \neq \emptyset\) do
    \(\langle d, u \rangle = \text{ExtractMin}(Q)\)
    if \(d = dist[u]\) then
      for \((u, v) \in E \cap (\{u\} \times V)\) do
        if \(dist[u] + \delta(u, v) < dist[v]\) then
          \(dist[v] = dist[u] + \delta(u, v)\)
          Insert\((Q, \langle dist[v], v \rangle)\)
    return \(dist\)
```

\[Q = \langle 0, A \rangle\]

\[\langle 2, B \rangle, \langle 4, C \rangle, \langle 6, D \rangle, \langle 6, E \rangle\]
Using a visited set

\[
\text{proc Dijkstra}_4(V, E, \delta, s) \\
\text{dist}[v] = +\infty \text{ for all } v \in V \setminus \{s\} \\
\text{dist}[s] = 0 \\
\text{visited} = \emptyset \\
\text{Insert}(Q, (\text{dist}[s], s)) \\
\text{while } Q \neq \emptyset \text{ do} \\
\qquad (d, u) = \text{ExtractMin}(Q) \\
\qquad \text{if } u \notin \text{visited} \text{ then} \\
\qquad\quad \text{visited} = \text{visited} \cup \{u\} \\
\qquad\quad \text{for } (u, v) \in E \cap (\{u\} \times V) \text{ do} \\
\qquad\qquad \text{if } \text{dist}[u] + \delta(u, v) < \text{dist}[v] \text{ then} \\
\qquad\qquad\quad \text{dist}[v] = \text{dist}[u] + \delta(u, v) \\
\qquad\qquad\quad \text{Insert}(Q, (\text{dist}[v], v)) \\
\text{return dist}
\]
A shaky idea...

\[
\text{proc Dijkstra}_4(V, E, \delta, s) \\
\quad \text{dist}[v] = +\infty \text{ for all } v \in V \setminus \{s\} \\
\quad \text{dist}[s] = 0 \\
\quad \text{visited} = \emptyset \\
\quad \text{Insert}(Q, \langle \text{dist}[s], s \rangle) \\
\quad \text{while } Q \neq \emptyset \text{ do} \\
\quad \quad \langle u \rangle = \text{ExtractMin}(Q) \\
\quad \quad \text{if } u \notin \text{visited} \text{ then} \\
\quad \quad \quad \text{visited} = \text{visited} \cup \{u\} \\
\quad \quad \quad \text{for } (u, v) \in E \cap (\{u\} \times V) \text{ do} \\
\quad \quad \quad \quad \text{if dist}[u] + \delta(u, v) < \text{dist}[v] \text{ then} \\
\quad \quad \quad \quad \quad \text{dist}[v] = \text{dist}[u] + \delta(u, v) \\
\quad \quad \quad \quad \quad \text{Insert}(Q, \langle \text{dist}[v], v \rangle) \\
\quad \text{return dist}
\]

- \(Q\) only store nodes (save space)
- Comparator
- Key = current distance dist

\[\text{Heap invariants break}\]
Experimental study

- Implemented Dijkstra\(_4\) in Python
- Stress test on random cliques
- Binary heaps failed (default priority queue in Java and Python)
Binary heaps using $\text{dist}$ in a comparator fails

- outdated
- wrong placement
- not smallest key
- ignored since visited
Experimental study

- Implemented Dijkstra\textsubscript{4} in Python
- Stress test on random cliques

- Binary heaps \quad failed \quad (default priority queue in Java and Python)
- Skew heaps \quad worked
- Leftist heaps \quad worked
- Pairing heaps \quad worked
- Binomial queues \quad worked
- Post-order heaps \quad worked
- Binary heaps with top-down insertions \quad worked

```python
visited = set()
Q = Queue()
Q.insert(Item(0, source))
while not Q.empty():
    u = Q.extract_min().value
    if u not in visited:
        visited.add(u)
        for v in G.out[u]:
            dist_v = dist[u] + G.weights[(u, v)]
            if dist_v < dist[v]:
                dist[v] = dist_v
                parent[v] = u
                Q.insert(Item(dist[v], v))
```

\textbf{unexpected}

- Pointer based
- Implicit (space efficient)
Binary heap insertions – bottom-up vs top-down

Insert(7)

Diagram showing the process of inserting 7 into a binary heap using both bottom-up and top-down methods.
Definition: Priority queues with **decreasing keys**

- Items = \(\langle\text{key, value}\rangle\)
- Over time keys can decrease – *priority queue is not informed*
- Items are compared w.r.t. their **current keys**
- The **original key** of an item is the key when it was inserted

**Insert** (item)

**ExtractMin** () returns an item with **current key less than or equal to all original keys** in the priority queue
Theorem 1

Dijkstra\textsubscript{4} correctly computes shortest paths when using \textit{dist} as current key and a priority queue supporting \textit{decreasing keys}

Theorem 2

The following priority queues support \textit{decreasing keys} (out of the box)

- binary heaps with top-down insertions
- skew heaps
- leftist heaps
- pairing heaps
- binomial queues
- post-order heaps
Proof of Theorem 2 - Basic idea

- Decreased heap order
  - $u$ ancestor of $v \Rightarrow$ current key $u \leq$ original key $v$

- Root valid item to extract

- Top-down merging two paths preserves decreased heap order
  - $\Rightarrow$ skew heaps and leftist heaps support decreasing keys
Experimental evaluation of various heaps

- Cliques with uniform random weights
- With decreasing keys less comparisons (outdated items removed earlier)

The diagrams show the number of comparisons versus the number of nodes for different heap implementations. The left diagram represents the number of comparisons for \((key, value)\) pairs, while the right diagram shows the number of comparisons for decreasing keys. The graph indicates that some heaps have smaller comparisons than others, with the slopes of the lines suggesting the efficiency of each heap.
Reduction in comparisons
comparisons decreasing keys / comparisons \langle \text{key, value} \rangle \text{ pairs}
Postorder heap [Harvey and Zatloukal, FUN 2004]

- **Insert** amortized $O(1)$, **ExtractMin** amortized $O(\log n)$
- Implicit (space efficient)
- Best implicit comparison performance (and good time performance)
Summary of the unexpected journey

- Introduced notion of **priority queues with decreasing keys** ... as an approach to deal with outdated items in Dijkstra’s algorithm
- Experiments identified priority queues supporting decreasing keys ... just had to prove it
- Built-in priority queues in Java and Python are binary heaps ... do not support decreasing keys
- **Binary heaps with top-down insertions** do support decreasing keys ... and also
  - skew heaps, leftist heaps, pairing heaps, binomial queues, post-order heaps
The reviewer is always right
"If there was a implementation where the authors verified that everything did what it was supposed to, I would be more confident that things were correct (I am not talking about a practical implementation, I am talking about one to make sure all invariants hold).”

Anonymous reviewer
## Strict Fibonacci heaps

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(log n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>DecreaseKey</td>
<td>O(log n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Meld</td>
<td>-</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Strict Fibonacci heaps

+ many structural invariants
Python implementation

- 1589 lines
- 215 assert statements
- All claimed invariants turned into assert statements
- Validation methods to traverse full structure to verify all claimed invariants
- Stress test using random inputs
- Supported the theory

[Diagram of Fibonacci heap structure]

[Image link: www.cs.au.dk/~gerth/strict_fibonacci_heaps.py]
Code coverage

- Used the Python module `coverage`
- Some code rarely executed
- Repeat random test 1,000,000 times
- Most code executed at least once

- Realized there was code for cases which provably never can occur
- Implementation → new invariants discovered
```python
odd_even.py
1  def f(x):
2      if x % 2 == 0:
3          return 'even'
4      elif x % 4 == 0:
6          return 'even more even'
8      import random
9      for i in range(10):
10     x = random.randint(0, 10)
11     print(x, f(x))
```

Shell

```bash
> coverage run odd_even.py
| 4 even
| 5 odd
| ...
| 1 odd
> coverage report -m odd_even.py
<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>odd_even.py</td>
<td>11</td>
<td>1</td>
<td>91%</td>
<td>5</td>
</tr>
<tr>
<td>TOTAL</td>
<td>11</td>
<td>1</td>
<td>91%</td>
<td></td>
</tr>
</tbody>
</table>
```

**Code coverage**

- Usually, code coverage is a measure of the quality of test cases
- ...but, can also help to identify missing logical insights
Branch coverage

- Thought code coverage would find all "logical errors"
- Found several if statements with no else part, where condition provably would always be true
- Implementation → new invariants discovered (and assertions added)

always exists
```python
def f(x):
    if x % 2 == 0:
        return 'even'
    elif x % 4 == 0:
        return 'even more even'
    elif x % 2 == 1:
        return 'odd'

import random
for i in range(10):
    x = random.randint(0, 10)
    print(x, f(x))
```

Shell output:
```
> coverage run --branch odd_even.py
  5 odd
  4 even
  ...
  8 even
> coverage report -m odd_even.py
  Name       Stmts  Miss  Branch  BrPart  Cover  Missing
  -------------------------------
  odd_even.py      11      1      8      2   84%   5, 6->exit
  -------------------------------
  TOTAL            11      1      8      2   84%
```
”The first main suggestion is to have at least one figure with a logical diagram of a non-trivial example structure, [...]. This would go a long way in giving some idea of what the structure is.”

Anonymous reviewer
- Hard to manually create a figure that was guaranteed to be a real example
- Could use implementation to automatically generate (LaTeX tikz) figures
- Generated random inputs
- Formalized requirements to figure as a loop condition
- Repeat until happy
After inserting $n$ random elements into an unbalanced binary search tree, what is the expected size of the subtree rooted at the minimum?
Implementations support stronger theory
Experimentation can identify what to prove
Invariants can be verified and identified using assertions in code
Stress tests and code coverage ensures integrity of code and theory