Constructing Combinatorial Designs for Software Testing

Michael Forbes
MIT
SURF, 2006 and 2007

Under Dr. James Lawrence
and Dr. Raghu Kacker
Math and Comp. Sci. Division
ITL, NIST

August 7, 2007
Kyoto University: 24 genes sufficient to make a cell a stem cell
Kyoto University: 24 genes sufficient to make a cell a stem cell
4 genes seem necessary
Kyoto University: 24 genes sufficient to make a cell a stem cell
4 genes seem necessary
Full testing requires $2^{24} = 16777216$ tests
Kyoto University: 24 genes sufficient to make a cell a stem cell
4 genes seem necessary
Full testing requires $2^{24} = 16777216$ tests, much too many!
Kyoto University: 24 genes sufficient to make a cell a stem cell
4 genes seem necessary
Full testing requires $2^{24} = 16777216$ tests, much too many!
Combinatorial designs
Kyoto University: 24 genes sufficient to make a cell a stem cell
4 genes seem necessary
Full testing requires $2^{24} = 16777216$ tests, much too many!
Combinatorial designs only need 2064 tests for subsets of 8
Kyoto University: 24 genes sufficient to make a cell a stem cell
4 genes seem necessary
Full testing requires $2^{24} = 16777216$ tests, much too many!
Combinatorial designs only need 2064 tests for subsets of 8, only 414 for subsets of 6
Kyoto University: 24 genes sufficient to make a cell a stem cell

4 genes seem necessary

Full testing requires $2^{24} = 16777216$ tests, much too many!

Combinatorial designs only need 2064 tests for subsets of 8, only 414 for subsets of 6 and only 72 for subsets of 4
Kyoto University: 24 genes sufficient to make a cell a stem cell
4 genes seem necessary
Full testing requires $2^{24} = 16777216$ tests, much too many!
Combinatorial designs only need 2064 tests for subsets of 8, only 414 for subsets of 6 and only 72 for subsets of 4
No statistics yet, but acceptable for software
Some Approaches to Software Testing
Some Approaches to Software Testing

- Implement standards
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard
- Automated Code Checkers
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard
- Automated Code Checkers
  - False positives
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard

- Automated Code Checkers
  - False positives
  - Doesn’t capture logic errors
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard
- Automated Code Checkers
  - False positives
  - Doesn’t capture logic errors
- Test Inputs
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard
- Automated Code Checkers
  - False positives
  - Doesn’t capture logic errors
- Test Inputs
  - Can identify many bugs
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard
- Automated Code Checkers
  - False positives
  - Doesn’t capture logic errors
- Test Inputs
  - Can identify many bugs
  - General testing framework
Some Approaches to Software Testing

- Implement standards
  - Ill-supported standards
  - Incomplete standard

- Automated Code Checkers
  - False positives
  - Doesn’t capture logic errors

- Test Inputs
  - Can identify many bugs
  - General testing framework
  - Requires a good set of test input
Testing with a Set of Test Inputs

- The components:
Testing with a Set of Test Inputs

- The components:
  - Generating a “good” set of test inputs
The components:

- Generating a “good” set of test inputs
- Verifying the program runs correctly on these inputs
Testing with a Set of Test Inputs

- The components:
  - Generating a “good” set of test inputs
  - Verifying the program runs correctly on these inputs
- Some methods for generating “good” sets:
Testing with a Set of Test Inputs

■ The components:
  □ Generating a “good” set of test inputs
  □ Verifying the program runs correctly on these inputs

■ Some methods for generating “good” sets:
  □ All possible inputs
Testing with a Set of Test Inputs

- The components:
  - Generating a “good” set of test inputs
  - Verifying the program runs correctly on these inputs

- Some methods for generating “good” sets:
  - All possible inputs — too many to test!
Testing with a Set of Test Inputs

- The components:
  - Generating a “good” set of test inputs
  - Verifying the program runs correctly on these inputs

- Some methods for generating “good” sets:
  - All possible inputs — too many to test!
  - Manual selection
Testing with a Set of Test Inputs

The components:
- Generating a “good” set of test inputs
- Verifying the program runs correctly on these inputs

Some methods for generating “good” sets:
- All possible inputs — too many to test!
- Manual selection — not scalable
Testing with a Set of Test Inputs

- The components:
  - Generating a “good” set of test inputs
  - Verifying the program runs correctly on these inputs

- Some methods for generating “good” sets:
  - All possible inputs — too many to test!
  - Manual selection — not scalable
  - Automated selection
Testing with a Set of Test Inputs

■ The components:
  □ Generating a “good” set of test inputs
  □ Verifying the program runs correctly on these inputs

■ Some methods for generating “good” sets:
  □ All possible inputs — too many to test!
  □ Manual selection — not scalable
  □ Automated selection — good, but how?
Every error is caused by the interaction of certain inputs
A Helpful Assumption

- Every error is caused by the interaction of certain inputs
- We assume errors are caused by a small number of inputs being in a certain state
A Helpful Assumption

- Every error is caused by the interaction of certain inputs
- We assume errors are caused by a small number of inputs being in a certain state
- Empirical studies show that most errors are caused by the dependence of no more than 6 inputs
Defining the Coverage Property
Defining the Coverage Property

- A general system:
Defining the Coverage Property

- A general system:
  - $\kappa$ is the number of inputs
Defining the Coverage Property

- A general system:
  - $\kappa$ is the number of inputs
  - $\nu$ is the number of values each input can take on
A general system:

- $k$ is the number of inputs
- $v$ is the number of values each input can take on
- $t$ is the maximum number of inputs an error can be caused by
A general system:
- $k$ is the number of inputs
- $v$ is the number of values each input can take on
- $t$ is the maximum number of inputs an error can be caused by

We can define $t$-way coverage of a set of $k$ inputs as a set of test sets such that for any $t$ inputs, all $v^t$ possible input combinations are represented.
A general system:

- $k$ is the number of inputs
- $v$ is the number of values each input can take on
- $t$ is the maximum number of inputs an error can be caused by

We can define $t$-way coverage of a set of $k$ inputs as a set of test sets such that for any $t$ inputs, all $v^t$ possible input combinations are represented.

A set of test inputs that satisfies the $t$-way coverage property is a *covering array*.
Example of 2-way coverage
Example of 2-way coverage

Testing the NIST web page with:
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS, Graphical or Text-only browsing:
Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS, Graphical or Text-only browsing:

<table>
<thead>
<tr>
<th>Browser</th>
<th>OS</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Windows</td>
<td>Graphical</td>
</tr>
<tr>
<td>IE</td>
<td>Windows</td>
<td>Text</td>
</tr>
<tr>
<td>IE</td>
<td>MacOS</td>
<td>Graphical</td>
</tr>
<tr>
<td>IE</td>
<td>MacOS</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>Windows</td>
<td>Graphical</td>
</tr>
<tr>
<td>Firefox</td>
<td>Windows</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>MacOS</td>
<td>Graphical</td>
</tr>
<tr>
<td>Firefox</td>
<td>MacOS</td>
<td>Text</td>
</tr>
</tbody>
</table>
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS, Graphical or Text-only browsing:

<table>
<thead>
<tr>
<th>Browser</th>
<th>OS</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Windows</td>
<td>Graphical</td>
</tr>
<tr>
<td>IE</td>
<td>MacOS</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>Windows</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>MacOS</td>
<td>Graphical</td>
</tr>
</tbody>
</table>
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS, Graphical or Text-only browsing:

<table>
<thead>
<tr>
<th>Browser</th>
<th>OS</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Windows</td>
<td>Graphical</td>
</tr>
<tr>
<td>IE</td>
<td>MacOS</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>Windows</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>MacOS</td>
<td>Graphical</td>
</tr>
</tbody>
</table>
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS, Graphical or Text-only browsing:

<table>
<thead>
<tr>
<th>Browser</th>
<th>OS</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Windows</td>
<td>Graphical</td>
</tr>
<tr>
<td>IE</td>
<td>MacOS</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>Windows</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>MacOS</td>
<td>Graphical</td>
</tr>
</tbody>
</table>
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS, Graphical or Text-only browsing:

<table>
<thead>
<tr>
<th>Browser</th>
<th>OS</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Windows</td>
<td>Graphical</td>
</tr>
<tr>
<td>IE</td>
<td>MacOS</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>Windows</td>
<td>Text</td>
</tr>
<tr>
<td>Firefox</td>
<td>MacOS</td>
<td>Graphical</td>
</tr>
</tbody>
</table>
Example of 2-way coverage

Testing the NIST web page with: Internet Explorer (IE) or Firefox, Windows or MacOS, Graphical or Text-only browsing:

<table>
<thead>
<tr>
<th>Browser</th>
<th>OS</th>
<th>Mode</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Windows</td>
<td>Graphical</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IE</td>
<td>MacOS</td>
<td>Text</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Firefox</td>
<td>Windows</td>
<td>Text</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Firefox</td>
<td>MacOS</td>
<td>Graphical</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

→
Strategies for Generation
Strategies for Generation

- Complete Search
Strategies for Generation

- Complete Search
  - Optimal results
Strategies for Generation

- **Complete Search**
  - Optimal results
  - Exponential run time
Strategies for Generation

- Complete Search
  - Optimal results
  - Exponential run time
- Greedy Search
Strategies for Generation

- Complete Search
  - Optimal results
  - Exponential run time

- Greedy Search — A greedy search is one that assumes that optimal short-run choices will produce optimal long-run choices
Strategies for Generation

- Complete Search
  - Optimal results
  - Exponential run time

- Greedy Search — A greedy search is one that assumes that optimal short-run choices will produce optimal long-run choices
  - Decent results
Strategies for Generation

- Complete Search
  - Optimal results
  - Exponential run time

- Greedy Search — A greedy search is one that assumes that optimal short-run choices will produce optimal long-run choices
  - Decent results
  - Faster runtime
Examples of Greedy Algorithms
Examples of Greedy Algorithms

- The Hedonist
Examples of Greedy Algorithms

- The Hedonist
  - Gets marginally richer
Examples of Greedy Algorithms

- The Hedonist
  - Gets marginally richer
  - Loses out on friends
Examples of Greedy Algorithms

- The Hedonist
  - Gets marginally richer
  - Loses out on friends

- The Friendly Neighbor
Examples of Greedy Algorithms

- The Hedonist
  - Gets marginally richer
  - Loses out on friends
- The Friendly Neighbor
  - Calculates the route without much effort
Examples of Greedy Algorithms

- The Hedonist
  - Gets marginally richer
  - Loses out on friends

- The Friendly Neighbor
  - Calculates the route without much effort
  - Fairly efficient route
The In-Parameter-Order (IPO) Strategy

- Implemented in FireEye software, developed by NIST, University of Texas at Arlington, and George Mason University
The In-Parameter-Order (IPO) Strategy

- Implemented in FireEye software, developed by NIST, University of Texas at Arlington, and George Mason University
- Based on IPO algorithm developed by Kuo-Chung Tai and Jeff Lei
The In-Parameter-Order (IPO) Strategy

- Implemented in FireEye software, developed by NIST, University of Texas at Arlington, and George Mason University
- Based on IPO algorithm developed by Kuo-Chung Tai and Jeff Lei
- Start with a covering array of $k - 1$ columns
The In-Parameter-Order (IPO) Strategy

- Implemented in FireEye software, developed by NIST, University of Texas at Arlington, and George Mason University
- Based on IPO algorithm developed by Kuo-Chung Tai and Jeff Lei
- Start with a covering array of $k - 1$ columns
- *Horizontal Growth*: add a column to best preserve the $t$-way coverage property
The In-Parameter-Order (IPO) Strategy

- Implemented in FireEye software, developed by NIST, University of Texas at Arlington, and George Mason University
- Based on IPO algorithm developed by Kuo-Chung Tai and Jeff Lei
- Start with a covering array of \( k - 1 \) columns
- **Horizontal Growth**: add a column to best preserve the \( t \)-way coverage property
- **Vertical Growth**: restore the \( t \)-way coverage property by adding rows
Covering array with 2-way coverage of 4 inputs, 2 values per input:
Covering array with 2-way coverage of 4 inputs, 2 values per input:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Covering array with 2-way coverage of 4 inputs, 2 values per input:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Covering array with 2-way coverage of 4 inputs, 2 values per input:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Covering array with 2-way coverage of 4 inputs, 2 values per input:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Covering array with 2-way coverage of 4 inputs, 2 values per input:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Covering array with 2-way coverage of 4 inputs, 2 values per input:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Covering array with 2-way coverage of 4 inputs, 2 values per input:

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{array}
\]
Vertical growth is near optimal in both array size and speed
Assessing IPO’s Performance

- Vertical growth is near optimal in both array size and speed
- Most of the time is spent in the horizontal growth stage
Assessing IPO’s Performance

- Vertical growth is near optimal in both array size and speed
- Most of the time is spent in the horizontal growth stage
- Horizontal growth algorithms mostly determine optimality of results
Assessing IPO’s Performance

- Vertical growth is near optimal in both array size and speed
- Most of the time is spent in the horizontal growth stage
- Horizontal growth algorithms mostly determine optimality of results
- Horizontal growth algorithms could achieve better results
Improving IPO

Making the algorithm more greedy
Making the algorithm more greedy

- The row order was fixed
Making the algorithm more greedy

- The row order was fixed
- Allow the row order to be determined greedily
Making the algorithm more greedy

- The row order was fixed
- Allow the row order to be determined greedily
- Search through all row/value pairs
Making the algorithm more greedy

- The row order was fixed
- Allow the row order to be determined greedily
- Search through all row/value pairs
- Record which row/value pairs give the most coverage
Making the algorithm more greedy

- The row order was fixed
- Allow the row order to be determined greedily
- Search through all row/value pairs
- Record which row/value pairs give the most coverage
- Randomly pick one of listed pairs
Improving IPO

Making the algorithm more greedy

- The row order was fixed
- Allow the row order to be determined greedily
- Search through all row/value pairs
- Record which row/value pairs give the most coverage
- Randomly pick one of listed pairs
- Extend the array with this pair
Making the algorithm more greedy

- The row order was fixed
- Allow the row order to be determined greedily
- Search through all row/value pairs
- Record which row/value pairs give the most coverage
- Randomly pick one of listed pairs
- Extend the array with this pair

Use dynamic programming
Making the algorithm more greedy

- The row order was fixed
- Allow the row order to be determined greedily
- Search through all row/value pairs
- Record which row/value pairs give the most coverage
- Randomly pick one of listed pairs
- Extend the array with this pair

Use dynamic programming - store calculations so we can use them later in an efficient way
Covering Array Size Comparison

CA(3,k,3) Size

\[ k \]

IPO’

FireEye
Execution Time Comparison

![Graph showing execution time comparison for FireEye/IPO ratio against k.]
Best Known Covering Arrays

\[ \text{CA}(4,k,3) \text{ Size} \]

- IPO'
- Best

$k$ vs. \( \text{CA}(4,k,3) \text{ Size} \)
Some Conclusions

- Covering arrays help testing software
Some Conclusions

- Covering arrays help testing software
- There exist several algorithms for generating covering arrays
Some Conclusions

- Covering arrays help testing software
- There exist several algorithms for generating covering arrays
- The greedy approach is key to these algorithms
Some Conclusions

- Covering arrays help testing software
- There exist several algorithms for generating covering arrays
- The greedy approach is key to these algorithms
- Making algorithms “more greedy” increases optimality
Some Conclusions

• Covering arrays help testing software
• There exist several algorithms for generating covering arrays
• The greedy approach is key to these algorithms
• Making algorithms “more greedy” increases optimality
• Dynamic programming makes the program run faster
Some Conclusions

- Covering arrays help testing software
- There exist several algorithms for generating covering arrays
- The greedy approach is key to these algorithms
- Making algorithms “more greedy” increases optimality
- Dynamic programming makes the program run faster
- Produced covering arrays smaller than ever seen before
**Horizontal Growth:**

- Original time bound: \( O(rv \binom{k-1}{t-1}) \)
- My time bound (approx.): \( O\left(\frac{r^2}{vt} v \binom{k-1}{t-1}\right) \) (note \( r > vt \))
Theoretical Results

**Horizontal Growth:**

- Original time bound: $O(rv\binom{k-1}{t-1})$
- My time bound (approx.): $O\left(\frac{r^2}{vt}v\binom{k-1}{t-1}\right)$ (note $r > vt$)

The theoretical covering array bound:

$$CA(t, k, v) \leq \frac{\ln(v^t\binom{k}{t})}{\ln\left(\frac{vt}{vt-1}\right)} < tv^t \ln(vk)$$