Automatically Computing Path Complexity of Programs

Lucas Bang, Abdulbaki Aydin, Tevfik Bultan
{bang,baki,bultan}@cs.ucsb.edu

Department of Computer Science
University of California, Santa Barbara

UCSB GSWC 2016
Overview: What did we do?
Overview: What did we do?
Overview: What did we do?

PAth Complexity Analyzer (PAC)

Program Counting Function, path $n$

Path Length Bound, $n$

Number of paths within length $n$

Asymptotic Behavior $\text{path}(n) = \Theta(f(n))$
Overview: What did we do?

Program → PAth Complexity Analyzer (PAC)
Overview: What did we do?

Program

JAVA

PPath
Complexity
Analyzer (PAC)

Counting
Function,
path(n)
Overview: What did we do?

Program

JAVA

PATH Complexity Analyzer (PAC)

Path Length Bound, \( n \)

Counting Function, \( path(n) \)
Overview: What did we do?

Program

PATH

Complexity Analyzer (PAC)

Path Length Bound, \( n \)

Counting Function, \( path(n) \)

Number of paths within length \( n \)
Overview: What did we do?

Program

JAVA

PATH Complexity Analyzer (PAC)

Path Length Bound, \( n \)

Counting Function, \( path(n) \)

Asymptotic Behavior

\( path(n) = \Theta(f(n)) \)

Number of paths within length \( n \)
Can you solve it, Will Hunting?
Can you solve it, Will Hunting?

Given the graph

Find 1) the adjacency matrix $A$

2) the matrix giving the number of 3 step walks

3) the generating function for walks from point $i \to j$

4) the generating function for walks from points $1 \to 3$. 
Outline

Motivation

Path Complexity

Experiments
Motivation

Program Path Coverage
Motivation

Program Path Coverage

- Modern automated software testing techniques focus on program path coverage.
Motivation

Program Path Coverage

- Modern automated software testing techniques focus on program path coverage.
- The number of execution paths could be infinite.
Motivation

Program Path Coverage

- Modern automated software testing techniques focus on program path coverage.
- The number of execution paths could be infinite.
- Practical solution: explore up to a given depth bound.
Motivation

Program Path Coverage

- Modern automated software testing techniques focus on program path coverage.
- The number of execution paths could be infinite.
- Practical solution: explore up to a given depth bound.
- We propose a metric, the **path complexity**, an upper bound on the number of paths needed to explore up to a given depth.
Motivation

Program Path Coverage

- Modern automated software testing techniques focus on program path coverage.
- The number of execution paths could be infinite.
- Practical solution: explore up to a given depth bound.
- We propose a metric, the **path complexity**, an upper bound on the number of paths needed to explore up to a given depth.
- This provides a measure of the difficulty of achieving path coverage.
Path Complexity

```java
boolean passCheck1()
{
    while (i < n)
    {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}
```
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}
Path Complexity

boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}

Given a control flow graph and a length bound $n$, let

- $count(n)$ be the number of paths of length **exactly** $n$. 

```java
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}
```
Path Complexity

boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}

Given a control flow graph and a length bound $n$, let

- $\textit{count}(n)$ be the number of paths of length \textbf{exactly} $n$.
- $\textit{path}(n)$ be the number of paths of length \textbf{less than or equal} $n$, i.e. the accumulated sum of $\textit{count}(n)$. 
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}

Given a control flow graph and a length bound \( n \), let

- \( \text{count}(n) \) be the number of paths of length \textbf{exactly} \( n \).
- \( \text{path}(n) \) be the number of paths of length \textbf{less than or equal} \( n \), i.e. the accumulated sum of \( \text{count}(n) \).
- \textbf{Path Complexity} is given by \( \text{path}(n) \).
Path Complexity

```java
boolean passCheck1()
    while (i<n)
        if (p[i] != pass[i])
            return false;
        i++;
    return true;
```

<table>
<thead>
<tr>
<th>n</th>
<th>count(n)</th>
<th>path(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Given a control flow graph and a length bound $n$, let:

- $count(n)$ be the number of paths of length **exactly** $n$.
- $path(n)$ be the number of paths of length **less than or equal** $n$, i.e. the accumulated sum of $count(n)$.
- **Path Complexity** is given by $path(n)$.
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}

Given a control flow graph and a length bound \( n \), let

- \(\text{count}(n)\) be the number of paths of length exactly \( n \).
- \(\text{path}(n)\) be the number of paths of length less than or equal \( n \), i.e. the accumulated sum of \(\text{count}(n)\).
- **Path Complexity** is given by \(\text{path}(n)\).
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}

<table>
<thead>
<tr>
<th>n</th>
<th>count(n)</th>
<th>path(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Given a control flow graph and a length bound \( n \), let

- \( \text{count}(n) \) be the number of paths of length exactly \( n \).
- \( \text{path}(n) \) be the number of paths of length less than or equal \( n \), i.e. the accumulated sum of \( \text{count}(n) \).
- **Path Complexity** is given by \( \text{path}(n) \).
Path Complexity

boolean passCheck1()
{
    while (i<n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}

Given a control flow graph and a length bound $n$, let

- $\text{count}(n)$ be the number of paths of length exactly $n$.
- $\text{path}(n)$ be the number of paths of length less than or equal $n$, i.e. the accumulated sum of $\text{count}(n)$.
- **Path Complexity** is given by $\text{path}(n)$.  

<table>
<thead>
<tr>
<th>n</th>
<th>count($n$)</th>
<th>path($n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Path Complexity

Given a control flow graph and a length bound $n$, let

- $\textit{count}(n)$ be the number of paths of length exactly $n$.
- $\textit{path}(n)$ be the number of paths of length less than or equal $n$, i.e. the accumulated sum of $\textit{count}(n)$.

- **Path Complexity** is given by $\textit{path}(n)$.
Given a control flow graph and a length bound $n$, let

- $\textit{count}(n)$ be the number of paths of length \textit{exactly} $n$.
- $\textit{path}(n)$ be the number of paths of length \textit{less than or equal} $n$, i.e. the accumulated sum of $\textit{count}(n)$.
- \textbf{Path Complexity} is given by $\textit{path}(n)$. 

```java
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}
```
Given a control flow graph and a length bound $n$, let

- $\textit{count}(n)$ be the number of paths of length \textbf{exactly} $n$.
- $\textit{path}(n)$ be the number of paths of length \textbf{less than or equal} $n$, i.e. the accumulated sum of $\textit{count}(n)$.
- \textbf{Path Complexity} is given by $\textit{path}(n)$.
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}
boolean passCheck1() {
    while (i < n) {
        if (p[i] != pass[i])
            return false;
        i++;
    }
    return true;
}

<table>
<thead>
<tr>
<th>n</th>
<th>count(n)</th>
<th>path(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Appears to grow linearly... is \( \text{path}(n) \approx \frac{2}{3} n \)?
boolean passCheck2() {
    matched = true;
    while (i < n) {
        if (p[i] != pass[i]) {
            matched = false;
            i++;
        }
    }
    return matched;
}
boolean passCheck2() {
    matched = true;
    while (i < n) {
        if (p[i] != pass[i]) {
            matched = false;
            i++;
        }
    }
    return matched;
}
Path Complexity

boolean passCheck2()
{
    matched = true;
    while (i < n) {
        if (p[i] != pass[i])
            matched = false;
        i++;
    }
    return matched;
}

<table>
<thead>
<tr>
<th>n</th>
<th>count(n)</th>
<th>path(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
boolean passCheck2() {
    matched = true;
    while (i < n) {
        if (p[i] != pass[i])
            matched = false;
        i++;
    }
    return matched;
}

Could be polynomial or exponential?
The path complexity problem:
Path Complexity

The path complexity problem:

▶ How to compute $\text{path}(n)$ automatically?
The path complexity problem:

- How to compute $\text{path}(n)$ \textbf{automatically}? 
- What is the \textbf{asymptotic behavior} of $\text{path}(n)$?
Matrix Exponentiation

- We can compute $\text{path}(n)$ using the $p \times p$ adjacency matrix, $A$, of the CFG, augmented with an additional 1 entry in the final column and final row.

- $\text{path}(n) = (A^n)_{1,p}$
Matrix Exponentiation

- We can compute \( \text{path}(n) \) using the \( p \times p \) adjacency matrix, \( A \), of the CFG, augmented with an additional 1 entry in the final column and final row.

- \( \text{path}(n) = (A^n)_{1,p} \)

\[
A = \begin{bmatrix}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
A^6 = \begin{bmatrix}
1 & 0 & 0 & 4 \\
0 & 1 & 0 & 4 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Drawback: repeated evaluations become expensive.
Matrix Exponentiation

- We can compute $\text{path}(n)$ using the $p \times p$ adjacency matrix, $A$, of the CFG, augmented with an additional 1 entry in the final column and final row.

- $\text{path}(n) = (A^n)_{1,p}$

$$A = \begin{bmatrix}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}$$

$$A^6 = \begin{bmatrix}
1 & 0 & 0 & 4 \\
0 & 1 & 0 & 4 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1
\end{bmatrix}$$

$\text{path}(6) = 4$
Matrix Exponentiation

- We can compute $\text{path}(n)$ using the $p \times p$ adjacency matrix, $A$, of the CFG, augmented with an additional 1 entry in the final column and final row.
- $\text{path}(n) = (A^n)_{1,p}$

![Graph]

$A = \begin{bmatrix}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}$

$A^6 = \begin{bmatrix}
1 & 0 & 0 & 4 \\
0 & 1 & 0 & 4 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1
\end{bmatrix}$

$\text{path}(6) = 4$

Drawback: repeated evaluations become expensive.
Generating Functions

- **Generating functions** are a mathematical tool for representing sequences.

The generating function for counting paths in a graph is given by

\[ g(z) = (-1)^{m+1} \det(1 - zA):^{m,1} \det(1 - zA) \]

In our example CFG, the generating function is

\[ g(z) = z(1+z)(1-z)(1-z^3) \]

Path \((n)\) is given by the \(n\)th Taylor series coefficient of \(g(z)\).

\[ g(z) = g(0) + g'(0) \frac{z^1}{1!} + g''(0) \frac{z^2}{2!} + g'''(0) \frac{z^3}{3!} + \ldots \]

For our example, the Taylor-series expansion is

\[ g(z) = 0z^0 + 1z^1 + 2z^2 + 2z^3 + 3z^4 + 4z^5 + 4z^6 + \ldots \]
Generating Functions

- **Generating functions** are a mathematical tool for representing sequences. (i.e. \( path(n) = 0, 1, 2, 2, 3, 4, 4, \ldots \))
Generating Functions

- Generating functions are a mathematical tool for representing sequences. (i.e. $\text{path}(n) = 0, 1, 2, 2, 3, 4, 4, \ldots$)
- The generating function for counting paths in a graph is given by

$$g(z) = (-1)^{m+1} \frac{\det(\mathbb{I} - zA : m, 1)}{\det(\mathbb{I} - zA)}$$

- The $n$th Taylor series coefficient of $g(z)$ is given by the $n$th Taylor series coefficient of $g(z)$. $g(z) = g(0) + \frac{g'(0)}{1!}z + \frac{g''(0)}{2!}z^2 + \frac{g'''(0)}{3!}z^3 + \ldots$

- For our example, the Taylor-series expansion is

$$g(z) = 0 + 1z + 2z^2 + 2z^3 + 3z^4 + 4z^5 + 4z^6 + \ldots$$
Generating Functions

- **Generating functions** are a mathematical tool for representing sequences. (i.e. \( \text{path}(n) = 0, 1, 2, 2, 3, 4, 4, \ldots \))

- The generating function for counting paths in a graph is given by

\[
g(z) = (-1)^{m+1} \frac{\det(\mathbb{I} - zA : m, 1)}{\det(\mathbb{I} - zA)}
\]

- In our example CFG, the generating function is

\[
g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)}
\]
Generating Functions

- **Generating functions** are a mathematical tool for representing sequences, (i.e. $\text{path}(n) = 0, 1, 2, 2, 3, 4, 4, \ldots$)
- The generating function for counting paths in a graph is given by

$$g(z) = (-1)^{m+1} \frac{\det(\mathbb{I} - zA : m, 1)}{\det(\mathbb{I} - zA)}$$

- In our example CFG, the generating function is

$$g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)}$$

- $\text{path}(n)$ is given by the $n^{th}$ Taylor series coefficient of $g(z)$.

$$g(z) = \frac{g(0)}{0!}z^0 + \frac{g'(0)}{1!}z^1 + \frac{g''(0)}{2!}z^2 + \frac{g'''(0)}{3!}z^3 + \ldots$$
Generating Functions

- **Generating functions** are a mathematical tool for representing sequences. (i.e. \( path(n) = 0, 1, 2, 2, 3, 4, 4, \ldots \))
- The generating function for counting paths in a graph is given by

\[
g(z) = (-1)^{m+1} \frac{\det(\mathbb{I} - zA : m, 1)}{\det(\mathbb{I} - zA)}
\]

- In our example CFG, the generating function is

\[
g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)}
\]

- \( path(n) \) is given by the \( n^{th} \) Taylor series coefficient of \( g(z) \).

\[
g(z) = \frac{g(0)}{0!} z^0 + \frac{g'(0)}{1!} z^1 + \frac{g''(0)}{2!} z^2 + \frac{g'''(0)}{3!} z^3 + \ldots
\]

- For our example, the Taylor-series expansion is

\[
g(z) = 0z^0 + 1z^1 + 2z^2 + 2z^3 + 3z^4 + 4z^5 + 4z^6 + 5z^7 + \ldots
\]
Generating Functions

- **Generating functions** are a mathematical tool for representing sequences. (i.e. $\text{path}(n) = 0, 1, 2, 2, 3, 4, 4, \ldots$)
- The generating function for counting paths in a graph is given by

$$g(z) = (-1)^{m+1} \frac{\det(\mathbb{I} - zA : m, 1)}{\det(\mathbb{I} - zA)}$$

- In our example CFG, the generating function is

$$g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)}$$

- $\text{path}(n)$ is given by the $n^{th}$ Taylor series coefficient of $g(z)$.

$$g(z) = \frac{g(0)}{0!} z^0 + \frac{g'(0)}{1!} z^1 + \frac{g''(0)}{2!} z^2 + \frac{g'''(0)}{3!} z^3 + \ldots$$

- For our example, the Taylor-series expansion is

$$g(z) = 0z^0 + 1z^1 + 2z^2 + 2z^3 + 3z^4 + 4z^5 + 4z^6 + 5z^7 + \ldots$$

\[ \text{path}(6) = 4 \]
Good job, Will Hunting!

This is correct. Who did this?
Good job, Will Hunting!
Good job, Will Hunting!
Good job, Will Hunting!
Closed-form Solution

A closed-form solution can be computed from the generating function.

\[ g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)} \]
Closed-form Solution

A closed-form solution can be computed from the generating function.

\[ g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)} \]

Find the roots of the denominator

\[ (1 - z)(1 - z^3) = 0 \implies z = 1, 1, \frac{-1 + \sqrt{3}i}{2}, \frac{-1 - \sqrt{3}i}{2} \]
Closed-form Solution

A closed-form solution can be computed from the generating function.

\[ g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)} \]

- Find the roots of the denominator

\[(1 - z)(1 - z^3) = 0 \quad \implies \quad z = 1, 1, \frac{-1 + \sqrt{3}i}{2}, \frac{-1 - \sqrt{3}i}{2} \]

- Take a linearly independent combination of exponentiated roots:

\[ path(n) = c_1 \cdot 1^n + c_2 n \cdot 1^n + c_3 \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + c_4 \left( \frac{-1 - \sqrt{3}i}{2} \right)^n \]
Closed-form Solution

A closed-form solution can be computed from the generating function.

\[ g(z) = \frac{z(1 + z)}{(1 - z)(1 - z^3)} \]

- Find the roots of the denominator

\[(1 - z)(1 - z^3) = 0 \implies z = 1, \frac{-1 + \sqrt{3}i}{2}, \frac{-1 - \sqrt{3}i}{2} \]

- Take a linearly independent combination of exponentiated roots:

\[ path(n) = c_1 \cdot 1^n + c_2 n \cdot 1^n + c_3 \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + c_4 \left( \frac{-1 - \sqrt{3}i}{2} \right)^n \]

- Solve for coefficients \( c_1, c_2, \ldots \)

\[ path(n) = \frac{1}{3} + \frac{2}{3} n + \left( \frac{-3 + \sqrt{3}}{18} \right) \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + \left( \frac{-3 - \sqrt{3}}{18} \right) \left( \frac{-1 - \sqrt{3}i}{2} \right)^n \]
Tight bounds for path(n)

Our solution looks very... complex

\[
path(n) = \frac{1}{3} + \frac{2}{3}n + \left(\frac{-3 + \sqrt{3}}{18}\right) \left(\frac{-1 + \sqrt{3}i}{2}\right)^n + \left(\frac{-3 - \sqrt{3}}{18}\right) \left(\frac{-1 - \sqrt{3}i}{2}\right)^n
\]
Tight bounds for $\text{path}(n)$

Our solution looks very... complex

$$
\text{path}(n) = \frac{1}{3} + \frac{2}{3}n + \left( \frac{-3 + \sqrt{3}}{18} \right) \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + \left( \frac{-3 - \sqrt{3}}{18} \right) \left( \frac{-1 - \sqrt{3}i}{2} \right)^n
$$

- For any complex number $w$, we have the tight bounds

$$
-2|w|^n \leq |w^n + \overline{w}^n| \leq 2|w|^n
$$
Tight bounds for $\text{path}(n)$

Our solution looks very... complex

\[
\text{path}(n) = \frac{1}{3} + \frac{2}{3}n + \left( \frac{-3 + \sqrt{3}}{18} \right) \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + \left( \frac{-3 - \sqrt{3}}{18} \right) \left( \frac{-1 - \sqrt{3}i}{2} \right)^n
\]

For any complex number $w$, we have the tight bounds

\[
-2|w|^n \leq |w^n + \overline{w}^n| \leq 2|w|^n
\]

\[
-\frac{1}{3} \leq \left( \frac{-3 + \sqrt{3}}{18} \right) \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + \left( \frac{-3 - \sqrt{3}}{18} \right) \left( \frac{-1 - \sqrt{3}i}{2} \right)^n \leq \frac{1}{3}
\]
Tight bounds for $\text{path}(n)$

Our solution looks very... complex

$$\text{path}(n) = \frac{1}{3} + \frac{2}{3}n + \left( \frac{-3 + \sqrt{3}}{18} \right) \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + \left( \frac{-3 - \sqrt{3}}{18} \right) \left( \frac{-1 - \sqrt{3}i}{2} \right)^n$$

► For any complex number $w$, we have the tight bounds

$$-2|w|^n \leq |w^n + \overline{w}^n| \leq 2|w|^n$$

$$-\frac{1}{3} \leq \left( \frac{-3 + \sqrt{3}}{18} \right) \left( \frac{-1 + \sqrt{3}i}{2} \right)^n + \left( \frac{-3 - \sqrt{3}}{18} \right) \left( \frac{-1 - \sqrt{3}i}{2} \right)^n \leq \frac{1}{3}$$

Now, it looks much simpler:

$$\frac{2n}{3} \leq \text{path}(n) \leq \frac{2n}{3} + \frac{2}{3}$$
Tight bounds for \( \text{path}(n) \)
Asymptotic Behavior

- We extract the highest order term using standard asymptotic analysis from calculus

\[ f = \Theta(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \]
Asymptotic Behavior

- We extract the highest order term using standard asymptotic analysis from calculus

\[ f = \Theta(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \]

- Applied to our examples:
Asymptotic Behavior

- We extract the highest order term using standard asymptotic analysis from calculus

\[ f = \Theta(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \]

- Applied to our examples:
  - Function `passCheck1()`
  
  \[ path(n) = \Theta(n) \]
Asymptotic Behavior

- We extract the highest order term using standard asymptotic analysis from calculus

\[ f = \Theta(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1 \]

- Applied to our examples:
  - Function `passCheck1()`
    \[ path(n) = \Theta(n) \]
  - Function `passCheck2()`
    \[ path(n) = \Theta(1.221^n) \]
Complexity Classes

Classify path complexities as constant, polynomial, or exponential.
Complexity Classes

Classify path complexities as constant, polynomial, or exponential.

Examples from Java SDK 7.
Complexity Classes

Classify path complexities as **constant**, polynomial, or exponential.

Examples from Java SDK 7.

```java
private static void rangeCheck(int length, int fromIndex, int toIndex) {
    if (fromIndex > toIndex) {
        throw new IllegalArgumentException("fromIndex(" + fromIndex + ") > toIndex(" + toIndex + ")");
    }
    if (fromIndex < 0) {
        throw new ArrayIndexOutOfBoundsException(fromIndex);
    }
    if (toIndex > length) {
        throw new ArrayIndexOutOfBoundsException(toIndex);
    }
}
```
Complexity Classes

Classify path complexities as **constant**, polynomial, or exponential.

Examples from Java SDK 7.

```java
private static void rangeCheck(int length, int fromIndex, int toIndex) {
    if (fromIndex > toIndex) {
        throw new IllegalArgumentException("fromIndex(\(\) + fromIndex + ") >
            toIndex(" + toIndex + ");
    }
    if (fromIndex < 0) {
        throw new ArrayIndexOutOfBoundsException(fromIndex);
    }
    if (toIndex > length) {
        throw new ArrayIndexOutOfBoundsException(toIndex);
    }
}
```

- Path Complexity: 4
Complexity Classes

Classify path complexities as **constant**, polynomial, or exponential.

Examples from Java SDK 7.

```java
private static void rangeCheck(int length, int fromIndex, int toIndex) {
    if (fromIndex > toIndex) {
        throw new IllegalArgumentException(
            "fromIndex(" + fromIndex + ") >
            toIndex(" + toIndex + ")");
    }
    if (fromIndex < 0) {
        throw new ArrayIndexOutOfBoundsException(fromIndex);
    }
    if (toIndex > length) {
        throw new ArrayIndexOutOfBoundsException(toIndex);
    }
}
```

- **Path Complexity**: 4
- **Asymptotic**: $\Theta(1)$
- **Complexity Class**: Constant
Complexity Classes

Classify path complexities as constant, **polynomial**, or exponential.

Examples from Java SDK 7.

```java
public Matcher reset() {
    first = -1;
    last = 0;
    oldLast = -1;
    for(int i=0; i<groups.length; i++)
        groups[i] = -1;
    for(int i=0; i<locals.length; i++)
        locals[i] = -1;
    lastAppendPosition = 0;
    from = 0;
    to = getTextLength();
    return this;
}
```
Complexity Classes

Classify path complexities as constant, **polynomial**, or exponential.

Examples from Java SDK 7.

```java
public Matcher reset() {
    first = -1;
    last = 0;
    oldLast = -1;
    for(int i=0; i<groups.length; i++)
        groups[i] = -1;
    for(int i=0; i<locals.length; i++)
        locals[i] = -1;
    lastAppendPosition = 0;
    from = 0;
    to = getTextLength();
    return this;
}
```

▶ Path Complexity: $0.12n^2 + 1.25n + 3$
Complexity Classes

Classify path complexities as constant, **polynomial**, or exponential.

Examples from Java SDK 7.

```java
public Matcher reset() {
    first = -1;
    last = 0;
    oldLast = -1;
    for(int i=0; i<groups.length; i++)
        groups[i] = -1;
    for(int i=0; i<locals.length; i++)
        locals[i] = -1;
    lastAppendPosition = 0;
    from = 0;
    to = getTextLength();
    return this;
}
```

- Path Complexity: $0.12n^2 + 1.25n + 3$
- Asymptotic: $\Theta(n^2)$
- Complexity Class: Polynomial
Complexity Classes
Classify path complexities as constant, polynomial, or exponential.

Examples from Java SDK 7.

```java
private static int binarySearch0(long[] a,
        int fromIndex, int toIndex, long key) {
    int low = fromIndex;
    int high = toIndex - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        long midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found.
}
```
Complexity Classes

Classify path complexities as constant, polynomial, or exponential.

Examples from Java SDK 7.

private static int binarySearch0(long[] a, int fromIndex, int toIndex, long key) {
    int low = fromIndex;
    int high = toIndex - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        long midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found.
}

▶ Path Complexity: $(6.86)(1.17)^n + (0.22)(1.1)^n + (0.13)(0.84)^n + 2$
Complexity Classes

Classify path complexities as constant, polynomial, or exponential.

Examples from Java SDK 7.

```java
private static int binarySearch0(long[] a,
        int fromIndex, int toIndex, long key) {
    int low = fromIndex;
    int high = toIndex - 1;
    while (low <= high) {
        int mid = (low + high) >>> 1;
        long midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found.
}
```

- Path Complexity: $(6.86)(1.17)^n + (0.22)(1.1)^n + (0.13)(0.84)^n + 2$
- Asymptotic: $\Theta(1.17^n)$
- Complexity Class: Exponential
Other Complexity Measures

- Cyclomatic complexity: the maximum number of linearly independent paths in the CFG.
- NPATH Complexity: the number of acyclic paths in the CFG.

Limitation: Both cyclomatic and NPATH return constant numbers, regardless of loops.

Comparison of cyclomatic, NPATH, and path complexities:

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclomatic</th>
<th>NPATH</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymptotic Complexity</td>
<td>Θ(1)</td>
<td>Θ(n^2)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>rangeCheck()</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>reset()</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>binarySearch0()</td>
<td>4</td>
<td>4</td>
<td>(6.86)</td>
</tr>
<tr>
<td></td>
<td>(1.17) n + (0.22)</td>
<td></td>
<td>(1.17) n + (0.13) (0.84)</td>
</tr>
</tbody>
</table>

*The complexity notation Θ(*) represents the asymptotic behavior of the function, indicating the upper and lower bounds of the function's growth rate.*
Other Complexity Measures

- **Cyclomatic complexity:** the maximum number of linearly independent paths in the CFG.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclomatic</th>
<th>NPATH Complexity</th>
<th>Path Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>rangeCheck()</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>reset()</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>binarySearch0()</td>
<td>4</td>
<td>4</td>
<td>1.17n + (0.13n + 2)</td>
</tr>
</tbody>
</table>
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
- **NPATH Complexity**: the number of acyclic paths in the CFG.
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
- **NPATH Complexity**: the number of acyclic paths in the CFG.
- Limitation: Both cyclomatic and NPATH return constant numbers, regardless of loops.
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
- **NPATH Complexity**: the number of acyclic paths in the CFG.
- Limitation: Both cyclomatic and NPATH return constant numbers, regardless of loops.
- Comparison of cyclomatic, NPATH, and path complexities.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclomatic Complexity</th>
<th>NPATH Complexity</th>
<th>Path Complexity</th>
<th>Asymptotic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>rangeCheck()</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>reset()</td>
<td>3</td>
<td>4</td>
<td>$0.12n^2 + 1.25n + 3$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>binarySearch0()</td>
<td>4</td>
<td>4</td>
<td>$(6.86)1.17^n + (0.22)1.1^n + (0.13)(0.84)^n + 2$</td>
<td>$\Theta(1.17^n)$</td>
</tr>
</tbody>
</table>
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
- **NPATH Complexity**: the number of acyclic paths in the CFG.
- Limitation: Both cyclomatic and NPATH return constant numbers, regardless of loops.
- Comparison of cyclomatic, NPATH, and path complexities.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclomatic Complexity</th>
<th>NPATH Complexity</th>
<th>Path Complexity</th>
<th>Asymptotic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>rangeCheck()</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>reset()</td>
<td>3</td>
<td>4</td>
<td>$0.12n^2 + 1.25n + 3$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>binarySearch0()</td>
<td>4</td>
<td>4</td>
<td>$(6.86)1.17^n + (0.22)1.1^n + (0.13)(0.84)^n + 2$</td>
<td>$\Theta(1.17^n)$</td>
</tr>
</tbody>
</table>
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.

- **NPATH Complexity**: the number of acyclic paths in the CFG.

- Limitation: Both cyclomatic and NPATH return constant numbers, regardless of loops.

- Comparison of cyclomatic, NPATH, and path complexities.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclomatic Complexity</th>
<th>NPATH Complexity</th>
<th>Path Complexity</th>
<th>Asymptotic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>rangeCheck()</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>reset()</td>
<td>3</td>
<td>4</td>
<td>$0.12n^2 + 1.25n + 3$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>binarySearch0()</td>
<td>4</td>
<td>4</td>
<td>$(6.86)1.17^n + (0.22)1.1^n + (0.13)(0.84)^n + 2$</td>
<td>$\Theta(1.17^n)$</td>
</tr>
</tbody>
</table>
Experiments

- Tested our analysis on Java 7 SDK (132K methods, ≈ 2.5 hr.) and Apache Commons (44K methods, ≈ 1 hr.) libraries.
- Separated methods into complexity classes:
  - $C = 1$ Unique path
  - $C > 1$ Constant number of paths
  - $n^k$ Polynomial
  - $b^n$ Exponential
Experiments

- Tested our analysis on Java 7 SDK (132K methods, ≈ 2.5 hr.) and Apache Commons (44K methods, ≈ 1 hr.) libraries.
- Separated methods into complexity classes:
  - $C = 1$  Unique path
  - $C > 1$  Constant number of paths
  - $n^k$  Polynomial
  - $b^n$  Exponential

### Java 7 SDK
- $C = 1$: 60.0%
- $C > 1$: 30.1%
- $n^k$: 5.3%
- $b^n$: 4.6%

### Apache Commons
- $C = 1$: 60.8%
- $C > 1$: 27.0%
- $n^k$: 5.5%
- $b^n$: 6.7%
Our tool is called PAth Complexity Analyzer (PAC).

- [vlab.cs.ucsb.edu/PAC/](vlab.cs.ucsb.edu/PAC/)
Our tool is called PAtlh Complexity Analyzer (PAC).

- vlab.cs.ucsb.edu/PAC/
- Implemented using ASM Framework and MATHEMATICA.
Our tool is called PAth Complexity Analyzer (PAC).

- [vlab.cs.ucsb.edu/PAC/](vlab.cs.ucsb.edu/PAC/)
- Implemented using ASM Framework and MATHEMATICA.
  - Source code and experimental results are available.
Our tool is called PAth Complexity Analyzer (PAC).

- vlab.cs.ucsb.edu/PAC/
- Implemented using ASM Framework and MATHEMATICA.
  - Source code and experimental results are available.
- Web version.
  1. Upload Java .class or .jar file.
  2. Output a table of cyclomatic, NPATH, and (asymptotic) path complexities for all methods.
Future Work

- Case study: experimentally validate that path complexity is a good measure of the difficulty of achieving path coverage.
- Extend analysis to inter-procedural calls using the theory of generating functions for generative grammars.
- Path complexity may count infeasible paths–provides only an upper bound. Refine path complexity with predicate abstraction to consider path conditions.
- Apply path complexity results to side-channel analysis for timing attacks: path length $\approx$ execution time side channel.
Thank you.