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

ESEC FSE 2015
Overview: What did we do?
Overview: What did we do?
Overview: What did we do?

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

Program

JAVA

PATH
Complexity
Analyzer
(PAC)
Overview: What did we do?

Program → PAth 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 Length Bound, $n$

Path Counting Function, $path(n)$

Number of paths within length $n$
Overview: What did we do?

Program

Path Complexity Analyzer (PAC)

Path Length Bound, \( n \)

Counting Function, \( path(n) \)

Number of paths within length \( n \)

Asymptotic Behavior
\[ path(n) = \Theta(f(n)) \]
Can you solve it, Will Hunting?
Can you solve it, Will Hunting?

- **G** is the graph

  - Find i) the adjacency matrix $A$

  - 2) the matrix giving the number of 3 step walks

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

  - 4) the generating function for walks from points $1 \rightarrow 3$
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

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

```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

- $\text{count}(n)$ be the number of paths of length exactly $n$. 

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

- $\textit{count}(n)$ 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 $\textit{count}(n)$. 
Path Complexity

```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

- \( count(n) \) 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 \( path(n) \).
Path Complexity

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

<table>
<thead>
<tr>
<th></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;
}

### Path Complexity

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 $path(n)$.

<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>

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

```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

- $\text{count}(n)$ 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

```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

- \( \text{count}(n) \) 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 \( \text{count}(n) \).
- 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

```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 $path(n)$. 

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 $\textit{path}(n)$.
boolean passCheck1()
{
    while (i < n)
    {
        if (p[i] != pass[i])
            return false;
        i++;
    } // i is now greater than n, so we can exit the loop
    return true;
}
boolean passCheck1()
    {  
        while (i < n)  
            {  
            if (p[i] != pass[i])  
                return false;  
                i++;  
        }  
        return true;  
    }

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
n    & count(n) & path(n) \\
\hline
20    & 1        & 14  \\
21    & 0        & 15  \\
22    & 1        & 15  \\
23    & 1        & 16  \\
24    & 0        & 16  \\
25    & 1        & 17  \\
26    & 1        & 18  \\
\hline
\end{tabular}
\end{table}

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

Also appears to be linear...
boolean passCheck2() {
    matched = true;
    while (i < n) {
        if (p[i] != pass[i])
            matched = false;
        i++;
    }
    return matched;
}

Also appears to be linear...

<table>
<thead>
<tr>
<th>n</th>
<th>count(n)</th>
<th>path(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>11</td>
<td>69</td>
</tr>
<tr>
<td>21</td>
<td>16</td>
<td>85</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>106</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>128</td>
</tr>
<tr>
<td>24</td>
<td>27</td>
<td>155</td>
</tr>
<tr>
<td>25</td>
<td>37</td>
<td>192</td>
</tr>
<tr>
<td>26</td>
<td>43</td>
<td>235</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;
}

Also appears to be linear... or is it?

<table>
<thead>
<tr>
<th>n</th>
<th>count(n)</th>
<th>path(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>11</td>
<td>69</td>
</tr>
<tr>
<td>21</td>
<td>16</td>
<td>85</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>106</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>128</td>
</tr>
<tr>
<td>24</td>
<td>27</td>
<td>155</td>
</tr>
<tr>
<td>25</td>
<td>37</td>
<td>192</td>
</tr>
<tr>
<td>26</td>
<td>43</td>
<td>235</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;
}

Also appears to be linear...or is it?
Could be polynomial or exponential.

<table>
<thead>
<tr>
<th>n</th>
<th>count(n)</th>
<th>path(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>11</td>
<td>69</td>
</tr>
<tr>
<td>21</td>
<td>16</td>
<td>85</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>106</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>128</td>
</tr>
<tr>
<td>24</td>
<td>27</td>
<td>155</td>
</tr>
<tr>
<td>25</td>
<td>37</td>
<td>192</td>
</tr>
<tr>
<td>26</td>
<td>43</td>
<td>235</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Computing Path Complexity

The path complexity problem:
Computing Path Complexity

The path complexity problem:

- How to compute $\text{path}(n)$ automatically?
The path complexity problem:
▶ How to compute $\text{path}(n)$ automatically?
▶ What is the asymptotic behavior of $\text{path}(n)$?
Matrix Exponentiation

For a particular \( n \), 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.

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

\[
A_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

\[
A_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}
\]

\[
A_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

\[
A_4 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}
\]

\[
\text{path}(1) = 1, \quad \text{path}(2) = 2, \quad \text{path}(3) = 2, \quad \text{path}(4) = 3.
\]

Drawback: repeated evaluations become expensive.
Matrix Exponentiation

For a particular $n$, 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.

$\text{path}(1) = 1$
$\text{path}(2) = 2$
$\text{path}(3) = 2$
$\text{path}(4) = 3$

Drawback: repeated evaluations become expensive.
Matrix Exponentiation

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

- For a particular $n$, 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

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

![Diagram]

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

- For a particular $n$, we can compute $path(n)$ using the $p \times p$ adjacency matrix, $A$, of the CFG, augmented with an additional 1 entry in the final column and final row.

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

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

\[
A^2 = \begin{bmatrix}
0 & 0 & 1 & 2 \\
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Matrix Exponentiation

- For a particular $n$, 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\[1\] = \[
\begin{bmatrix}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

A\[2\] = \[
\begin{bmatrix}
0 & 0 & 1 & 2 \\
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

A\[3\] = \[
\begin{bmatrix}
1 & 0 & 0 & 2 \\
0 & 1 & 0 & 2 \\
0 & 0 & 1 & 2 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Drawback: repeated evaluations become expensive.
Matrix Exponentiation

- For a particular \( n \), 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.

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

\[
A^1 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^2 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^3 = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^4 = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 3 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]
Matrix Exponentiation

- For a particular $n$, 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.

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

$$
A^1 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
A^2 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
A^3 = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
A^4 = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 3 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$\text{path}(1) = 1 \\
\text{path}(2) = 2 \\
\text{path}(3) = 2 \\
\text{path}(4) = 3$
Matrix Exponentiation

- For a particular $n$, 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.

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

Drawback: repeated evaluations become expensive.
Matrix exponentiation works. Is there a better way?
Generating Functions
Generating Functions

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

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

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

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

- The $n$th Taylor series coefficient of $g(z)$ is given by

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

- 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. \(\text{path}(n) = 0, 1, 2, 2, 3, 4, 4, \ldots\))
- The generating function for counting paths in a graph is given by
  \[
g(z) = (-1)^{m+1} \frac{\det(\mathbb{1} - zA : m, 1)}{\det(\mathbb{1} - zA)}
  \]
- 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. \( path(n) = 0, 1, 2, 2, 3, 4, 4, \ldots \))
- 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 \( g(z) \).

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

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

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

- 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 \( n^{\text{th}} \) 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!

This is correct. Who did this?
Good job, Will Hunting!
Good job, Will Hunting!

\[ \sum_{h=0}^{18} \omega_n(i \rightarrow j) = n \]

\[ \det (I - zA) \]

\[ \det (I - z \mathbf{A}) \]

(\( p_i \rightarrow P(i-j) \))
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 \( r \) roots of the denominator

\[ (1 - z)(1 - z^3) = 0 \implies z = 1, \]
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 \( r \) roots of the denominator

\[
(1 - z)(1 - z^3) = 0 \implies z = 1, 1,
\]
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 \( r \) roots of the denominator

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

\[
(1 - z)(1 - z^3) = 0 \quad \Rightarrow \quad z = 1, 1, \frac{-1 + \sqrt{3}i}{2}, \frac{-1 - \sqrt{3}i}{2}
\]
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 \( r \) roots of the denominator

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

- Take a linearly independent combination of exponentiated roots:
**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 \( r \) roots of the denominator

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

- Take a linearly independent combination of exponentiated roots:

\[ path(n) = c_1 \cdot 1^n \]
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 \( r \) roots of the denominator

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

Take a linearly independent combination of exponentiated roots:

\[ \text{path}(n) = c_1 \cdot 1^n + c_2 n \cdot 1^n + \]
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 \( r \) roots of the denominator

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

- Take a linearly independent combination of exponentiated roots:

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

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

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

- Take a linearly independent combination of exponentiated roots:

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

- Solve for coefficients \( c_1, \ldots, c_r \) using \( g(z), g'(z), \ldots, g^{(r)}(z) \)

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

Our solution looks very...
Tight bounds for \( \text{path}(n) \)

Our solution looks very... complex

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

Now, it looks much simpler:

\[
2 \frac{n}{3} \leq \text{path}(n) \leq 2 \frac{n}{3} + 2 \frac{3}{3}
\]
Tight bounds for \textit{path}(n)

Our solution looks very... complex

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

\[\leq \]

- 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 \( path(n) \)
Asymptotic Behavior

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

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

- Applied to our examples:
  - Function passCheck1()
    \[ \text{path}(n) = \Theta(n) \]
  - Function passCheck2()
    \[ \text{path}(n) = \Theta(1.221n) \]
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()`

  \[ \text{path}(n) = \Theta(n) \]

  Function `passCheck2()`

  \[ \text{path}(n) = \Theta(1.221^n) \]
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()`
    
    \[ path(n) = \Theta(n) \]
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(\n            + 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);
  }
}
```
Complexity Classes

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

Examples from Java SDK 7.

```
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 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;
}
```
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 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.
}
```

Path Complexity: $n + (0.22)n + (0.13)(0.84)n + 2$

Asymptotic: $\Theta(1.17n)$

Complexity Class: Exponential
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.

```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: \((6.86)(1.17)^n + (0.22)(1.1)^n + (0.13)(0.84)^n + 2\)
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 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.
- A set of paths is linearly independent if and only if each path contains at least one edge that is not included in any other path.
- 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 Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>rangeCheck()</td>
<td>4</td>
<td>4</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>reset()</td>
<td>3</td>
<td>4</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>binarySearch0()</td>
<td>4</td>
<td>4</td>
<td>$(6.86, 1.17, n + (0.22, 1.1))$</td>
</tr>
</tbody>
</table>

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

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
  - A set of paths is linearly independent if and only if each path contains at least one edge that is not included in any other path.
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
  
  - A set of paths is linearly independent if and only if each path contains at least one edge that is not included in any other path.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclomatic</th>
<th>NPATH</th>
<th>Path</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>0.12n</td>
</tr>
<tr>
<td>binarySearch()</td>
<td>4</td>
<td>4</td>
<td>(6.86n + 0.22)</td>
</tr>
</tbody>
</table>

Asymptotic Complexity

- \(\Theta(1)\)
- \(\Theta(n^2)\)
- \(\Theta(1.17n) + (0.13)(0.84n^2)\)
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
  - A set of paths is linearly independent if and only if each path contains at least one edge that is not included in any other path.

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

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

<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>(6.86, 1.17n)</td>
</tr>
</tbody>
</table>
Other Complexity Measures

- **Cyclomatic complexity**: the maximum number of linearly independent paths in the CFG.
  - A set of paths is linearly independent if and only if each path contains at least one edge that is not included in any other path.

- **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>
# Complexity Comparison

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Control Flow Graph</th>
<th>Cyclomatic Complexity</th>
<th>NPATH Complexity</th>
<th>Asymptotic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$ If-Else in sequence</td>
<td><img src="sequence_diagram.png" alt="Sequence Diagram" /></td>
<td>$K + 1$</td>
<td>$2^K$</td>
<td>$2^K$</td>
</tr>
<tr>
<td>$K$ If-Else nested</td>
<td><img src="nested_diagram.png" alt="Nested Diagram" /></td>
<td>$K + 1$</td>
<td>$K + 1$</td>
<td>$K + 1$</td>
</tr>
<tr>
<td>$K$ Loops in sequence</td>
<td><img src="loop_sequence_diagram.png" alt="Loop Sequence Diagram" /></td>
<td>$K + 1$</td>
<td>$2^K$</td>
<td>$\Theta(n^K)$</td>
</tr>
<tr>
<td>$K$ Loops nested</td>
<td><img src="loop_nested_diagram.png" alt="Loop Nested Diagram" /></td>
<td>$K + 1$</td>
<td>$K + 1$</td>
<td>$\Theta(b^n)$</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%
- $n^k$: 4.6%
- $b^n$: 5.3%

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

- vlab.cs.ucsb.edu/PAC/
Our tool is called PAth 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/](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

- 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 to consider simple path conditions.
- Apply path complexity results to side-channel analysis for timing attacks.
Thank you.