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

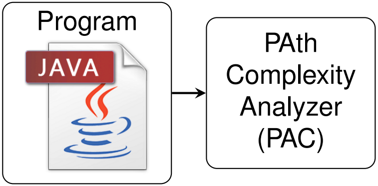
Overview: What did we do?

Overview: What did we do?

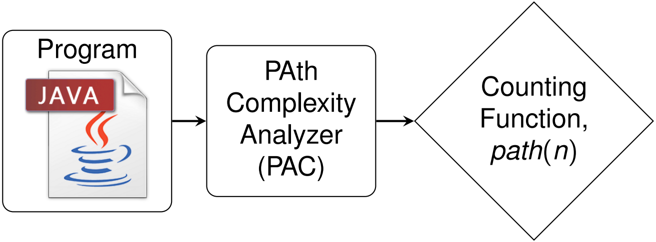
Overview: What did we do?

PAth
Complexity
Analyzer
(PAC)

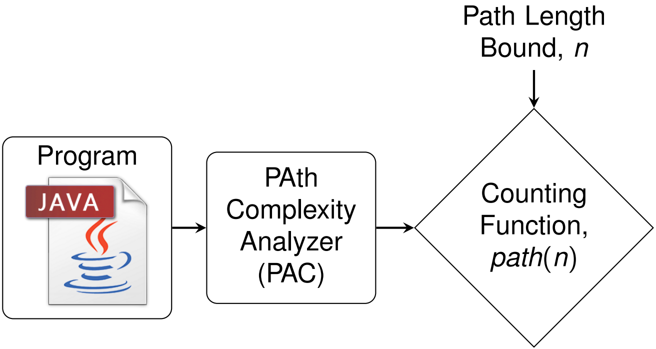
Overview: What did we do?



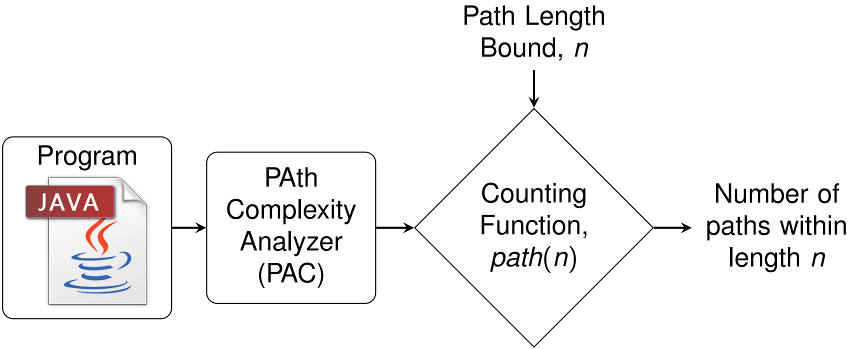
Overview: What did we do?



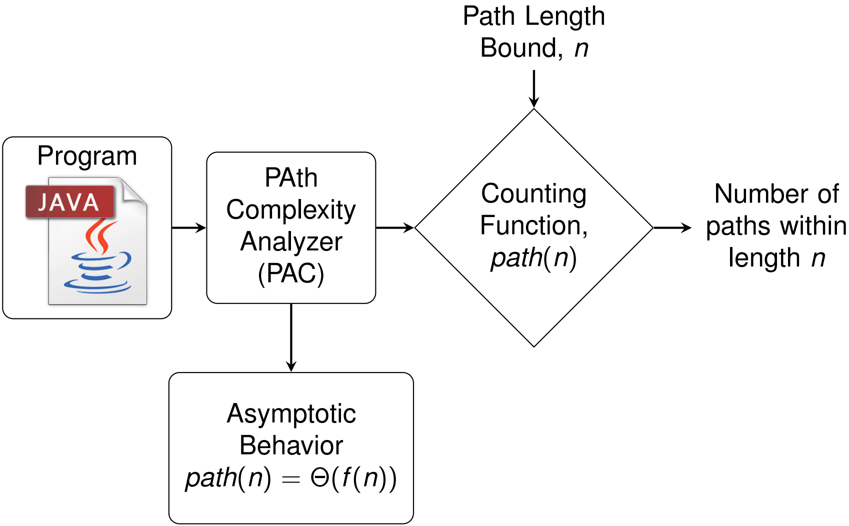
Overview: What did we do?



Overview: What did we do?

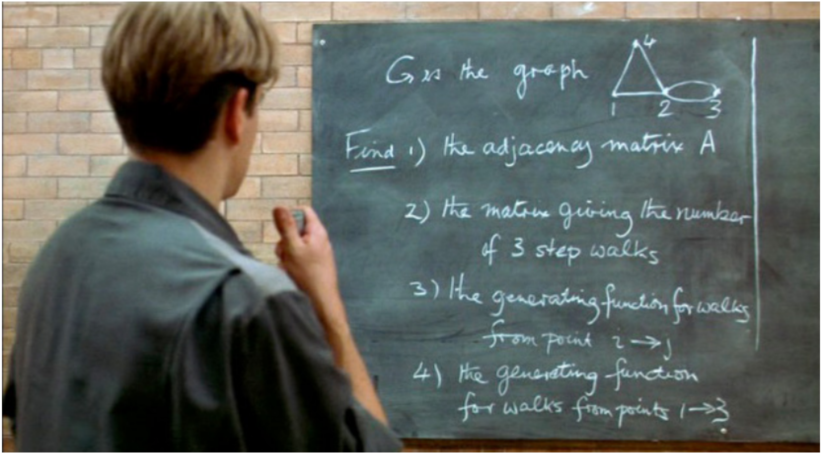


Overview: What did we do?



Can you solve it, Will Hunting?

Can you solve it, Will Hunting?



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

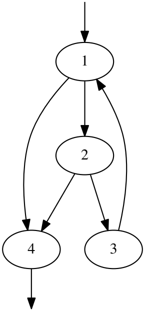
- Modern automated software testing techniques focus on program path coverage.
- The number of execution paths could be infinite.
- Practical solution: explore up to a given depth bound.
- We propose a metric, the **path complexity**, an upper bound on the number of paths needed to explore up to a given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

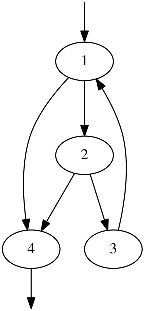
Path Complexity

```
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;
}
```

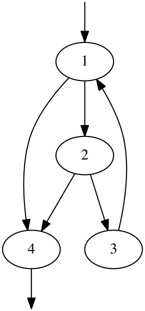


Given a control flow graph and a length bound n, let

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

Path Complexity

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

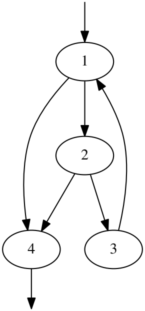


Given a control flow graph and a length bound *n*, let

- *count*(*n*) be the number of paths of length **exactly** *n*.
- *path*(*n*) be the number of paths of length **less than or equal** n, i.e. the accumulated sum of *count*(*n*).

Path Complexity

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

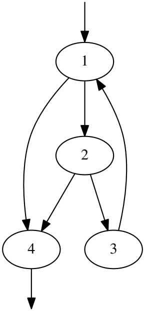


Given a control flow graph and a length bound n, let

- *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;
}
```



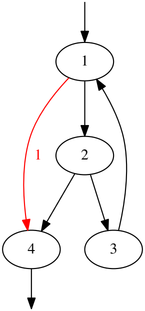
n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

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;
}
```



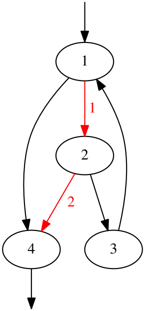
n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

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;
}
```



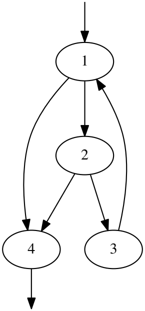
n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

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;
}
```



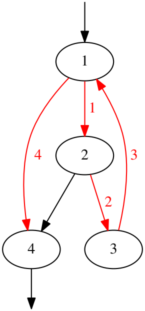
n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

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;
}
```



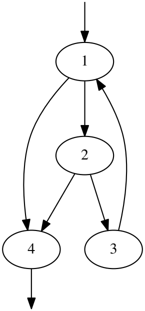
n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

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;
}
```



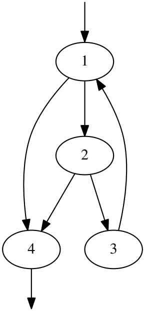
n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

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;
}
```



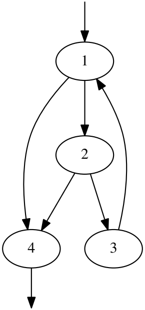
n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

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*).

Computing Path Complexity

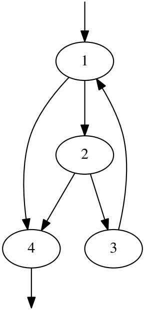
```
boolean passCheck1(){
 while(i<n){
  if(p[i] != pass[i])
   return false;
  i++;
 }
return true;
}
```



n	*count*(n)	*path*(n)
0	0	0
1	1	1
2	1	2
3	0	2
4	1	3
5	1	4
6	0	4
⋮	⋮	⋮

Computing Path Complexity

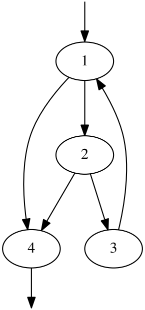
```
boolean passCheck1(){
 while(i<n){
  if(p[i] != pass[i])
   return false;
  i++;
 }
return true;
}
```



n	*count*(*n*)	*path*(*n*)
20	1	14
21	0	15
22	1	15
23	1	16
24	0	16
25	1	17
26	1	18
⋮	⋮	⋮

Computing Path Complexity

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



n	*count*(n)	*path*(n)
20	1	14
21	0	15
22	1	15
23	1	16
24	0	16
25	1	17
26	1	18
⋮	⋮	⋮

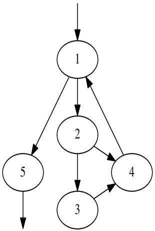
Appears to grow linearly... is it $\frac{2}{3}n$?

Computing Path Complexity

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

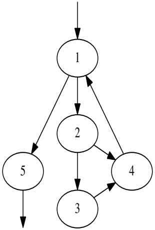
Computing Path Complexity

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



Computing Path Complexity

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

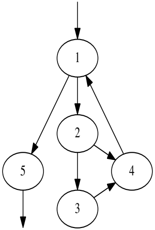


n	*count*(*n*)	*path*(*n*)
0	0	0
1	1	1
2	0	1
3	0	1
4	1	2
5	1	3
6	0	3
⋮	⋮	⋮

Also appears to be linear...

Computing Path Complexity

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

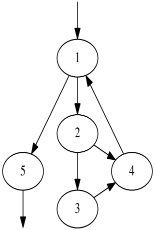


n	*count*(*n*)	*path*(*n*)
20	11	69
21	16	85
22	21	106
23	22	128
24	27	155
25	37	192
26	43	235
⋮	⋮	⋮

Also appears to be linear...

Computing Path Complexity

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

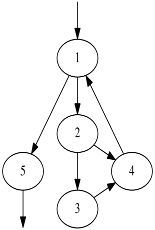


n	*count*(*n*)	*path*(*n*)
20	11	69
21	16	85
22	21	106
23	22	128
24	27	155
25	37	192
26	43	235
⋮	⋮	⋮

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

Computing Path Complexity

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



n	*count*(*n*)	*path*(*n*)
20	11	69
21	16	85
22	21	106
23	22	128
24	27	155
25	37	192
26	43	235
⋮	⋮	⋮

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

Computing Path Complexity

The path complexity problem:

Computing Path Complexity

The path complexity problem:

- How to compute *path*(n) **automatically**?

Computing Path Complexity

The path complexity problem:

- How to compute *path*(n) **automatically**?
- What is the **asymptotic behavior** of *path*(n)?

Matrix Exponentiation

Matrix Exponentiation

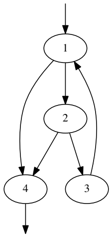
- For a particular n, we can compute *path*(n) using the $p \times p$ adjacency matrix, A, of the CFG, augmented with an additional 1 entry in the final column and final row.

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

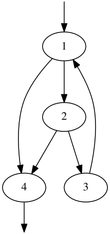
Matrix Exponentiation

- For a particular n, we can compute $path(n)$ 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

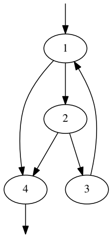
- For a particular n, we can compute $path(n)$ using the $p \times p$ adjacency matrix, A, of the CFG, augmented with an additional 1 entry in the final column and final row.
- $path(n) = (A^n)_{1,p}$



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

Matrix Exponentiation

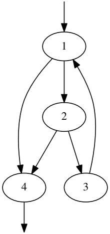
- For a particular n, we can compute $path(n)$ using the $p \times p$ adjacency matrix, A, of the CFG, augmented with an additional 1 entry in the final column and final row.
- $path(n) = (A^n)_{1,p}$



$$A^1 = \begin{bmatrix} 0 & 1 & 0 & \boxed{1} \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^2 = \begin{bmatrix} 0 & 0 & 1 & \boxed{2} \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Exponentiation

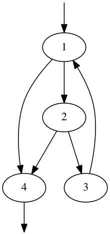
- For a particular n, we can compute $path(n)$ using the $p \times p$ adjacency matrix, A, of the CFG, augmented with an additional 1 entry in the final column and final row.
- $path(n) = (A^n)_{1,p}$



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

Matrix Exponentiation

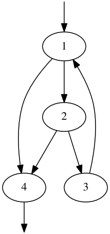
- For a particular n, we can compute $path(n)$ using the $p \times p$ adjacency matrix, A, of the CFG, augmented with an additional 1 entry in the final column and final row.
- $path(n) = (A^n)_{1,p}$



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

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

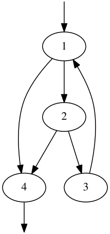


$$A^1 = \begin{bmatrix} 0 & 1 & 0 & \boxed{1} \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A^2 = \begin{bmatrix} 0 & 0 & 1 & \boxed{2} \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} A^3 = \begin{bmatrix} 1 & 0 & 0 & \boxed{2} \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\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

$path(1) = 1$ $\quad$ $path(2) = 2$ $\quad$ $path(3) = 2$ $\quad$ $path(4) = 3$

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

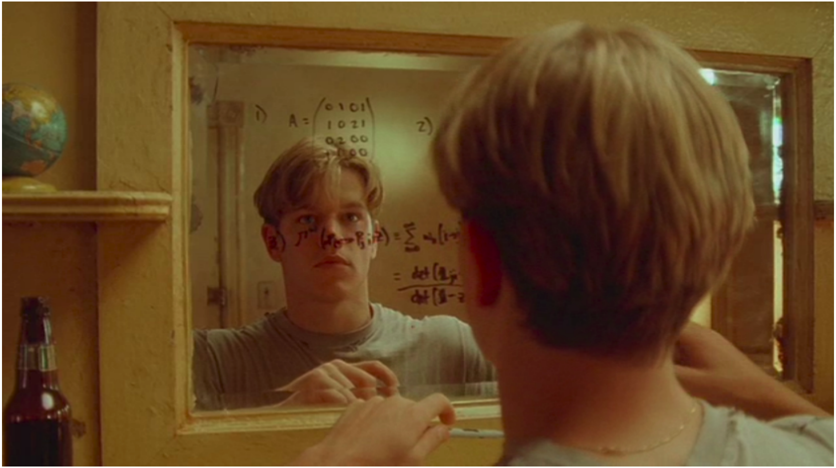


$$A^1 = \begin{bmatrix} 0 & 1 & 0 & \boxed{1} \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A^2 = \begin{bmatrix} 0 & 0 & 1 & \boxed{2} \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} A^3 = \begin{bmatrix} 1 & 0 & 0 & \boxed{2} \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\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

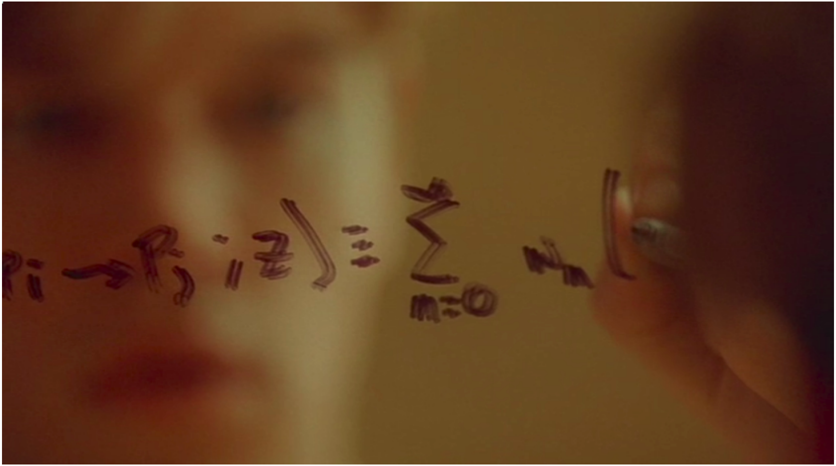
$path(1) = 1$ $\quad$ $path(2) = 2$ $\quad$ $path(3) = 2$ $\quad$ $path(4) = 3$

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.

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. $path(n) = 0, 1, 2, 2, 3, 4, 4, \ldots$)
- The generating function for counting paths in a graph is given by

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

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

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

- 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

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

- In our example CFG, the generating function is

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

- $path(n)$ is given by the n^{th} Taylor series coefficent 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

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

- In our example CFG, the generating function is

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

- $path(n)$ is given by the n^{th} Taylor series coefficent 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. $path(n) = 0, 1, 2, 2, 3, 4, 4, \ldots$)
- 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

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

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

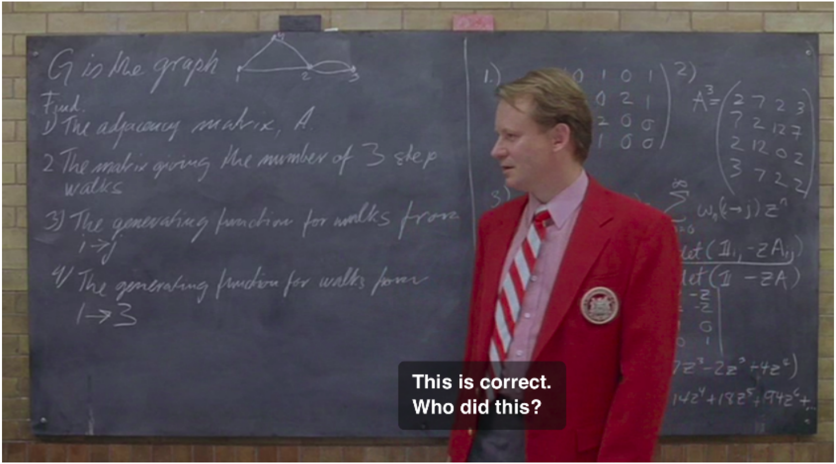
$$g(z) = \frac{g(0)}{0!} z^0 + \frac{g'(0)}{1!} z^1 + \frac{g''(0)}{2!} z^2 + \frac{g'''(0)}{3!} z^3 + \ldots$$

- For our example, the Taylor-series expansion is

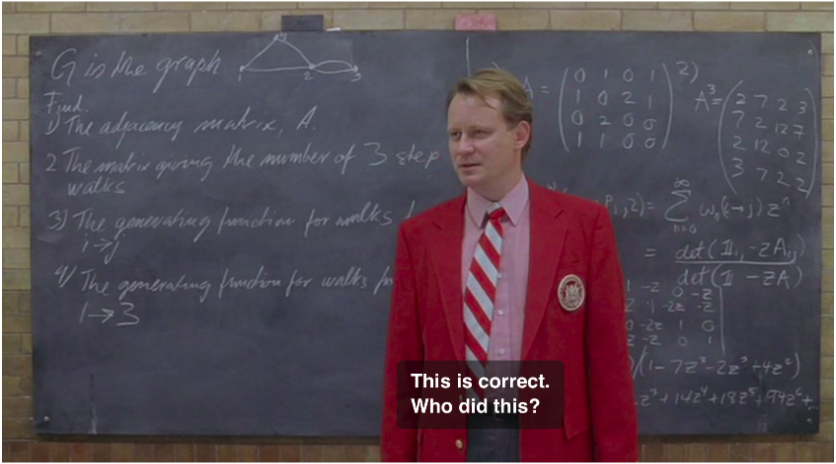
$$g(z) = 0z^0 + 1z^1 + 2z^2 + 2z^3 + 3z^4 + 4z^5 + \boxed{4}\, z^{\boxed{6}} + 5z^7 + \ldots$$

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

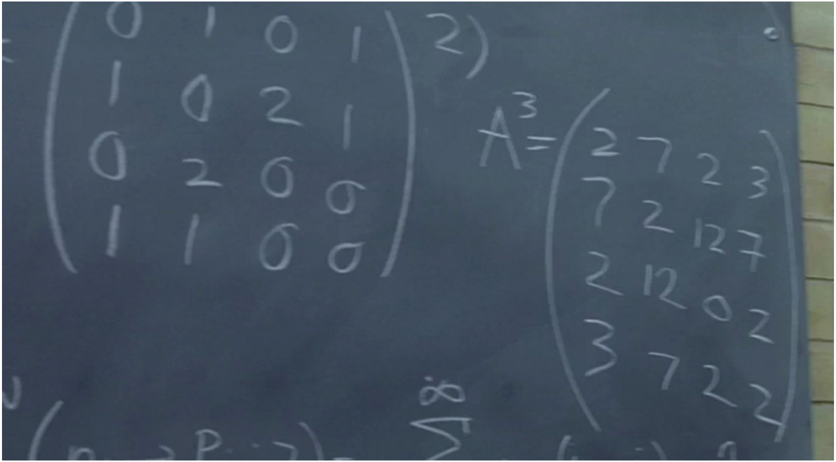
Good job, Will Hunting!



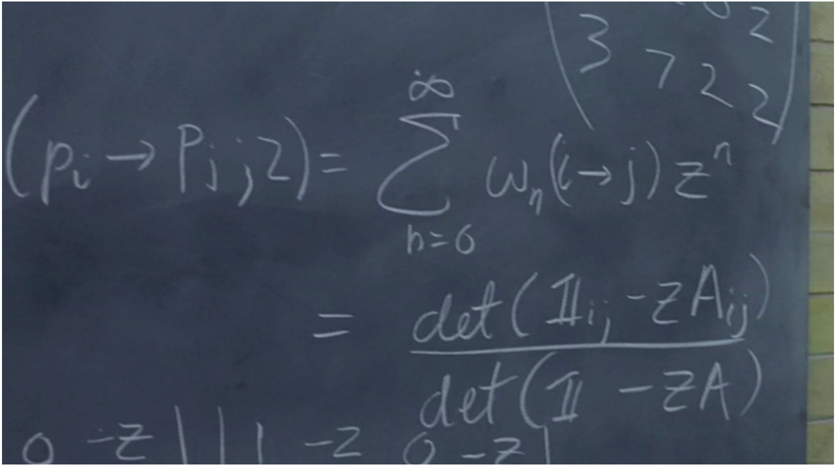
Good job, Will Hunting!



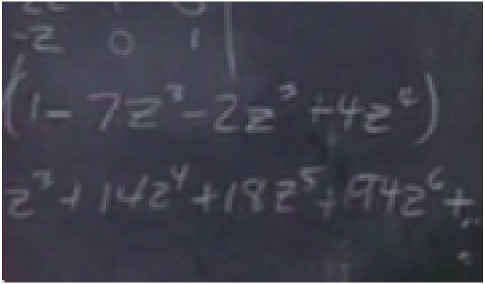
Good job, Will Hunting!



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 r roots of the denominator

$$(1-z)(1-z^3) = 0 \quad \Longrightarrow \quad z = 1,$$

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

$$(1-z)(1-z^3) = 0 \quad \Longrightarrow \quad z = 1, 1,$$

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

$$(1-z)(1-z^3) = 0 \quad \Longrightarrow \quad z = 1, 1, \frac{-1+\sqrt{3}i}{2},$$

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

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

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 r roots of the denominator

$$(1-z)(1-z^3) = 0 \quad \Longrightarrow \quad z = 1, 1, \frac{-1+\sqrt{3}i}{2}, \frac{-1-\sqrt{3}i}{2}$$

- 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

$$(1-z)(1-z^3) = 0 \quad \Longrightarrow \quad z = 1, 1, \frac{-1+\sqrt{3}i}{2}, \frac{-1-\sqrt{3}i}{2}$$

- 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:

$$path(n) = c_1 \cdot 1^n + c_2 n \cdot 1^n +$$

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

$$(1-z)(1-z^3) = 0 \quad \Longrightarrow \quad z = 1, 1, \frac{-1+\sqrt{3}i}{2}, \frac{-1-\sqrt{3}i}{2}$$

- Take a linearly independent combination of exponentiated roots:

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

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

$$(1-z)(1-z^3) = 0 \quad \Longrightarrow \quad z = 1, 1, \frac{-1+\sqrt{3}i}{2}, \frac{-1-\sqrt{3}i}{2}$$

- Take a linearly independent combination of exponentiated roots:

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

- 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 *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 *path*(n)

Our solution looks very... complex

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

- 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 *path*(n)

Our solution looks very... complex

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

- 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 *path*(n)

Our solution looks very... complex

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

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

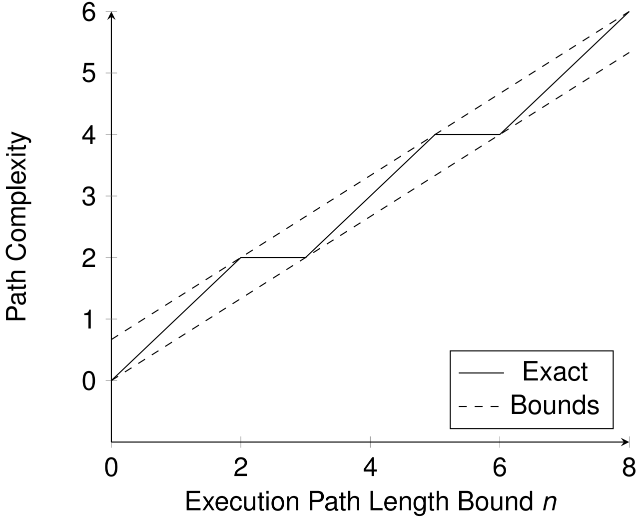
$$-2|w|^n \leq |w^n + \overline{w}^n| \leq 2|w|^n$$

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

Now, it looks much simpler:

$$\frac{2n}{3} \leq path(n) \leq \frac{2n}{3} + \frac{2}{3}$$

Tight bounds for *path*(*n*)



Asymptotic Behavior

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

$$f = \Theta(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

Asymptotic Behavior

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

$$f = \Theta(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

- Applied to our examples:

Asymptotic Behavior

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

$$f = \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

- Applied to our examples:
  - Function `passCheck1()`

$$path(n) = \Theta\left(n\right)$$

Asymptotic Behavior

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

$$f = \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

- Applied to our examples:
  - Function `passCheck1()`

$$path(n) = \Theta\left(n\right)$$

  - Function `passCheck2()`

$$path(n) = \Theta\left(1.221^n\right)$$

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.

```
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);
  }
}
```

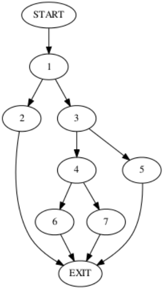
Complexity Classes

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);
  }
}
```



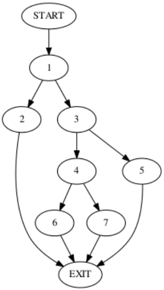
Complexity Classes

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

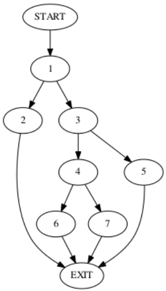
Complexity Classes

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

Examples from Java SDK 7.

```
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: $\Theta(1)$

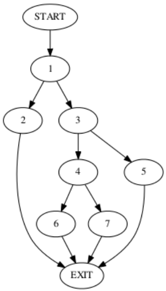
Complexity Classes

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

Examples from Java SDK 7.

```
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.

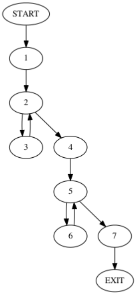
```
public Matcher reset() {
  first = -1;
  last = 0;
  oldLast = -1;
  for(int i=0; i<groups.length; i++)
    groups[i] = -1;
  for(int i=0; i<locals.length; i++)
    locals[i] = -1;
  lastAppendPosition = 0;
  from = 0;
  to = getTextLength();
  return this;
}
```

Complexity Classes

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

Examples from Java SDK 7.

```
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;
}
```

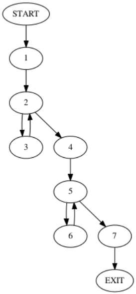


Complexity Classes

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

Examples from Java SDK 7.

```
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;
}
```



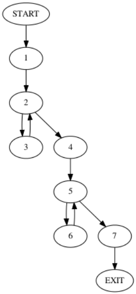
- Path Complexity: $0.12n^2 + 1.25n + 3$

Complexity Classes

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

Examples from Java SDK 7.

```
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;
}
```



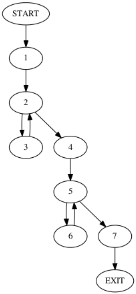
- Path Complexity: $0.12n^2 + 1.25n + 3$
- Asymptotic: $\Theta(n^2)$

Complexity Classes

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

Examples from Java SDK 7.

```
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.

```
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.
}
```

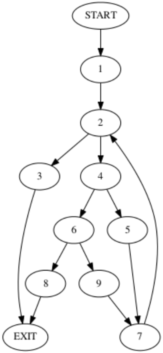
Complexity Classes

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.
}
```



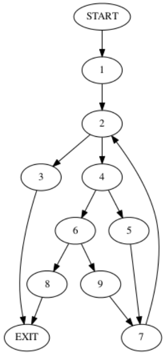
Complexity Classes

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: $(6.86)(1.17)^n + (0.22)(1.1)^n + (0.13)(0.84)^n + 2$

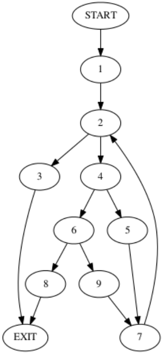
Complexity Classes

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

Examples from Java SDK 7.

```
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: $\Theta(1.17^n)$

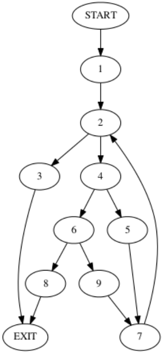
Complexity Classes

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

Examples from Java SDK 7.

```
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

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.

Other Complexity Measures

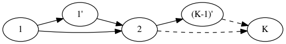
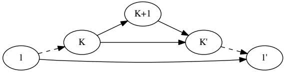
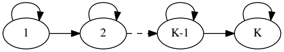
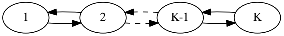
- **Cyclomatic complexity:** the maximum number of linearly independent paths in the CFG.
  - A set of paths is linearly independent if and only if each path contains at least one edge that is not included in any other path.
- **NPATH Complexity:** the number of acyclic paths in the CFG.
- Limitation: Both cyclomatic and NPATH return constant numbers, regardless of loops.

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.

Method	Cyclomatic Complexity	NPATH Complexity	Path Complexity	Asymptotic Complexity
rangeCheck()	4	4	4	$\Theta(1)$
reset()	3	4	$0.12n^2 + 1.25n + 3$	$\Theta(n^2)$
binarySearch0()	4	4	$(6.86)1.17^n + (0.22)1.1^n$ $+(0.13)(0.84)^n + 2$	$\Theta(1.17^n)$

Complexity Comparison

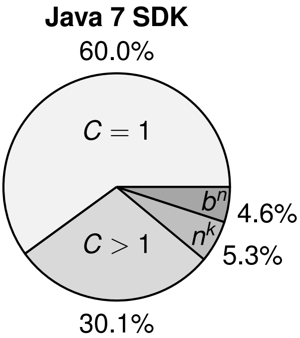
Pattern	Control Flow Graph	Cyclomatic Complexity	NPATH Complexity	Asymptotic Complexity
K If-Else in sequence		$K+1$	2^K	2^K
K If-Else nested		$K+1$	$K+1$	$K+1$
K Loops in sequence		$K+1$	2^K	$\Theta(n^K)$
K Loops nested		$K+1$	$K+1$	$\Theta(b^n)$

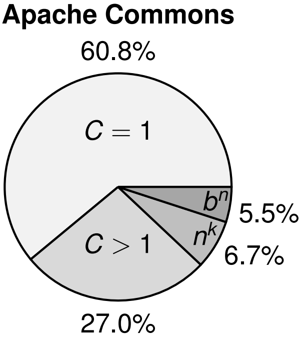
Experiments

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





Replication Package

Our tool is called PAth Complexity Analyzer (PAC).

- `vlab.cs.ucsb.edu/PAC/`

Replication Package

Our tool is called PAth Complexity Analyzer (PAC).

- `vlab.cs.ucsb.edu/PAC/`
- Implemented using ASM Framework and MATHEMATICA.

Replication Package

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.

Replication Package

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

- Experimentally validate that path complexity is a good measure of the difficulty of acheiving 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.