Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Sandeep Murthy, 29 October 2024

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications (Sandeep Murthy)

Graphs

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Graphs

- Graphs (in the sense of computer science and discrete mathematics): discrete structures made up of vertices/nodes representing objects, attributes or states of a system or process, and edges representing relations or transitions between the nodes

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Graphs

- Graphs (in the sense of computer science and discrete mathematics): discrete structures made up of vertices/nodes representing objects, attributes or states of a system or process, and edges representing relations or transitions between the nodes
- Directed graphs have directed edges, e.g. family trees, travel routes, epidemic spread graphs, network flow diagrams, dependency graphs, CI pipelines, control flow graphs (CFG) etc.

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

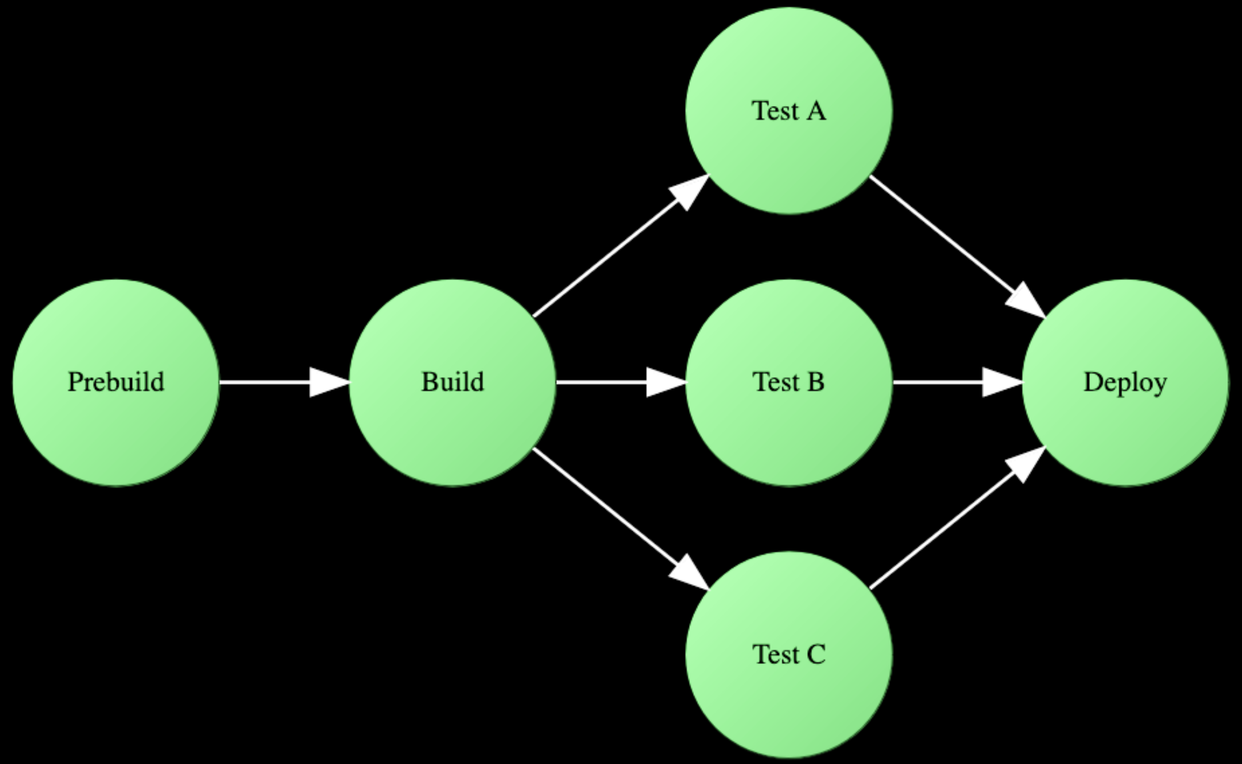
Graphs

- Graphs (in the sense of computer science and discrete mathematics): discrete structures made up of vertices/nodes representing objects, attributes or states of a system or process, and edges representing relations or transitions between the nodes
- Directed graphs have directed edges, e.g. family trees, travel routes, epidemic spread graphs, network flow diagrams, dependency graphs, CI pipelines, control flow graphs (CFG) etc.
- Directed acyclic graphs (DAG) are directed graphs with no cycles/loops/circuits, e.g. CFGs, CI pipelines

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Graphs



An example DAG of a simple CI pipeline

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Graphs

- Graphs (in the sense of computer science and discrete mathematics): discrete structures made up of vertices/nodes representing objects, attributes or states of a system or process, and edges representing relations or transitions between the nodes
- Directed graphs have directed edges, e.g. family trees, travel routes, epidemic spread graphs, network flow diagrams, dependency graphs, CI pipelines, control flow graphs (CFG) etc.
- Directed acyclic graphs (DAG) are directed graphs with no cycles/loops/circuits, e.g. CFGs, CI pipelines
- Code written in imperative languages representable as DAGs, e.g. functions/routines/procedures, classes, callables

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Graphs

- Graphs (in the sense of computer science and discrete mathematics): discrete structures made up of vertices/nodes representing objects, attributes or states of a system or process, and edges representing relations or transitions between the nodes
- Directed graphs have directed edges, e.g. family trees, travel routes, epidemic spread graphs, network flow diagrams, dependency graphs, CI pipelines, control flow graphs (CFG) etc.
- Directed acyclic graphs (DAG) are directed graphs with no cycles/loops/circuits, e.g. CFGs, CI pipelines
- Code written in imperative languages representable as DAGs, e.g. functions/routines/procedures, classes, callables
- Key #1: DAGs have a notion of start/initial/entry states and stop/final/exit states

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

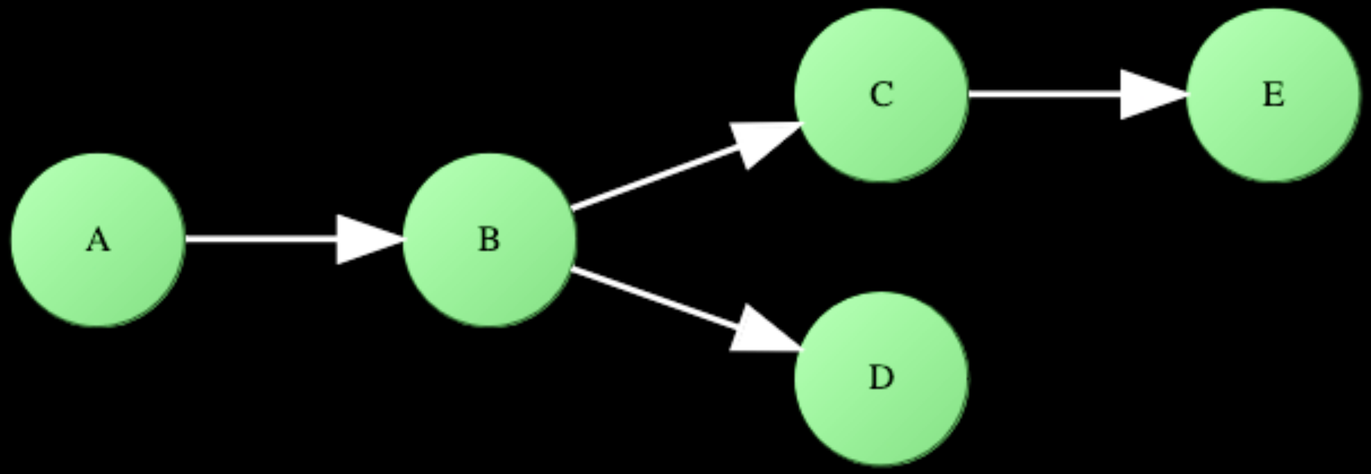
Graphs

- Graphs (in the sense of computer science and discrete mathematics): discrete structures made up of vertices/nodes representing objects, attributes or states of a system or process, and edges representing relations or transitions between the nodes
- Directed graphs have directed edges, e.g. family trees, travel routes, epidemic spread graphs, network flow diagrams, dependency graphs, CI pipelines, control flow graphs (CFG) etc.
- Directed acyclic graphs (DAG) are directed graphs with no cycles/loops/circuits, e.g. CFGs, CI pipelines
- Code written in imperative languages representable as DAGs, e.g. functions/routines/procedures, classes, callables
- Key #1: DAGs have a notion of start/initial/entry states and stop/final/exit states
- Key #2: Linearly independent (LI) DAG paths are unique ways of moving from an initial to a final state

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Graphs



An example DAG with two LI paths: A -> B -> C -> E and A -> B -> D

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Rust, Bash, PL-SQL etc.

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Rust, Bash, PL-SQL etc.
- Code (concrete, language-dependent) vs algorithm (abstract, language-independent)

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Rust, Bash, PL-SQL etc.
- Code (concrete, language-dependent) vs algorithm (abstract, language-independent)
- Source lines of code (sloc) vs logical lines of code (lloc) - usually lloc are what we refer to as statements

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Rust, Bash, PL-SQL etc.
- Code (concrete, language-dependent) vs algorithm (abstract, language-independent)
- Source lines of code (sloc) vs logical lines of code (lloc) - usually lloc are what we refer to as statements
- Levels of code: source code vs interpreted code (bytecode or assembly code) vs compiled code (binary or object code)

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

```
1   def sign(x: int | float) -> typing.Literal[-1, 0, 1]:
2       if x < 0:
3           return -1
4       if x == 0:
5           return 0
6       return 1
```

Source listing of a simple Python version of the sign function for real numbers

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

```
  1           0 RESUME                   0

  2           2 LOAD_FAST                0 (x)
              4 LOAD_CONST               1 (0)
              6 COMPARE_OP               0 (<)
             12 POP_JUMP_FORWARD_IF_FALSE     2 (to 18)

  3          14 LOAD_CONST               2 (-1)
             16 RETURN_VALUE

  4     >>   18 LOAD_FAST                0 (x)
             20 LOAD_CONST               1 (0)
             22 COMPARE_OP               2 (==)
             28 POP_JUMP_FORWARD_IF_FALSE     2 (to 34)

  5          30 LOAD_CONST               1 (0)
             32 RETURN_VALUE

  6     >>   34 LOAD_CONST               3 (1)
             36 RETURN_VALUE
```

Bytecode listing of the Python implementation of the sign function for real numbers

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

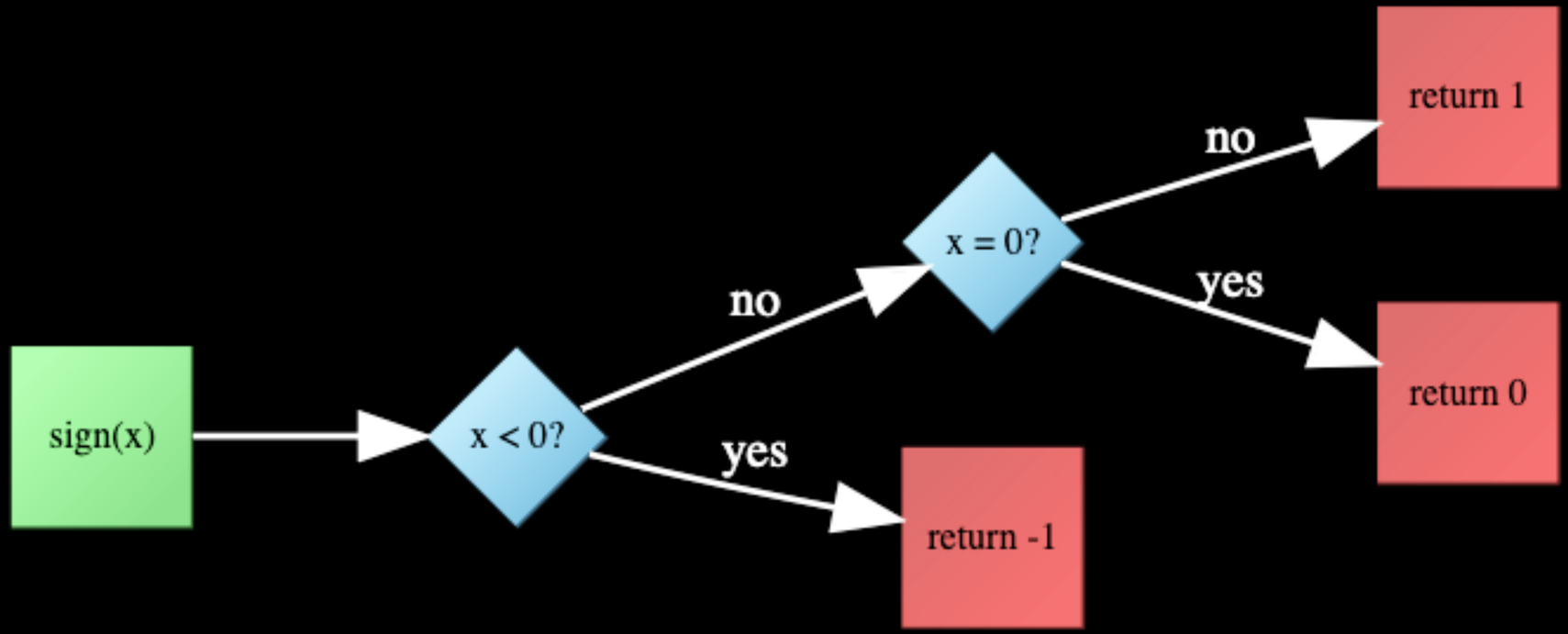
```
b'\x97\x00|\x00d\x01k\x00\x00\x00\x00\x00r\x02d\x02S\x00|
\x00d\x01k\x02\x00\x00\x00\x00r\x02d\x01S\x00d\x03S\x00'
```

Object code listing of the Python implementation of the sign function for real numbers

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code



DAG for the Python sign function

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Bash, Rust, PL-SQL etc.
- Code (concrete, language-dependent) vs algorithm (abstract, language-independent)
- Source lines of code (sloc) vs logical lines of code (lloc) - usually lloc are what we refer to as statements
- Levels of code: source code vs interpreted code (bytecode or assembly code) vs compiled code (binary or object code)
- Imperative vs declarative: control flow vs no control flow => naturally modelled as DAGs/CFGs where nodes represent statements/instructions/byte sequences and edges represent transitions between them

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Rust, Bash, PL-SQL etc.
- Code (concrete, language-dependent) vs algorithm (abstract, language-independent)
- Source lines of code (sloc) vs logical lines of code (lloc) - usually lloc are what we refer to as statements
- Levels of code: source code vs interpreted code (bytecode or assembly code) vs compiled code (binary or object code)
- Imperative vs declarative: control flow vs no control flow => naturally modelled as DAGs/CFGs where nodes represent statements/instructions/byte sequences and edges represent transitions between them
- Key #1: Initial states are called entry points, decision/branching statements represent potential transitions between states and are called decision/branching points, final states are terminal and are called exit points

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Rust, Bash, PL-SQL etc.
- Code (concrete, language-dependent) vs algorithm (abstract, language-independent)
- Source lines of code (sloc) vs logical lines of code (lloc) - usually lloc are what we refer to as statements
- Levels of code: source code vs interpreted code (bytecode or assembly code) vs compiled code (binary or object code)
- Imperative vs declarative: control flow vs no control flow => naturally modelled as DAGs/CFGs where nodes represent statements/instructions/byte sequences and edges represent transitions between them
- Key #1: Initial states are called entry points, decision/branching statements represent potential transitions between states and are called decision/branching points, final states are terminal and are called exit points

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Code

- “Code" (or programs) refers to a coded representation of an algorithm or computational process written in a programming language, e.g. BASIC, C, C++, Prolog, LISP, Java, Python, JavaScript, C#, Rust, Bash, PL-SQL etc.
- Code (concrete, language-dependent) vs algorithm (abstract, language-independent)
- Source lines of code (sloc) vs logical lines of code (lloc) - usually lloc are what we refer to as statements
- Levels of code: source code vs interpreted code (bytecode or assembly code) vs compiled code (binary or object code)
- Imperative vs declarative: control flow vs no control flow => naturally modelled as DAGs/CFGs where nodes represent statements/instructions/byte sequences and edges represent transitions between them
- Key #1: Initial states are called entry points, decision/branching statements represent potential transitions between states and are called decision/branching points, final states are terminal and are called exit points
- Key #2: A linearly independent (LI) CFG path is a unique way of going from an entry point to an exit point

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Cyclomatic Complexity

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Cyclomatic Complexity

- General notion of “complexity”: diversity (or variability) and predictability of the modes of behaviour of a system or process, where higher complexity is usually associated with greater diversity and unpredictability of the modes of behaviour

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Cyclomatic Complexity

- General notion of “complexity”: diversity (or variability) and predictability of the modes of behaviour of a system or process, where higher complexity is usually associated with greater diversity and unpredictability of the modes of behaviour
- Complexity “emerges” from the diversity of the system parts and their interrelations and interactions

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Cyclomatic Complexity

- General notion of “complexity”: diversity (or variability) and predictability of the modes of behaviour of a system or process, where higher complexity is usually associated with greater diversity and unpredictability of the modes of behaviour
- Complexity “emerges” from the diversity of the system parts and their interrelations and interactions
- For (imperative) code one approach to understanding complexity is **cyclomatic**: counting the number of LI CFG paths, with a high number of LI paths indicating high complexity

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Cyclomatic Complexity

- General notion of “complexity”: diversity (or variability) and predictability of the modes of behaviour of a system or process, where higher complexity is usually associated with greater diversity and unpredictability of the modes of behaviour
- Complexity “emerges” from the diversity of the system parts and their interrelations and interactions
- For (imperative) code one approach to understanding complexity is **cyclomatic**: counting the number of LI CFG paths, with a high number of LI paths indicating high complexity
- Key #1: More decisions/branching in a CFG => more LI paths => more (cyclomatically) complex code

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Cyclomatic Complexity

- General notion of “complexity”: diversity (or variability) and predictability of the modes of behaviour of a system or process, where higher complexity is usually associated with greater diversity and unpredictability of the modes of behaviour
- Complexity “emerges” from the diversity of the system parts and their interrelations and interactions
- For (imperative) code one approach to understanding complexity is **cyclomatic**: counting the number of LI CFG paths, with a high number of LI paths indicating high complexity
- Key #1: More decisions/branching in a CFG => more LI paths => more (cyclomatically) complex code
- Key #2: Each LI path is a unique code execution path, associated with a particular behaviour and which can be tested

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Cyclomatic Complexity

- General notion of “complexity”: diversity (or variability) and predictability of the modes of behaviour of a system or process, where higher complexity is usually associated with greater diversity and unpredictability of the modes of behaviour
- Complexity “emerges” from the diversity of the system parts and their interrelations and interactions
- For (imperative) code one approach to understanding complexity is **cyclomatic**: counting the number of LI CFG paths, with a high number of LI paths indicating high complexity
- Key #1: More decisions/branching in a CFG => more LI paths => more (cyclomatically) complex code
- Key #2: Each LI path is a unique code execution path, associated with a particular behaviour and which can be tested
- Key #3: Cyclomatic complexity has limitations: ignores data flows, exogenous variables, e.g. environment, platform

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Test Coverage

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Test Coverage

- Each LI path in a CFG is a unique code execution path, representing a specific testable behaviour => write at least one test case for each LI path in a CFG!

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Test Coverage

- Each LI path in a CFG is a unique code execution path, representing a specific testable behaviour => write at least one test case for each LI path in a CFG!
- Rigorous test coverage => modelling functions and callables as CFGs, and writing test cases for each LI path

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Test Coverage

- Each LI path in a CFG is a unique code execution path, representing a specific testable behaviour => write at least one test case for each LI path in a CFG!
- Rigorous test coverage => modelling functions and callables as CFGs, and writing test cases for each LI path
- Ideal: tool to automatically generate test cases based on CFGs; time-saving and can improve productivity

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Test Coverage

- Each LI path in a CFG is a unique code execution path, representing a specific testable behaviour => write at least one test case for each LI path in a CFG!
- Rigorous test coverage => modelling functions and callables as CFGs, and writing test cases for each LI path
- Ideal: tool to automatically generate test cases based on CFGs; time-saving and can improve productivity
- CFGs ignore data flows, which can be incorporated with other specialised tools and techniques, e.g. in Python the bytecode DAG/CFG of a function or callable can pinpoint incoming and outgoing data, as well as data passing within the callable

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Test Coverage

- Each LI path in a CFG is a unique code execution path, representing a specific testable behaviour => write at least one test case for each LI path in a CFG!
- Rigorous test coverage => modelling functions and callables as CFGs, and writing test cases for each LI path
- Ideal: tool to automatically generate test cases based on CFGs; time-saving and can improve productivity
- CFGs ignore data flows, which can be incorporated with other specialised tools and techniques, e.g. in Python the bytecode DAG/CFG of a function or callable can pinpoint incoming and outgoing data, as well as data passing within the callable
- Incorporating data into test coverage means taking into account the type and range of data going into a function or callable, and going out of it, e.g. edge cases of a function often depend on “extreme” or “improbable” or “hidden” values of argument(s) which can be easily missed by developers writing tests

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Test Coverage

- Each LI path in a CFG is a unique code execution path, representing a specific testable behaviour => write at least one test case for each LI path in a CFG!
- Rigorous test coverage => modelling functions and callables as CFGs, and writing test cases for each LI path
- Ideal: tool to automatically generate test cases based on CFGs; time-saving and can improve productivity
- CFGs ignore data flows, which can be incorporated with other specialised tools and techniques, e.g. in Python the bytecode DAG/CFG of a function or callable can pinpoint incoming and outgoing data, as well as data passing within the callable
- Incorporating data into test coverage means taking into account the type and range of data going into a function or callable, and going out of it, e.g. edge cases of a function often depend on “extreme” or “improbable” or “hidden” values of argument(s) which can be easily missed by developers writing tests
- My (open source) GitHub project **ccm** (https://github.com/sr-murthy/ccm) is intended as an aid to automating test coverage in a rigorous way using the cyclomatic approach!

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Security - Vulnerability & Threat Detection

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Security - Vulnerability & Threat Detection

- Each LI path in a CFG is a unique code execution path => data flow modelling along LI paths

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Security - Vulnerability & Threat Detection

- Each LI path in a CFG is a unique code execution path => data flow modelling along LI paths
- Python bytecode DAGs/CFGs can be used to model data flows into, within, and out of, functions and callables

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Security - Vulnerability & Threat Detection

- Each LI path in a CFG is a unique code execution path => data flow modelling along LI paths
- Python bytecode DAGs/CFGs can be used to model data flows into, within, and out of, functions and callables
- This can be used with static code analysis techniques such as data flow analysis and taint tracking to pinpoint vulnerabilities and threats, e.g. SQL injection

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Security - Vulnerability & Threat Detection

- Each LI path in a CFG is a unique code execution path => data flow modelling along LI paths
- Python bytecode DAGs/CFGs can be used to model data flows into, within, and out of, functions and callables
- This can be used with static code analysis techniques such as data flow analysis and taint tracking to pinpoint vulnerabilities and threats, e.g. SQL injection
- Example: SQL injection attacks where malicious or destructive SQL code can be passed to databases and data-driven applications allowing unverified user input to be executed as commands

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Security - Vulnerability & Threat Detection

- Each LI path in a CFG is a unique code execution path => data flow modelling along LI paths
- Python bytecode DAGs/CFGs can be used to model data flows into, within, and out of, functions and callables
- This can be used with static code analysis techniques such as data flow analysis and taint tracking to pinpoint vulnerabilities and threats, e.g. SQL injection
- Example: SQL injection attacks where malicious or destructive SQL code can be passed to databases and data-driven applications allowing unverified user input to be executed as commands
- GitHub's CodeQL uses cyclomatic techniques in its code analysis engine to identify vulnerabilities - available as a CI workflow with GitHub Actions

Graphs, Code & Cyclomatic Complexity

A Brief Overview & Applications

Applications: Security - Vulnerability & Threat Detection

- Each LI path in a CFG is a unique code execution path => data flow modelling along LI paths
- Python bytecode DAGs/CFGs can be used to model data flows into, within, and out of, functions and callables
- This can be used with static code analysis techniques such as data flow analysis and taint tracking to pinpoint vulnerabilities and threats, e.g. SQL injection
- Example: SQL injection attacks where malicious or destructive SQL code can be passed to databases and data-driven applications allowing unverified user input to be executed as commands
- GitHub's CodeQL uses cyclomatic techniques in its code analysis engine to identify vulnerabilities - available as a CI workflow with GitHub Actions
- My (open source) GitHub project **ccm** (https://github.com/sr-murthy/ccm) is intended as an aid to identifying static vulnerabilities through data flow analysis for Python source code using bytecode CFGs