

COMP1531

5.3 Software Complexity

How complicated is
software?

No Silver Bullet

- A famous paper from 1986:
 - *No Silver Bullet – Essence and Accident in Software Engineering* by Fred Brooks
- Described software complexity by dividing it into two categories *essential* and *accidental*.
- Further conclusions of the paper are much debated

Essential

Complexity that is inherent to the problem.

For example, if the user or client requires the program to do 30 different things, then those 30 things are essential

Accidental

Complexity that is **not** inherent to the problem.

For example, generating or parsing data in specific formats.

Essential

Fundamentally can't be removed, but can be managed with good *software design*.

Accidental

Can be somewhat mitigated by engineering decisions; e.g. smart use of libraries, standards, etc.

Hard to remove entirely.

Open questions

- Is there a non-intuitive process for distinguishing accidental and essential complexity?
- How much of the complexity of modern software is accidental?
- To what degree has or will accidental complexity be removed in future?

Further reading

- The original No Silver Bullet paper:
 - <http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf>
- A more modern description:
 - <https://stevemcconnell.com/articles/software-engineering-principles/>
- A recent rebuttal:
 - <https://blog.ploeh.dk/2019/07/01/yes-silver-bullet/>

Can we measure
complexity?

Coupling

- A measure of how closely connected different software components are
- Usually expressed as a simple ordinal measure of "loose" and "tight"
- For example, web applications tend to have a frontend that is loosely coupled from the backend

Cohesion

- The degree to which elements of a module belong together
- Elements belong together if they're somehow related
- Usually expressed as a simple ordinal measure of "low" and "high"
- We'll come back to this in a later week

Cyclomatic complexity

- An *interval* measure of the branching complexity of functions
- Computed by counting the number of *linearly-independent* paths through a function

Cyclomatic complexity

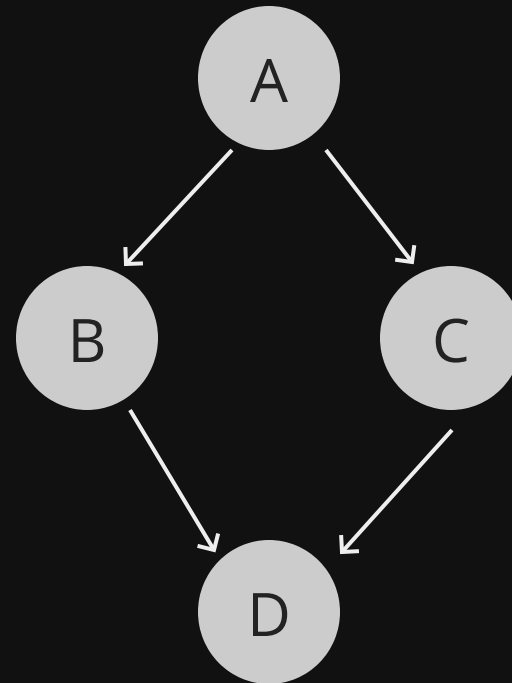
- To compute:
 1. Convert function into a graph
 2. Calculate the value of the formula

$$V(G) = e - n + 2$$

where e is the number of edges and n is the number of nodes

Example 1

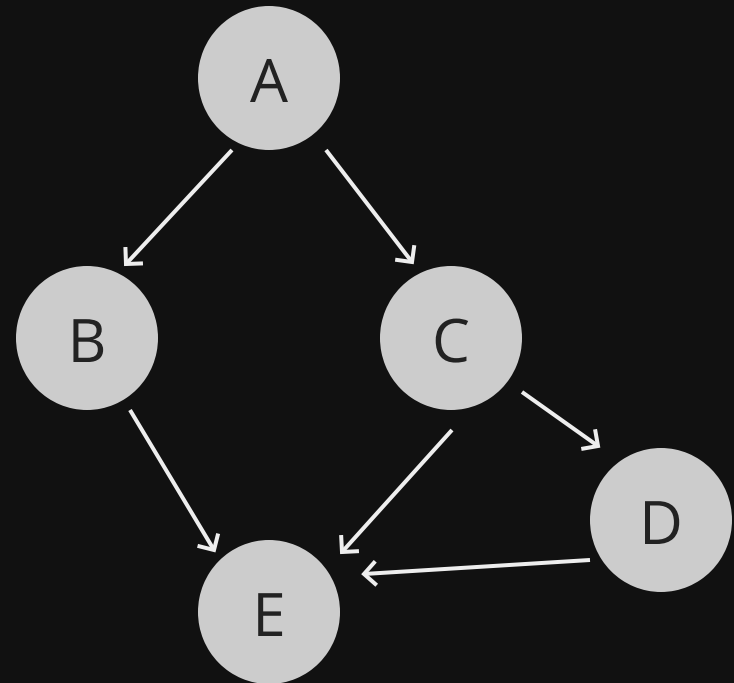
```
1 def foo():  
2     if A:  
3         B  
4     else:  
5         C  
6     D
```



$$V(G) = 4 - 4 + 2 = 2$$

Example 2

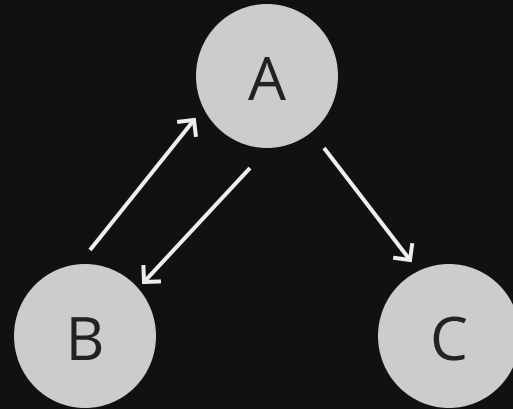
```
1 def foo():  
2     if A:  
3         B  
4     else:  
5         if C:  
6             D  
7         E
```



$$V(G) = 6 - 5 + 2 = 3$$

Example 3

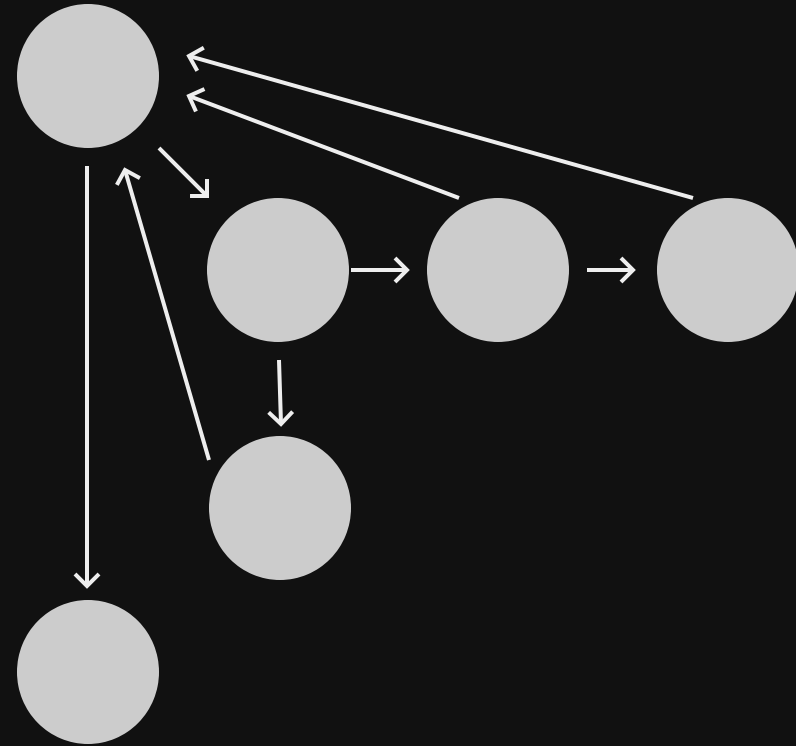
```
1 def foo():  
2     while A:  
3         B  
4     C
```



$$V(G) = 3 - 3 + 2 = 2$$

Example 4

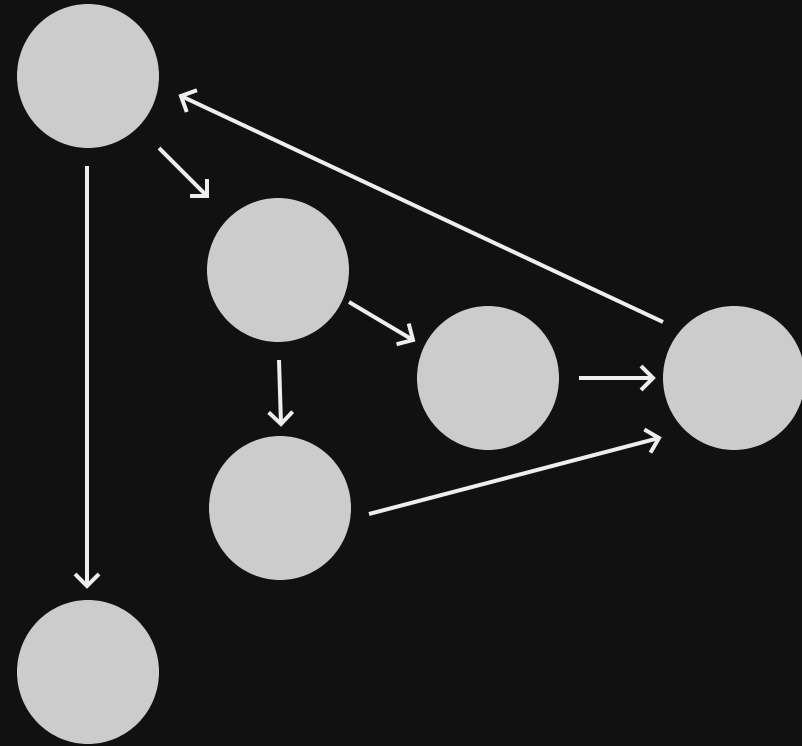
```
1 def day_to_year(days):
2     year = 1970
3
4     while days > 365:
5         if is_leap_year(year):
6             if days > 366:
7                 days -= 366
8                 year += 1
9         else:
10            days -= 365
11            year += 1
12
13     return year
```



$$V(G) = 8 - 6 + 2 = 4$$

Example 5

```
1 def day_to_year(days):  
2     year = 1970  
3  
4     while days > 0:  
5         if is_leap_year(year):  
6             days -= 366  
7         else:  
8             days -= 365  
9             year += 1  
10  
11     return year - 1
```



$$V(G) = 7 - 6 + 2 = 3$$

Usage

- A simple understandable measure of function complexity
- Some people argue 8 should be the maximum cyclomatic complexity of a function

Drawbacks

- Assumes non-branching statements have no complexity
- Keeping cyclomatic complexity low encourages splitting functions up, regardless of whether that really makes the code more understandable