# Software Engineering

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

November 11, 1996

# Code

```
double Vector::average( double low, double high ) const
{
  double sum = 0.0;
  int count = 0;

  for( int i=0; i<theLength; ++i )
  {
    if( (theStorage[i]>=low) && (theStorage[i]<=high) )
    {
      ++count; sum += theStorage[i];
    }
  }

  if( count>0 )
  {
    sum /= (double)count;
  } else {
    cerr << "Vector::average:"
         << "No data in bounds,"
         << "average returned as 0." << endl;
  }

  return( sum );
}
```

## Graph Analysis

Conversion to flow graph

- Atomic statements are nodes, flows are edges
- Convert composite statements/expressions into multiple nodes
- Add "join nodes" if necessary where flow converges

Basic paths

- A basic path is a logically possible execution of the graph
- Basic paths are found in order from short to long
- Each successive basic path traverses at least one new edge
- Each successive basic path has as few new edges as possible
- The set of basic paths is not unique

Cyclomatic complexity

- Complexity = Edges − Nodes + 2
- Provides an upper bound on the number of basic paths

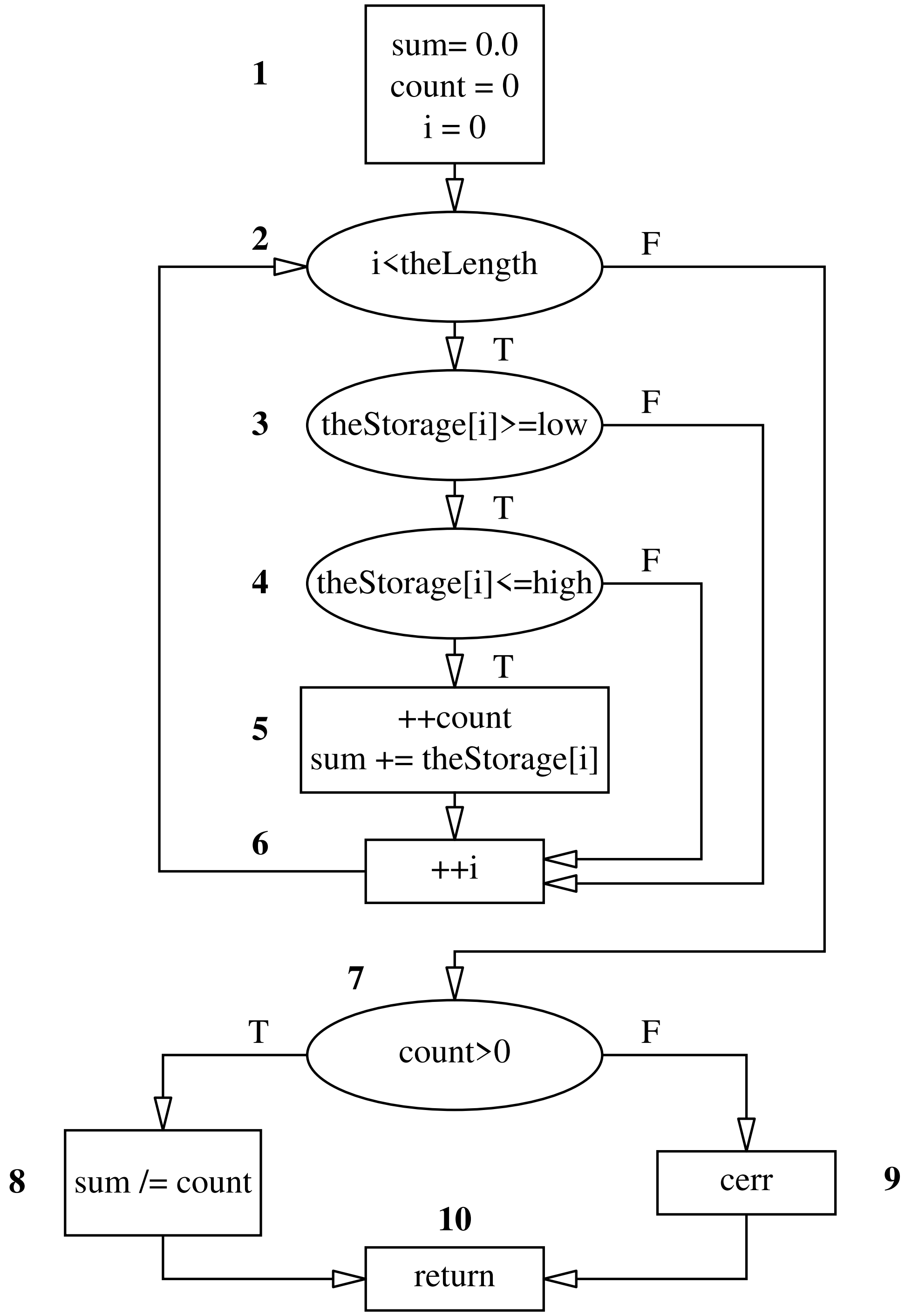
# Remarks

- Cyclomatic complexity
  - Counts all possible distinct paths in the graph
- Not all paths through the algorithm are possible
  - Preconditions may prevent

    `theLength` $>$ `0`
  - Program logic may prevent

    `count` $==$ `0` unless certain paths taken
- What do basic paths provide?
  - An assurance that each edge is traversed
  - An assurance that every logical branch is taken on the **true** and **false** side
- What don't basic paths provide?
  - An assurance that every possible logical sequence is covered
- White box testing should add paths to cover all logical sequences

# Flow Graph


1
sum= 0.0
count = 0
i = 0
2
i<theLength
F
T
3
theStorage[i]>=low
F
T
4
theStorage[i]<=high
F
T
5
++count
sum += theStorage[i]
6
++i
7
count>0
T
F
8
sum /= count
9
cerr
10
return

# Finding Basic Paths

Start with shortest logically possible path.
(Ties may be broken arbitrarily.)

Until the following is no longer possible:

- Find the next longest logically possible path.
  - Should have at least one new edge.
  - Should have the fewest number of new edges.

## Basic Paths for the Example

Complexity:

- $E - N + 2 = 13 - 10 + 2 = 5$
- $P + 1 = 4 + 1 = 5$

With regard to program logic:

- Path 1: **1-2-7-9-10**
- Path 2: **1-2-3-6-2-7-9-10**
- Path 3: **1-2-3-4-6-2-7-9-10**
- Path 4: **1-2-3-4-5-6-2-7-8-10**

There are no further new edges. Distinct new edges were:

- Path 2: **2-3**, **3-6**, **6-2**
- Path 3: **3-4**, **4-6**
- Path 4: **4-5**, **5-6**

# Logical Sequences

| 2 | 3 | 4 | 2 | 3 | 4 | 2 | ... | 7 | possible | path |
|---|---|---|---|---|---|---|---|---|---|---|
| F | | | | | | | | F | yes | 1 |
| F | | | | | | | | T | no | |
| T | F | | F | | | | | F | yes | 2 |
| T | F | | F | | | | | T | no | |
| T | T | F | F | | | | | F | yes | 3 |
| T | T | F | F | | | | | T | no | |
| T | T | T | F | | | | | F | no | |
| T | T | T | F | | | | | T | yes | 4 |
| T | T | T | T | F | | F | | F | no | |
| T | T | T | T | F | | F | | T | yes | |
| T | T | T | T | T | F | F | | F | no | |
| T | T | T | T | T | F | F | | T | yes | |
| T | T | T | T | T | T | F | | F | no | |
| T | T | T | T | T | T | F | | T | yes | |

## More Complete View of Example's Basic Paths

- Path 1: **1-2-7-9-10**
- Path 2: **1-2-3-6-2-** $\cdots$ **-7-9-10**
- Path 3: **1-2-3-4-6-2-** $\cdots$ **-7-9-10**
- Path 4: **1-2-3-4-5-6-2-** $\cdots$ **-7-8-10**

We get the extended sequences by using arrays of size zero, size one, size two, size three ...

# White Box Data

- Path 1

```
theLength = 0
```

- Path 2 ($\cdots =$ `NULL`)

```
theLength = 1
theStorage[0] = 3
low = 5
```

- Path 2 ($\cdots \neq$ `NULL`)

```
theLength = 2
theStorage[0] = 1, theStorage[1] = 3
low = 2, high = 4
```
or
```
low = 4, high = 6
```

- Path 3 ($\cdots =$ `NULL`)

```
theLength = 1
theStorage[0] = 5
low = 3, high = 4
```

- Path 3 ($\cdots \neq$ `NULL`)

```
theLength = 2
theStorage[0] = 2, theStorage[1] = 4
low = 0, high = 1
```
or
```
low = 1, high = 3
```

# Data Continued

- Path 4 ($\cdots =$ NULL)

```
theLength = 1
theStorage[0] = 4
low = 3, high = 5
```

- Path 4 ($\cdots \neq$ NULL)

```
theLength = 2
theStorage[0] = 2, theStorage[1] = 4
low = 0, high = 6
```

or

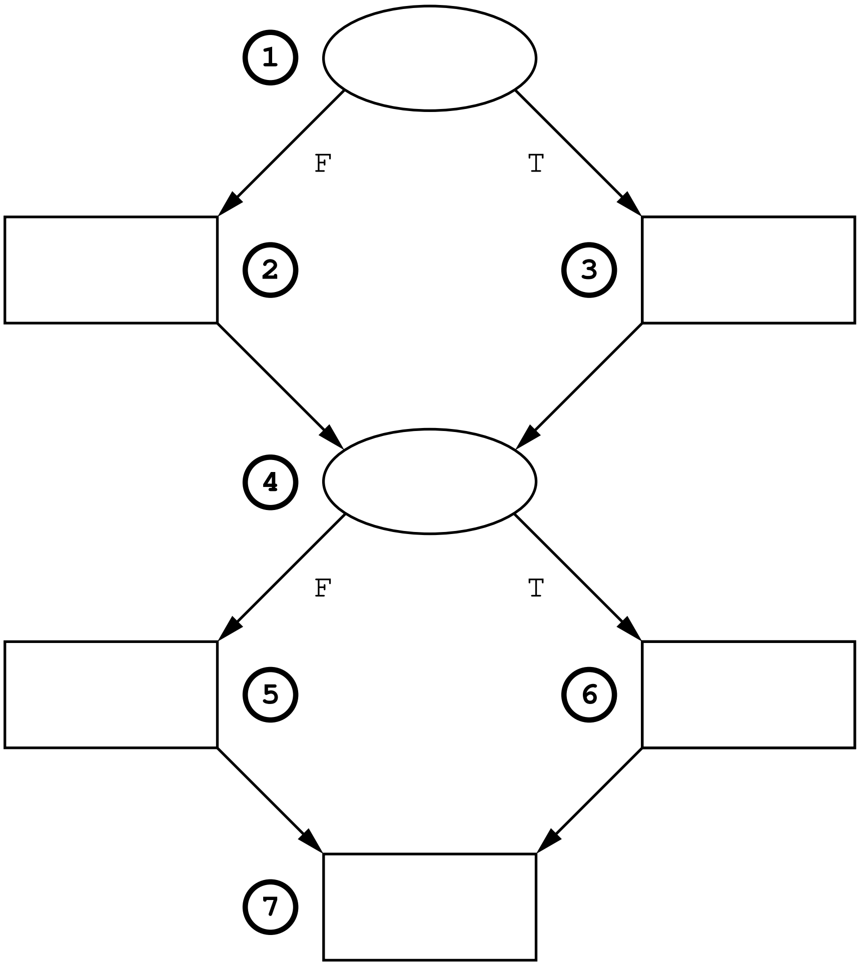
```
low = 2, high = 4
```

## Did we miss anything?

- Maybe:
  - An intermediate case?
    ```
    theLength = 3
    theStorage[0] = 1
    theStorage[1] = 3
    theStorage[2] = 5
    low = 2, high = 4
    ```
  - An unusual case?
    ```
    low = high
    ```

See black box testing.

# Do We Ever Have to Go Beyond Basic Paths?




- Complexity: $8 - 7 + 2 = 3$
- Paths: (**1-3-4-6-7**), (**1-3-4-5-7**), (**1-2-4-5-7**)
- Logical sequences: (**T-T**), (**T-F**), (**F-F**)
- Missing: (**F-T**)

## Black Box Testing

General approach:
(based upon specifications)

- Set up "equivalence sets" of "unusual cases" for input
- Set up "equivalence sets" of "boundary cases" for input
- Set up "equivalence sets" of "usual cases" for input
- Set up "equivalence sets" of "unusual cases" for output
- Set up "equivalence sets" of "boundary cases" for output
- Set up "equivalence sets" of "usual cases" for output

## Black Box Data for Example

Paying attention to input:

- `low > high`
  `low = high`
  `low < high`
- `theLength < 0`
  `theLength = 0`
  `theLength = 1`
  `theLength > 1`
- `theStorage` values all below `low`
  `theStorage` values all above `high`
  `theStorage` values all between `low` and `high`
  `theStorage` values mixed below, between, and above

Paying attention to output:

- data causing the error comment to be printed
- `theStorage` values so that the result averages no numbers
  `theStorage` values so that the result averages one number
  `theStorage` values so that the result averages several numbers
  `theStorage` values so that the result is negative
  `theStorage` values so that the result is positive
  `theStorage` values so that the result cancels to zero