Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

Software Engineering (CS 20006), Spring 2015-16

Selection Sort

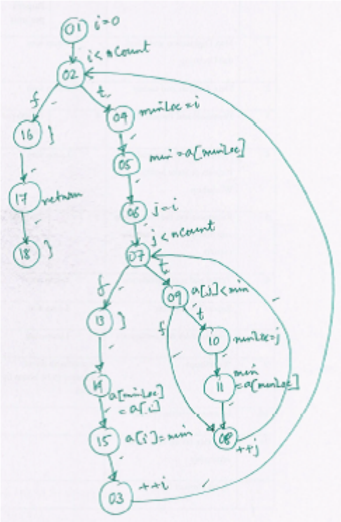
You have developed the `SelectionSort` function to sort an array `a` of `nCount` elements in ascending order. Now you need to prepare various white-box tests for the code. For this you would use program analysis techniques and prepare minimal set of test cases for every scenario.

```
    void SelectionSort(int a[], unsigned int nCount) { // Sorts by Selection
        unsigned int i, j; // Indices to run over the array

01:     for(i = 0;
02:         i < nCount;
03:         ++i) {
04:         unsigned int minLoc = i; // Where the min occurs
05:         int min = a[minLoc];     // Start a[i] as min
06:         for(j = i;
07:             j < nCount;
08:             ++j) { // Find the min in the rest of the array
09:             if (a[j] < min) {
10:                 minLoc = j;
11:                 min = a[minLoc];
12:             }
13:         } // Inner for loop
14:         a[minLoc] = a[i]; // Swap min with a[i]
15:         a[i] = min;
16:     } // Outer for loop
17:     return;
18: }
```

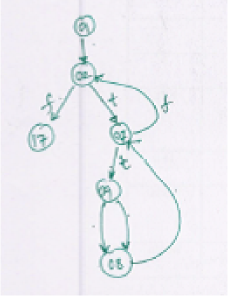
1. Construct the CFG (Control Flow Graph) for `SelectionSort` using the line numbers as shown. [**5**]

Answer:


01
i=0
i< nCount
02
f
t
16
}
17
return
18
}
04
minLoc=i
05
min=a[minLoc]
06
j=i
j< nCount
07
f
t
13
}
14
a[minLoc]
=a[i]
15
a[i]=min
03
++i
09
a[j]<min
t
f
10
minLoc=j
11
min
=a[minLoc]
08
++j

2. Compute the cyclomatic complexity of `SelectionSort` using the CFG in 1. [**2**]

Answer:



Number of nodes $N = 18$

Number of Edges $E = 10$

Cyclomatic Complexity $V(G) = E - N + 2 = 3$

3. Compute the Linearly Independent Paths (LIP) in 1. [**3**]
4. For every LIP in 3, design a test case each that forces the control to trace the LIP. [**3**]
5. Does tests in 4 guarantee 100% line coverage? Justify or identify an uncovered line. [**2**]
6. Show by a counter example that 100% block coverage in 1 may not guarantee 100% branch coverage. [**2**]
7. Compute the `DEF` & `USES` sets and `DU`-chains for `minLoc` and `min`. Design test cases to cover these `DU`-chains. [**2+2=4**]
8. For each of the following mutations (applied separately), check whether the test suite in 4 can kill the mutant. If not, design the killer test. [**2+2=4**]

   - Mutant 1:

     ```
     06:          for(j = i;
     ```

     changed to

     ```
     06:          for(j = 0;
     ```

   - Mutant 2:

     ```
     09:              if (a[j] < min) {
     ```

     changed to

     ```
     09:              if (a[j] > min) {
     ```