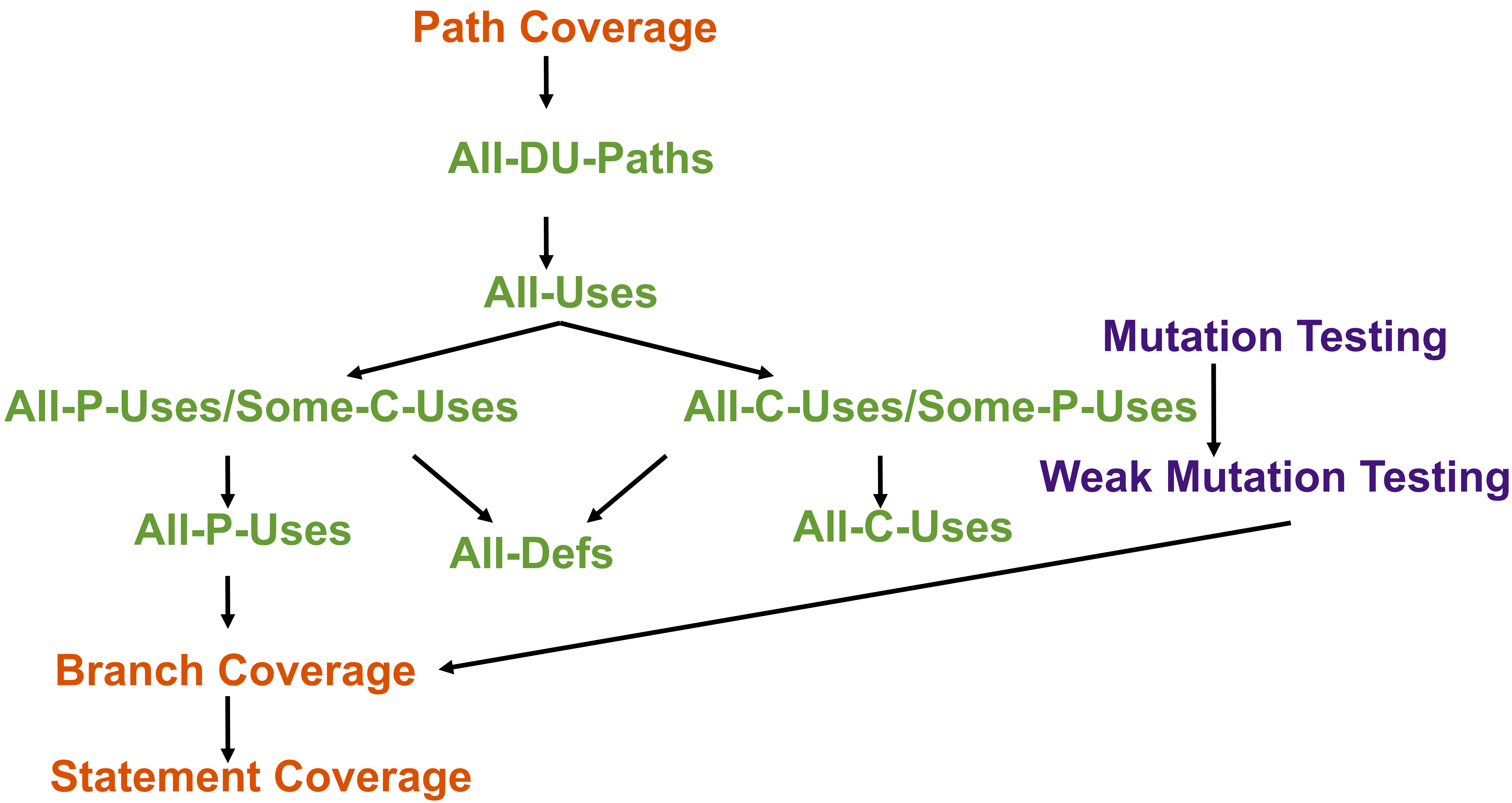
- 1. **(15 points)** Write one JUnit test class (with assertions, e.g., assertEquals and/or assertTrue) to test the following “mid” function with concrete test inputs. Please use test fixture, and mark the corresponding code portions with the following software testing concepts: test case, test fixture, and test oracle. Your tests should be able to cover all the possible program statements within “mid”. Note that you should not use parameterized tests

```
public class TestMe {
s0   public int mid(int x, int y, int z) {
s1         int m = z;
s2         if (y < z) {
s3               if (x < y)
s4                     m = y;
s5               else if (x < z)
s6                     m = x;
s7         } else {
s8               if (x > y)
s9                     m = y;
s10              else if (x > z)
s11                    m = x;
s12        }
s13        return m;
     }
}
```

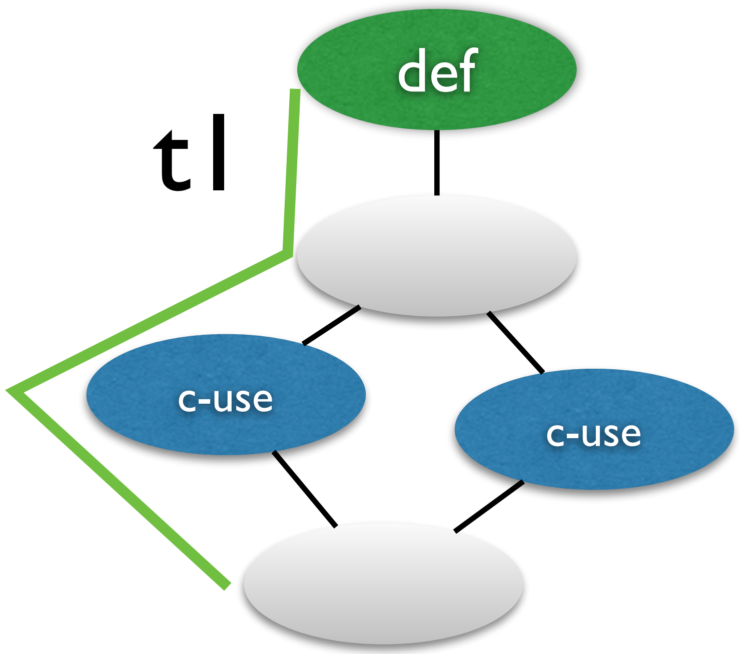
```
public class TestMeTest {
  private TestMe tester;

  @Before  /** Test fixture*/
   public void setUp(){
     tester=new TestMe();
   }

  @Test  /** Test case*/
    public void test() {
       assertEquals(2, tester.mid(1,2,3)); //Oracle
       assertEquals(2, tester.mid(3,2,1)); //Oracle
       assertEquals(2, tester.mid(2,1,3)); //Oracle
       assertEquals(2, tester.mid(2,3,1)); //Oracle
    }

}
```

```
public class TestMe {
s0    public int mid(int x, int y, int z) {
s1          int m = z;
s2          if (y < z) {
s3                if (x < y)
s4                      m = y;
s5                else if (x < z)
s6                      m = x;
s7          } else {
s8                if (x > y)
s9                      m = y;
s10               else if (x > z)
s11                     m = x;
s12         }
s13         return m;
      }
}
```

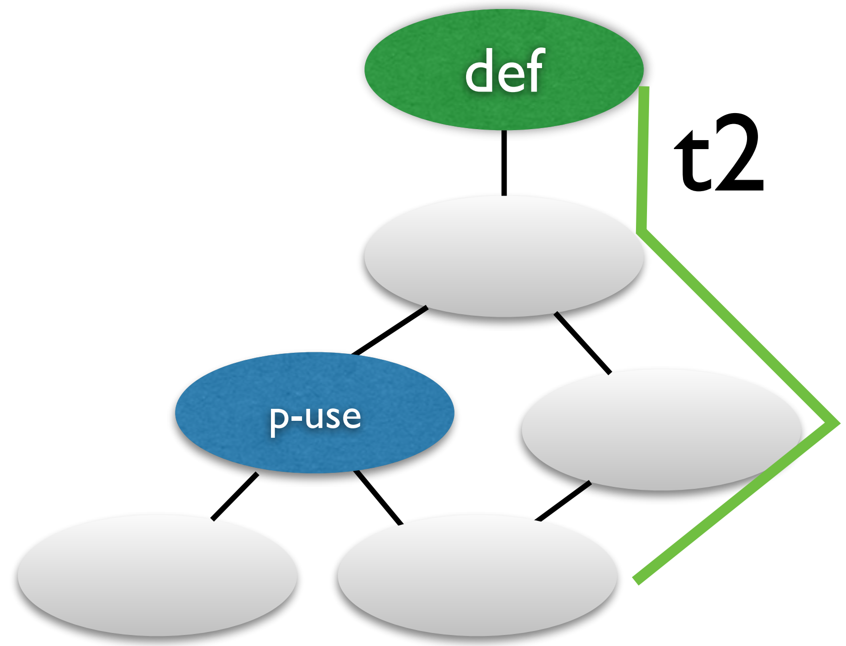
- 2. **(15 points)** Draw the subsumption relationship graph between path coverage, all-du-paths, all-uses, all-defs, all-p-uses/some-c-uses, all-p-uses, all-c-uses/some-p-uses, all-c-uses, branch coverage, statement coverage, mutation testing, and weak mutation testing. (5 points)



2 (a). Illustrate and proof the relationship between all-c-uses and all-defs (5 points).



CFG-1



CFG-2

t1 satisfies all-defs but not all-c-uses

t2 satisfies all-c-uses but not all-defs

all-defs and all-p-uses are incomparable!

2 (b). Illustrate and proof the relationship between weak mutation testing and mutation testing. (5 points)

- For any test suite:
    - If it can kill all the mutants, it can also weakly kill all the mutants
    - If it can weakly kill all the mutants, it may not kill all the mutants
- Thus, mutation testing strictly subsumes weak mutation testing

2 (c). Illustrate and proof the relationship between mutation testing and branch coverage. (5 points)

- For every conditional statement, we can mutate its expression into **false** and **true**
  - E.g., **m1: if(e)=>if(true)**, **m2: if(e)=>if(false)**
- In order to kill **m1**, we have to have internal state change, e.g., **e!=true => e=false**
  - Covering the **false branch** of the original program
- In order to kill **m2**, we have to have internal state change, e.g., **e!=false => e=true, covering the true branch**
  - Covering the **true branch** of the original program
- Thus, we can achieve full branch coverage whenever we have killed all non-equivalent mutants

- 3. **(10 points)** Assume that you want to test your program on different system configurations. There are three configurable variables, OS, CPU, and Protocol, each of which has two options, i.e., OS={Windows, Linux}, CPU={AMD, Intel}, and Protocol={IPV4, IPV6}. Use 3-way and 2-way combinatorial testing to design the minimal number of configurations, respectively.

Config	OS	CPU	Protocol
1	Windows	Intel	IPv4
2	Windows	Intel	IPv6
3	Windows	AMD	IPv4
4	Windows	AMD	IPv6
5	Linux	Intel	IPv4
6	Linux	AMD	IPv6
7	Linux	Intel	IPv4
8	Linux	AMD	IPv6

3-way combinatorial testing

Config	OS	CPU	Protocol
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Linux	AMD	IPv4
4	Linux	Intel	IPv6

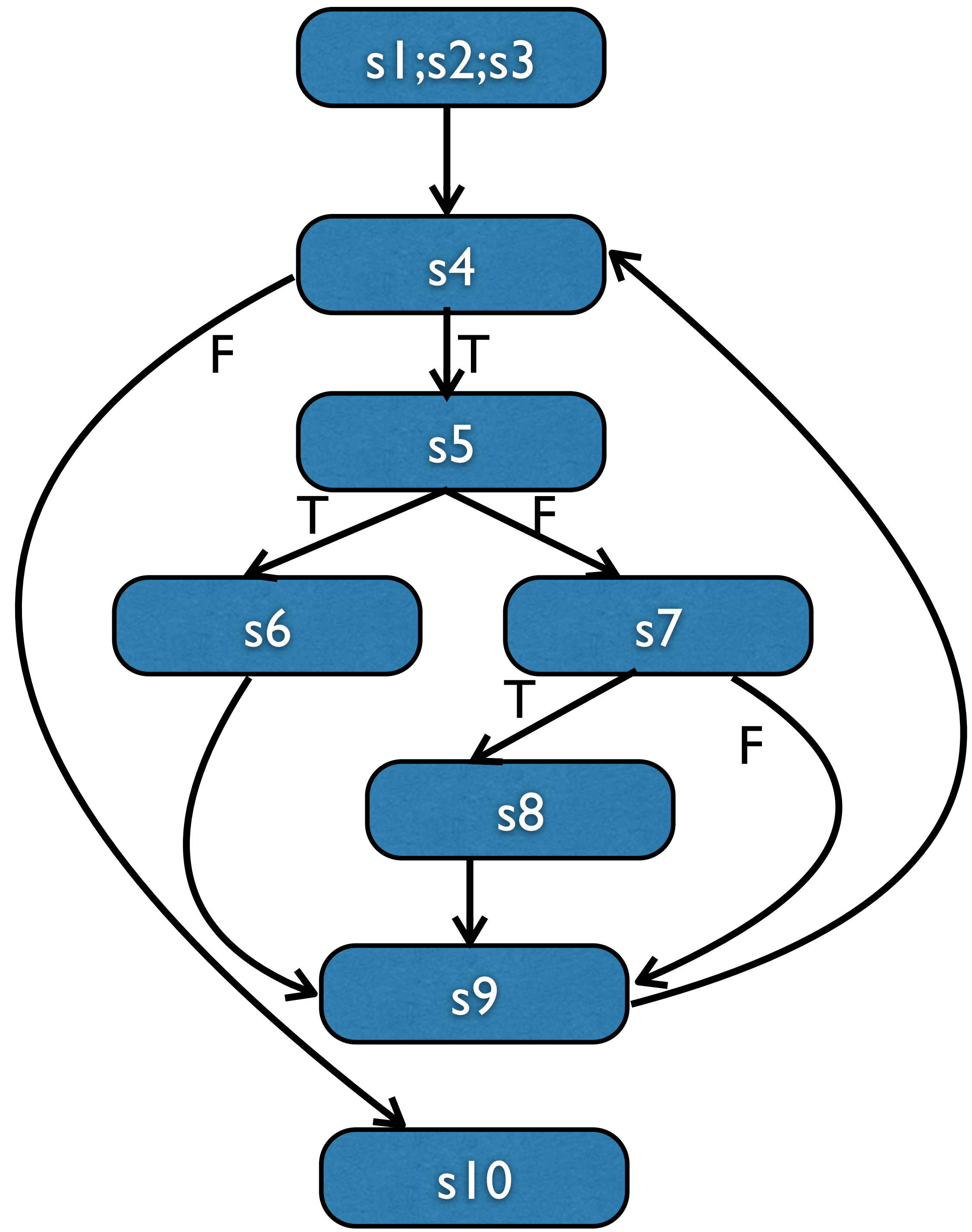
2-way combinatorial testing

- 4. Considering the following program (50points):

```
        public int compute(int x){
s1              int res=0;
s2              int[] a={2,2,3,4};
s3              int i=0;
s4              while(i<a.length){ //b1(true) b2(false)
s5                      if(a[i]>x*2) //b3(true) b4(false)
s6                              res+=2;
s7                      else if(a[i]==x*2) //b5(true) b6(false)
s8                              res+=1;
s9                      i++;
                }
s10             return res;
        }
```

- 4(a). Draw the control flow graph.

```
      public int compute(int x){
s1          int res=0;
s2          int[] a={2,2,3,4};
s3          int i=0;
s4          while(i<a.length){ //b1(true) b2(false)
s5                if(a[i]>x*2) //b3(true) b4(false)
s6                      res+=2;
s7                else if(a[i]==x*2) //b5(true) b6(false)
s8                      res+=1;
s9                i++;
            }
s10         return res;
      }
```



- 4(b). Compute the statement and branch coverage by for test input (x=4) of the above program. (10 points)

```
     public int compute(int x){
s1           int res=0;
s2           int[] a={2,2,3,4};
s3           int i=0;
s4           while(i<a.length){ //b1(true) b2(false)
s5                   if(a[i]>x*2) //b3(true) b4(false)
s6                           res+=2;
s7                   else if(a[i]==x*2) //b5(true) b6(false)
s8                           res+=1;
s9                   i++;
             }
s10          return res;
     }
```

	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10
x=4	1	1	1	1	1		1		1	1

Statement coverage

	b1	b2	b3	b4	b5	b6
x=4	1	1		1		1

Branch coverage

- 4(c). Compute the du-pairs covered by test input (x=2) with respect to variables **a** and **res**. (10 points)

```
     public int compute(int x){
s1       int res=0;
s2       int[] a={2,2,3,4};
s3       int i=0;
s4       while(i<a.length){ //b1(true) b2(false)
s5           if(a[i]>x*2) //b3(true) b4(false)
s6               res+=2;
s7           else if(a[i]==x*2) //b5(true) b6(false)
s8               res+=1;
s9           i++;
         }
s10      return res;
     }
```

Variable a:

(2,<4, 5>)
(2,<4,10>)
(2,<5, 7>)
(2,<7, 8>)
(2,<7, 9>)

- 4(c). Compute the du-pairs covered by test input (x=2) with respect to variables **a** and **res**. (10 points)

```
     public int compute(int x){
s1       int res=0;
s2       int[] a={2,2,3,4};
s3       int i=0;
s4       while(i<a.length){ //b1(true) b2(false)
s5           if(a[i]>x*2) //b3(true) b4(false)
s6               res+=2;
s7           else if(a[i]==x*2) //b5(true) b6(false)
s8               res+=1;
s9           i++;
         }
s10      return res;
     }
```

Variable a:

(2,<4, 5>)
(2,<4,10>)
(2,<5, 7>)
(2,<7, 8>)
(2,<7, 9>)

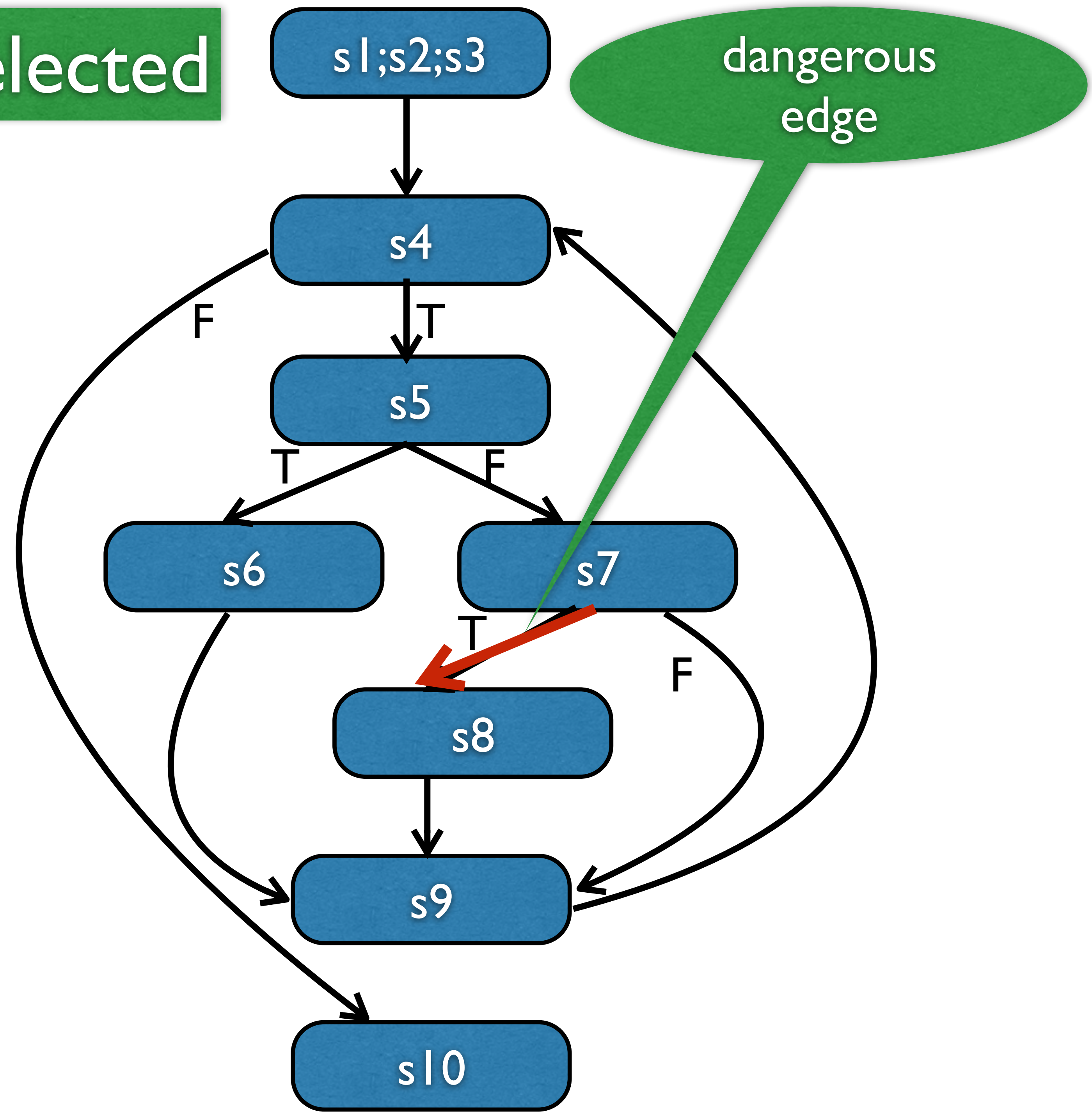
Variable res:

(1,8)
(8,10)

- 4(d). Suppose you have three tests for testing the program (x=0, x=2, and x=4).
  - i. Assume line **s8** is changed in a newer version, apply test selection, i.e., mark the dangerous edge in the control flow graph (5 points), and identify the selected test(s) (5 points).

x=2 will be selected

```
        public int compute(int x){
s1          int res=0;
s2          int[] a={2,2,3,4};
s3          int i=0;
s4          while(i<a.length){ //b1(true) b2(false)
s5              if(a[i]>x*2) //b3(true) b4(false)
s6                  res+=2;
s7              else if(a[i]==x*2) //b5(true) b6(false)
s8                  res+=1;
s9              i++;
            }
s10         return res;
        }
```



- 4(d). Suppose you have three tests for testing the program (x=0, x=2, and x=4).
  - ii. Use greedy or ILP-based test suite reduction based on branch coverage to reduce the tests. (10 points)

	b1	b2	b3	b4	b5	b6
t1=0	1	1	1			
t2=2	1	1		1	1	1
t3=4	1	1		1		1

Branch coverage

- The ILP constraints:
  - Objective: $\text{Minimize}(\sum_{j=1}^{m} x_j),\ x_j \in \{0,1\}$
  - Constraint: $(\sum_{j=1}^{m} s_{1j}x_j \geq 1) \wedge \ldots \wedge (\sum_{j=1}^{m} s_{nj}x_j \geq 1),\ sij \in \{0,1\}$

- Objective:
  - Minimize($t_1 + t_2 + t_3$)
- Constraints:
  - b_1: $1*t_1+1*t_2+1*t_3>=1$
  - b_2: $1*t_1+1*t_2+1*t_3>=1$
  - $\boxed{b_3: 1*t_1+0*t_2+0*t_3>=1}$
  - b_4: $0*t_1+1*t_2+1*t_3>=1$
  - $\boxed{b_5: 0*t_1+1*t_2+0*t_3>=1}$
  - b_6: $0*t_1+1*t_2+1*t_3>=1$

t1, t2

- 5. **(10 points)** The three conditions for a test to kill a mutant are reachability, necessity, and sufficiency. For the following mutant of the program in Question 3, describe the set of conditions satisfied by the following three tests: **x=0, x=2,** and **x=4**. *Note you will not get credit without explanation.*

```
        public int compute(int x){
s1          int res=0;
s2          int[] a={2,2,3,4};
s3          int i=0;
s4          while(i<a.length){
s5              if(a[i]>x*2)
s6                  res+=2;
s7              else if(a[i]<=x*2) // mutated statement, originally was "a[i]==x*2"
s8                  res+=1;
s9              i++;
            }
s10         return res;
        }
```

```
     public int compute(int x){
s1       int res=0;
s2       int[] a={2,2,3,4};
s3       int i=0;
s4       while(i<a.length){
s5           if(a[i]>x*2)
s6               res+=2;
s7           else if(a[i]==x*2)
s8               res+=1;
s9           i++;
         }
s10      return res;
     }
```

Value Replacement

```
     public int compute(int x){
s1       int res=0;
s2       int[] a={2,2,3,4};
s3       int i=0;
s4       while(i<a.length){
s5           if(a[i]>x*2)
s6               res+=2;
s7           else if(a[i]<=x*2)
s8               res+=1;
s9           i++;
         }
s10      return res;
     }
```

P **M**

tests	reachability	necessity	sufficiency	killed?
0				N
2	✓	✓	✓	Y
4	✓	✓	✓	Y

Thanks!