Assignment 2 solution

ECE 653

Yanxin Wang

20439686

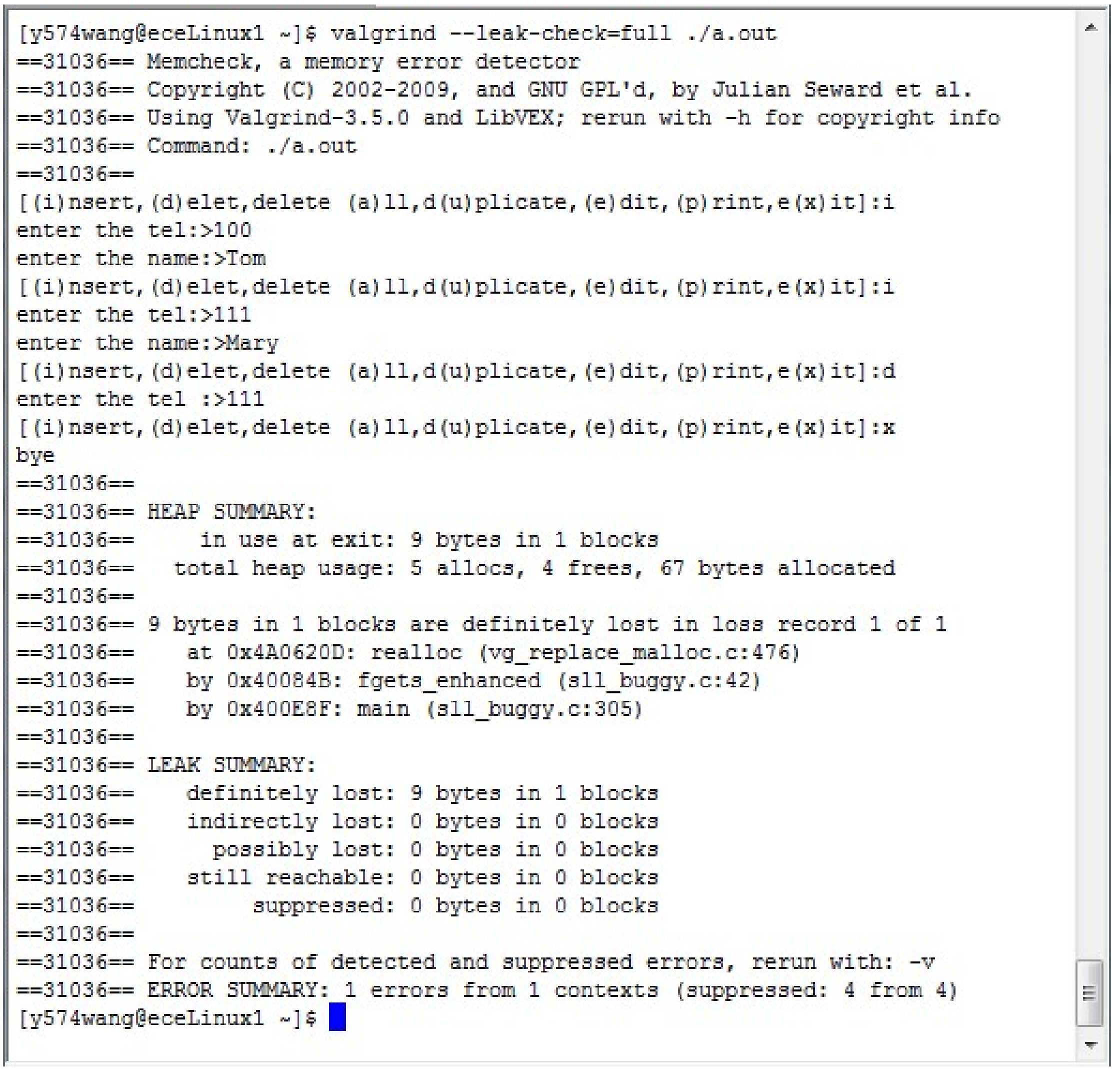
y574wang@uwaterloo.ca

Question 1

For an acyclic graph, Prime Path Coverage (PPC) subsumes Complete Path Coverage (CPC).
Proof: For an acyclic graph, if a test set satisfies PPC but not satisfies CPC, there must be a non-prime path P1 that is not covered. This P1 cannot be a sub-path of a prime path, otherwise it will be covered. Also, since P1 is not a prime path in an acyclic graph, P1 can be extended to a prime path, which means P1 is a sub-path of a prime path. Therefore, it is a contradiction. No such P1 exists.

Question 2

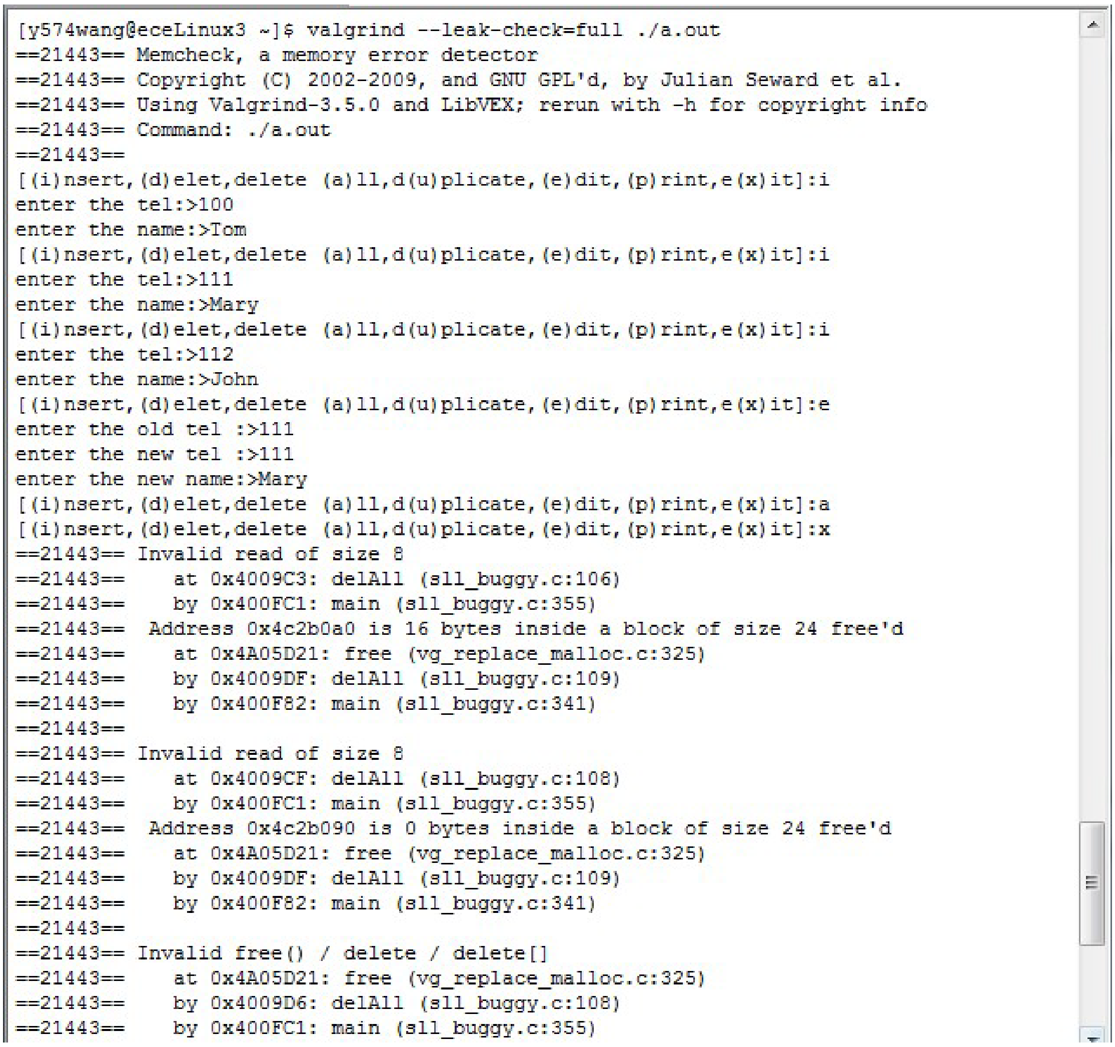
(a) There is a memory leak problem found after running TC1. This is caused by function void delNode(long int num). This function forgets to free the memory allocated to temp->str.



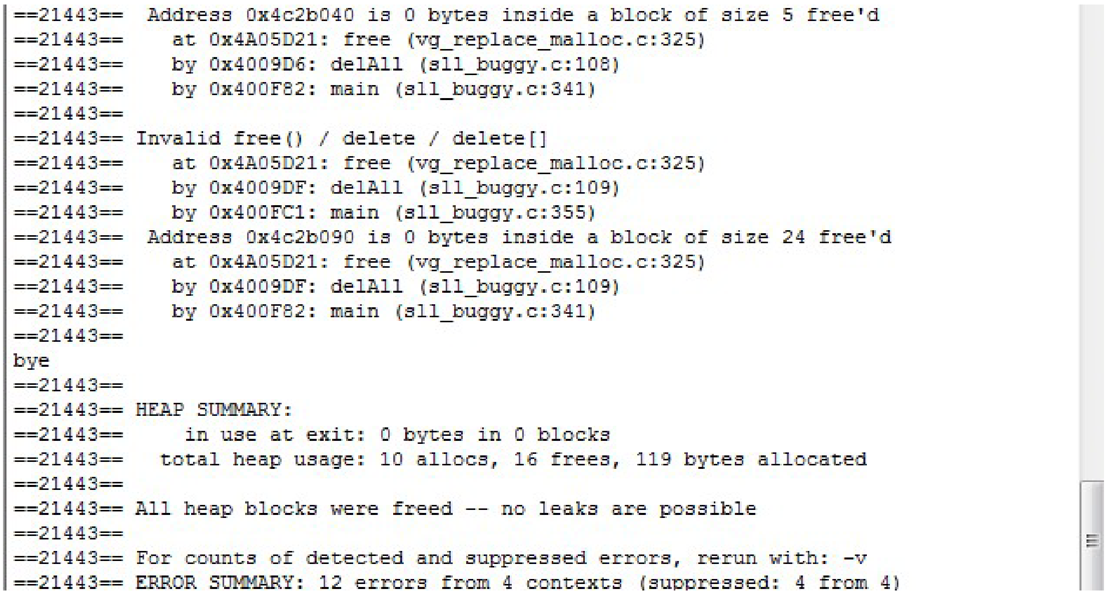
```
[y574wang@eceLinux1 ~]$ valgrind --leak-check=full ./a.out
==31036== Memcheck, a memory error detector
==31036== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==31036== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==31036== Command: ./a.out
==31036==
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Tom
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>111
enter the name:>Mary
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:d
enter the tel :>111
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==31036==
==31036== HEAP SUMMARY:
==31036==     in use at exit: 9 bytes in 1 blocks
==31036==   total heap usage: 5 allocs, 4 frees, 67 bytes allocated
==31036==
==31036== 9 bytes in 1 blocks are definitely lost in loss record 1 of 1
==31036==    at 0x4A0620D: realloc (vg_replace_malloc.c:476)
==31036==    by 0x40084B: fgets_enhanced (sll_buggy.c:42)
==31036==    by 0x400E8F: main (sll_buggy.c:305)
==31036==
==31036== LEAK SUMMARY:
==31036==    definitely lost: 9 bytes in 1 blocks
==31036==    indirectly lost: 0 bytes in 0 blocks
==31036==      possibly lost: 0 bytes in 0 blocks
==31036==    still reachable: 0 bytes in 0 blocks
==31036==         suppressed: 0 bytes in 0 blocks
==31036==
==31036== For counts of detected and suppressed errors, rerun with: -v
==31036== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
[y574wang@eceLinux1 ~]$
```

To fix this bug, this memory should be freed after execution. That is to add free(temp->str) before free(temp).

(b) There is a invalid use problem found after running TC2. This is because function delAll() only frees allocated memory and leaves pointer P pointing to a memory that has been freed. Since function exit() also calls function delAll(), the program tries to free memory a second time when exit() is executed after delAll() in TC2. This results in a invalid operation because the space which P points to is already freed.



```
[y574wang@eceLinux3 ~]$ valgrind --leak-check=full ./a.out
==21443== Memcheck, a memory error detector
==21443== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==21443== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==21443== Command: ./a.out
==21443==
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Tom
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>111
enter the name:>Mary
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>112
enter the name:>John
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:e
enter the old tel :>111
enter the new tel :>111
enter the new name:>Mary
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:a
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
==21443== Invalid read of size 8
==21443==    at 0x4009C3: delAll (sll_buggy.c:106)
==21443==    by 0x400FC1: main (sll_buggy.c:355)
==21443==  Address 0x4c2b0a0 is 16 bytes inside a block of size 24 free'd
==21443==    at 0x4A05D21: free (vg_replace_malloc.c:325)
==21443==    by 0x4009DF: delAll (sll_buggy.c:109)
==21443==    by 0x400F82: main (sll_buggy.c:341)
==21443==
==21443== Invalid read of size 8
==21443==    at 0x4009CF: delAll (sll_buggy.c:108)
==21443==    by 0x400FC1: main (sll_buggy.c:355)
==21443==  Address 0x4c2b090 is 0 bytes inside a block of size 24 free'd
==21443==    at 0x4A05D21: free (vg_replace_malloc.c:325)
==21443==    by 0x4009DF: delAll (sll_buggy.c:109)
==21443==    by 0x400F82: main (sll_buggy.c:341)
==21443==
==21443== Invalid free() / delete / delete[]
==21443==    at 0x4A05D21: free (vg_replace_malloc.c:325)
==21443==    by 0x4009D6: delAll (sll_buggy.c:108)
==21443==    by 0x400FC1: main (sll_buggy.c:355)
```



```
==21443==  Address 0x4c2b040 is 0 bytes inside a block of size 5 free'd
==21443==    at 0x4A05D21: free (vg_replace_malloc.c:325)
==21443==    by 0x4009D6: delAll (sll_buggy.c:108)
==21443==    by 0x400F82: main (sll_buggy.c:341)
==21443==
==21443== Invalid free() / delete / delete[]
==21443==    at 0x4A05D21: free (vg_replace_malloc.c:325)
==21443==    by 0x4009DF: delAll (sll_buggy.c:109)
==21443==    by 0x400FC1: main (sll_buggy.c:355)
==21443==  Address 0x4c2b090 is 0 bytes inside a block of size 24 free'd
==21443==    at 0x4A05D21: free (vg_replace_malloc.c:325)
==21443==    by 0x4009DF: delAll (sll_buggy.c:109)
==21443==    by 0x400F82: main (sll_buggy.c:341)
==21443==
bye
==21443==
==21443== HEAP SUMMARY:
==21443==     in use at exit: 0 bytes in 0 blocks
==21443==   total heap usage: 10 allocs, 16 frees, 119 bytes allocated
==21443==
==21443== All heap blocks were freed -- no leaks are possible
==21443==
==21443== For counts of detected and suppressed errors, rerun with: -v
==21443== ERROR SUMMARY: 12 errors from 4 contexts (suppressed: 4 from 4)
```

To fix this bug, pointer P needs to be set to NULL after free memory. That is add p=Null at the end of delAll(). This way, delAll() will not get into while loop when executed a second time and therefore will avoid invalid memory manipulation.

(c) TC3

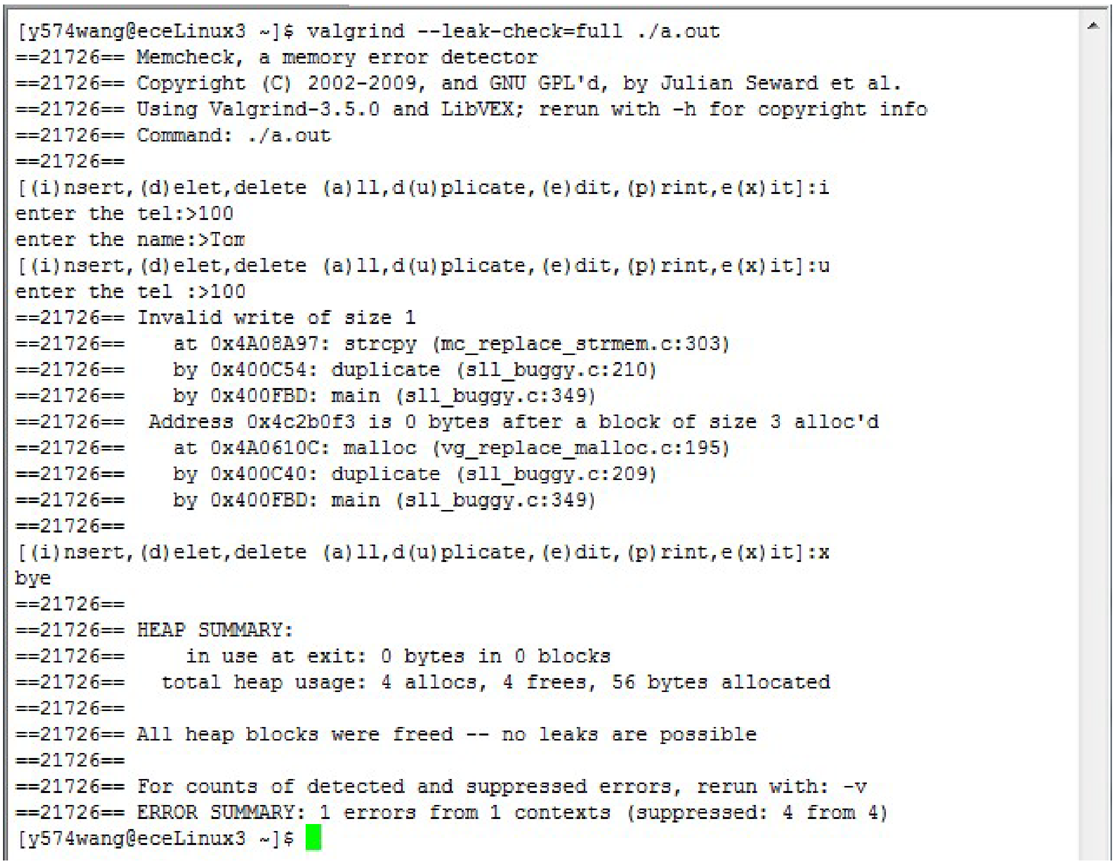
insert>duplicate>exit

[(i)nsert, (d)elet,delete (a)ll, d(u)plicate, (e)dit, (p)rint, e(x)it]:i
enter the tel:>100
enter the name:>Tom

[(i)nsert, (d)elet,delete (a)ll, d(u)plicate, (e)dit, (p)rint, e(x)it]:u
enter the tel:>100

[(i)nsert, (d)elet,delete (a)ll, d(u)plicate, (e)dit, (p)rint, e(x)it]:x

Results:



```
[y574wang@eceLinux3 ~]$ valgrind --leak-check=full ./a.out
==21726== Memcheck, a memory error detector
==21726== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==21726== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==21726== Command: ./a.out
==21726==
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:i
enter the tel:>100
enter the name:>Tom
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:u
enter the tel :>100
==21726== Invalid write of size 1
==21726==    at 0x4A08A97: strcpy (mc_replace_strmem.c:303)
==21726==    by 0x400C54: duplicate (sll_buggy.c:210)
==21726==    by 0x400FBD: main (sll_buggy.c:349)
==21726==  Address 0x4c2b0f3 is 0 bytes after a block of size 3 alloc'd
==21726==    at 0x4A0610C: malloc (vg_replace_malloc.c:195)
==21726==    by 0x400C40: duplicate (sll_buggy.c:209)
==21726==    by 0x400FBD: main (sll_buggy.c:349)
==21726==
[(i)nsert,(d)elet,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:x
bye
==21726==
==21726== HEAP SUMMARY:
==21726==     in use at exit: 0 bytes in 0 blocks
==21726==   total heap usage: 4 allocs, 4 frees, 56 bytes allocated
==21726==
==21726== All heap blocks were freed -- no leaks are possible
==21726==
==21726== For counts of detected and suppressed errors, rerun with: -v
==21726== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
[y574wang@eceLinux3 ~]$ █
```

The problem is caused by insufficient memory allocation in function duplicate(). Function strlen(*string) returns the length of string. But when allocating memory for a string, we need one more byte for the end flag \0.

To fix this bug, one more byte is needed when allocating memory for name. That is to change line: from char* name = malloc(len); to char* name = malloc(len+1);

Question 3

```
int binary search (int a[ ], int low, int high, int target) {
    // binary search for target in a[low, high]
    while ( low <= high ) { //mutant 1: while(low < high) {
    int middle = low + ( high - low ) / 2 ;
    if ( target < a[middle] ) // mutant 2: if( target <= a[middle])
        high = middle - 1 ;
    else if (target > a[middle])
        low = middle + 1 ;
    else
    return middle ;
    }
    return -1; // return -1 if target is not found in a[low, high]
}
```

Test input strongly kill mutant 1: a={1,2,3,4,5,6,7,8,9}, low=1, high=9, target=5

Expected output on original code: 4

Expected output on mutant: -1

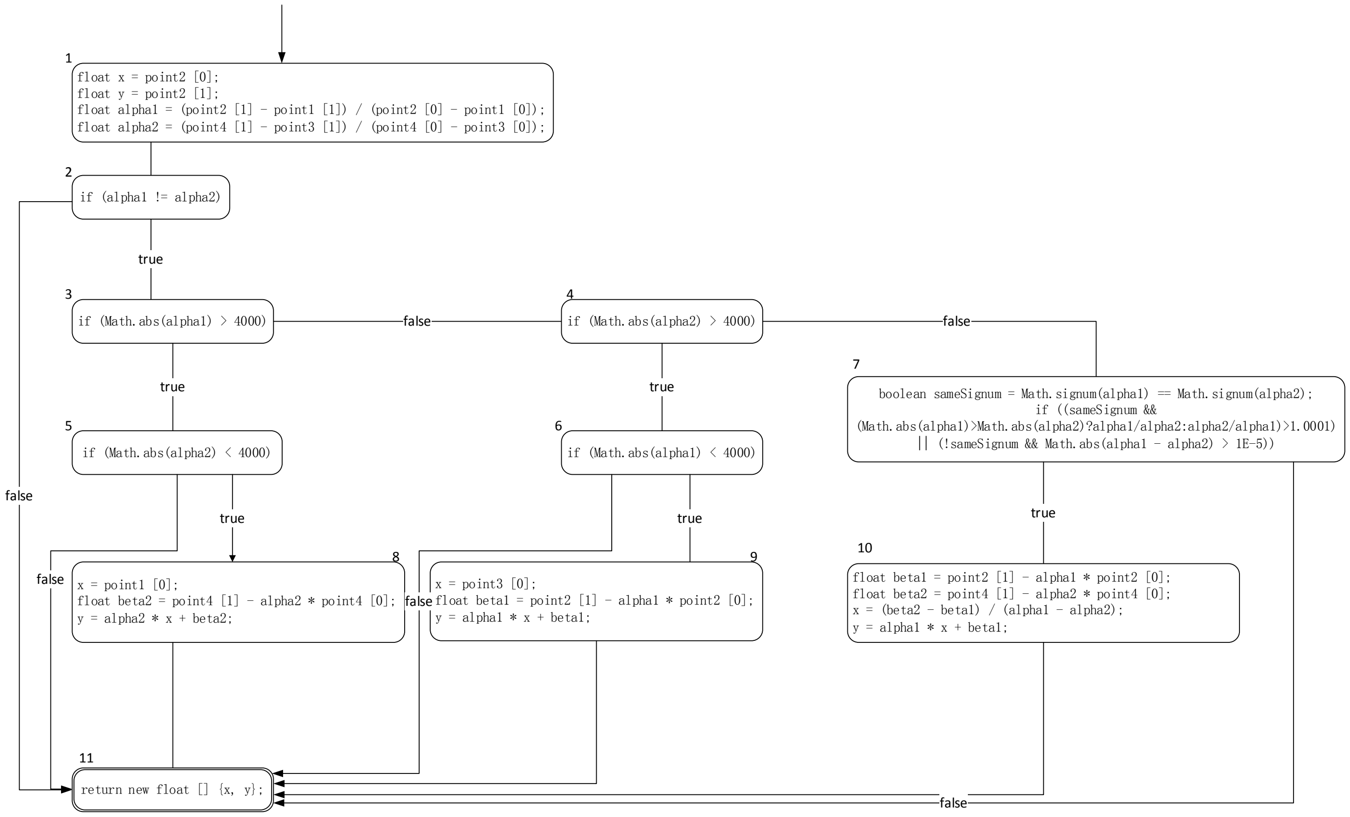
Test input strongly kill mutant 2: a={1,2,3,4,5,6,7,8,9}, low=1, high=9, target=5

Expected output on original code: 4

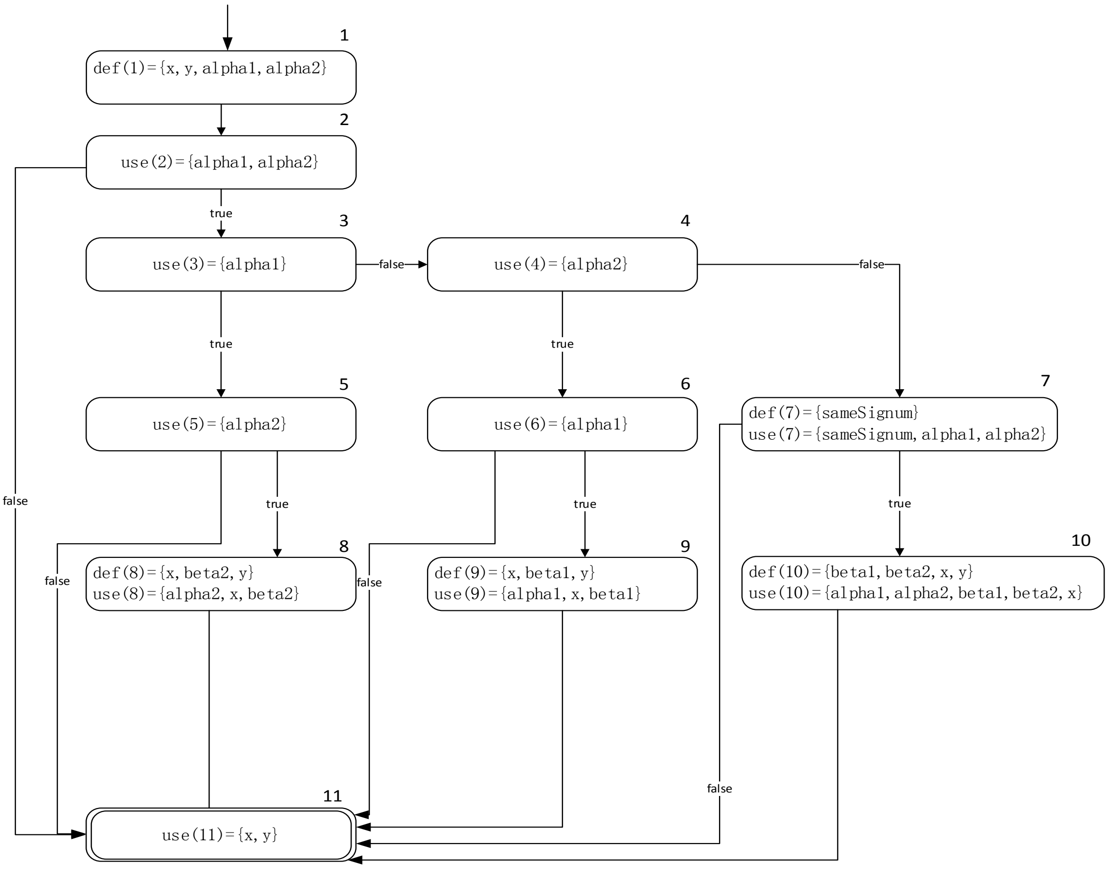
Expected output on mutant: -1

Question 4

(a) Control Flow Graph:



```
float x = point2 [0];
float y = point2 [1];
float alpha1 = (point2 [1] - point1 [1]) / (point2 [0] - point1 [0]);
float alpha2 = (point4 [1] - point3 [1]) / (point4 [0] - point3 [0]);

if (alpha1 != alpha2)

if (Math.abs(alpha1) > 4000)

if (Math.abs(alpha2) > 4000)

if (Math.abs(alpha2) < 4000)

if (Math.abs(alpha1) < 4000)

boolean sameSignum = Math.signum(alpha1) == Math.signum(alpha2);
if ((sameSignum &&
(Math.abs(alpha1)>Math.abs(alpha2)?alpha1/alpha2:alpha2/alpha1)>1.0001)
|| (!sameSignum && Math.abs(alpha1 - alpha2) > 1E-5))

x = point1 [0];
float beta2 = point4 [1] - alpha2 * point4 [0];
y = alpha2 * x + beta2;

x = point3 [0];
float beta1 = point2 [1] - alpha1 * point2 [0];
y = alpha1 * x + beta1;

float beta1 = point2 [1] - alpha1 * point2 [0];
float beta2 = point4 [1] - alpha2 * point4 [0];
x = (beta2 - beta1) / (alpha1 - alpha2);
y = alpha1 * x + beta1;

return new float [] {x, y};
```



(b) $TR_{PPC} =$

$\{[1,2,11], [1,2,3,5,11], [1,2,3,5,7,8,11], [1,2,3,4,6,11], [1,2,3,4,6,9,11], [1,2,3,4,7,11], [1,2,3,4,7,10,11]\}$

(c) With respect to x:

$def = 1,8,9,10$

$use = 8,9,10,11$

Definitions in node 8,9 and 10 are executed before uses in the same node. Therefore, there is no du-path from def in node 1 to uses in node 8,9 and 10

$TR_{ADC} = \{[1,2,11], [8], [9], [10]\}$

$TR_{AUC} = \{[1,2,11], [8], [9], [10], [8,11], [9,11], [10,11]\}$

TR_{ADUPC}

$= \{[1,2,11], [8], [9], [10], [8,11], [9,11], [10,11], [1,2,3,5,11], [1,2,3,4,6,11], [1,2,3,4,7,11]\}$

With respect to y:

$def = 1,8,9,10$

$use = 11$

$TR_{ADC} = \{[1,2,11], [8,11], [9,11], [10,11]\}$

$TR_{AUC} = \{[1,2,11], [8,11], [9,11], [10,11]\}$

$TR_{ADUPC} = \{[1,2,11], [8,11], [9,11], [10,11], [1,2,3,5,11], [1,2,3,4,6,11], [1,2,3,4,7,11]\}$

With respect to alpha1:

$def = 1$

$use = 2,3,6,7,9,10$

*Alpha1 is use twice in node 9, three times in node 10, four times in node 7(in if-statement).

*If-statement in node 7 is assumed to be fully executed every time.

$TR_{ADC} = \{[1,2]\}$

$TR_{AUC} = \{[1,2], [1,2,3], [1,2,3,4,6], [1,2,3,4,7], [1,2,3,4,6,9], [1,2,3,4,7,10]\}$

$TR_{ADUPC} = \{[1,2], [1,2,3], [1,2,3,4,6], [1,2,3,4,7], [1,2,3,4,6,9], [1,2,3,4,7,10]\}$

*$[1,2,3,4,6,9]$ denotes both du-paths to two uses in node 9;

*$[1,2,3,4,6,10]$ denotes all du-paths to three uses in node 10;

*$[1,2,3,4,7]$ denotes all du-paths to four uses in node 7

With respect to alpha2:

$def = 1$

$use = 2,4,5,7,8,10$

*Alpha2 is used twice in node 8 and node 10, four times in node 7(in if-statement).

*If-statement in node 7 is assumed to be fully executed every time.

$TR_{ADC} = \{[1,2]\}$

$TR_{AUC} = \{[1,2], [1,2,3,4], [1,2,3,5], [1,2,3,4,7], [1,2,3,5,8], [1,2,3,4,7,10]\}$

$TR_{ADUPC} = \{[1,2], [1,2,3,4], [1,2,3,5], [1,2,3,4,7], [1,2,3,5,8], [1,2,3,4,7,10]\}$

*$[1,2,3,5,8]$ denotes both du-paths to two uses in node 8;

*$[1,2,3,4,7,10]$ denotes both du-paths to two uses in node 10;

*$[1,2,3,4,7]$ denotes all du-paths to four uses in node 7

With respect to beta1:

$def = 9{,}10$

$use = 9{,}10$

*Beta1 is used twice in node 10

$TR_{ADC} = \{[9],[10]\}$

$TR_{AUC} = \{[9],[10]\}$

$TR_{ADUPC} = \{[9],[10]\}$

*$[10]$ denotes both du-paths to two uses

With respect to beta2:

$def = 8{,}10$

$use = 8{,}10$

$TR_{ADC} = \{[8],[10]\}$

$TR_{AUC} = \{[8],[10]\}$

$TR_{ADUPC} = \{[8],[10]\}$

With respect to sameSignum:

$def = 7$

$use = 7$

*SameSignum is used twice in node 7(in if-statement).

$TR_{ADC} = \{7\}$

$TR_{AUC} = \{7\}$

$TR_{ADUPC} = \{7\}$

(d) PPC subsumes ADC, AUC and ADUPC. But PPC is designed to test structures and ADC, AUC and ADUPC are designed to test data flow. The additional number of test requirements is worth it if the unit you are testing is critical. Otherwise, it is worthless.

(e) $$Test\ set = \left\{\begin{array}{c} (point1 = [0{,}0], point2 = [1{,}0], point3 = [0{,}1], point4 = [1{,}1]), \\ (point1 = [0{,}0], point2 = [1{,}4001], point3 = [0{,}1], point4 = [1{,}1]), \\ (point1 = [0{,}0], point2 = [1{,}4001], point3 = [0{,}1], point4 = [1{,}6001]), \\ (point1 = [0{,}0], point2 = [1{,}3999], point3 = [0{,}1], point4 = [1{,}4002]), \\ (point1 = [0{,}0], point2 = [1{,}4000], point3 = [0{,}1], point4 = [1{,}4002]), \\ (point1 = [0{,}0], point2 = [1{,}3000], point3 = [0{,}1], point4 = [1{,}1000]), \\ (point1 = [0{,}0], point2 = [1{,}1.0001], point3 = [0{,}1], point4 = [1{,}2]) \end{array}\right\}$$

We have test requirements for Prime Path Coverage(PPC):

$$TR_{PPC} = \{[1{,}2{,}11],[1{,}2{,}3{,}5{,}11],[1{,}2{,}3{,}5{,}7{,}8{,}11],[1{,}2{,}3{,}4{,}6{,}11],[1{,}2{,}3{,}4{,}6{,}9{,}11],[1{,}2{,}3{,}4{,}7{,}11],[1{,}2{,}3{,}4{,}7{,}10{,}11]\}$$

For$(point1 = [0{,}0], point2 = [1{,}0], point3 = [0{,}1], point4 = [1{,}1])$, alpha1=0, alpha2=0, alpha1=alpha2, this test case covers $[1{,}2{,}11]$

For$(point1 = [0{,}0], point2 = [1{,}4001], point3 = [0{,}1], point4 = [1{,}1])$, alpha1=4001>4000, alpha2=0<4000, this test case covers $[1{,}2{,}3{,}5{,}7{,}8{,}11]$

For($point1 = [0,0], point2 = [1,4001], point3 = [0,1], point4 = [1,6001]$), alpha1=4001>4000, alpha2=6000>4000, this test case covers $[1,2,3,5,11]$

For($point1 = [0,0], point2 = [1,3999], point3 = [0,1], point4 = [1,4002]$), alpha1=3999<4000, alpha2=4001>4000, this test case covers $[1,2,3,4,6,9,11]$

For($point1 = [0,0], point2 = [1,4000], point3 = [0,1], point4 = [1,4002]$), alpha1=4000, alpha2=4001>4000, this test case covers $[1,2,3,4,6,11]$

For($point1 = [0,0], point2 = [1,3000], point3 = [0,1], point4 = [1,1000]$), alpha1=3000<4000, alpha2=999<4000, alpha1/alpha2>1.0001, this covers $[1,2,3,4,7,10,11]$

For($point1 = [0,0], point2 = [1,1.0001], point3 = [0,1], point4 = [1,2]$), alpha1=1.0001<4000, alpha2=1<4000, alpha1/alpha2=1.0001, this covers $[1,2,3,4,7,11]$