SE 465 - Assignment 2

Bilal Khan - 20873614
b54khan@uwaterloo.ca

February 22, 2023

Question 1

(a)

EPC does not subsume PPC because individual edge-pairs can only individually test paths in the control flow graph of the program of length 2, prime paths can test paths in the control flow graph of the program of longer lengths.

(b)

PPC does subsume EPC because all edge pairs will be included as a subpath of some prime path in the CFG.

Question 2

(a)

The error is a memory leak bug. Every time we allocate a new node, we perform one call to `malloc` to allocte the space for the `node` itself and once to allocate the space for the `*char` buffer to hold the name. However, we only free the space for the `node` itself, and not the space for the `*char` buffer to hold the name. This means that we are leaking memory every time we allocate a new node, and the memory is not freed until the program terminates. The fix is to free the space for the `*char` buffer to hold the name before freeing the space for the `node` itself when deleting a node in `delete_node()`.

```
b54khan@ubuntu2004-010:~/se465/A2/skeleton/q2$ valgrind --leak-check=full
↪  ./sll_buggy.o < case1.txt
==98006== Memcheck, a memory error detector
==98006== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==98006== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==98006== Command: ./sll_buggy.o
==98006==
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:enter the
↪  tel:>enter the name:>[(i)nsert,(d)elete,delete
↪  (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:enter the tel:>enter the
↪  name:>[(i)nsert,(d)elete,delete
↪  (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:enter the tel
↪  :>[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:bye
```

```
==98006==
==98006== HEAP SUMMARY:
==98006==     in use at exit: 9 bytes in 1 blocks
==98006==   total heap usage: 7 allocs, 6 frees, 9,283 bytes allocated
==98006==
==98006== 9 bytes in 1 blocks are definitely lost in loss record 1 of 1
==98006==    at 0x483DFAF: realloc (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==98006==    by 0x1093A3: fgets_enhanced (sll_buggy.c:47)
==98006==    by 0x109A33: main (sll_buggy.c:279)
==98006==
==98006== LEAK SUMMARY:
==98006==    definitely lost: 9 bytes in 1 blocks
==98006==    indirectly lost: 0 bytes in 0 blocks
==98006==      possibly lost: 0 bytes in 0 blocks
==98006==    still reachable: 0 bytes in 0 blocks
==98006==         suppressed: 0 bytes in 0 blocks
==98006==
==98006== For lists of detected and suppressed errors, rerun with: -s
==98006== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

(b)

The error is a double free bug. The `a` command calls `delete_all()` which deletes and frees all the nodes in the list. However, when the `x` command calls `delete_all()` again, it tries to free the nodes again, which causes a double free error. The reason why there is a double free error is that, while `delete_all()` has a guard to check if the pointer for each is already `NULL` before attempting to call `free()` on it, it never sets the pointer to `NULL` after freeing it. This means that if `delete_all()` is called again, it will attempt to free the same node again, which causes a double free error. The fix is to set the head of the list to `NULL` after freeing it in `delete_all()`.

Note that this fix will only fix the particular bug in this test case that deals with calling `delete_all()` twice. There are multiple other locations in the program that call `free()` on other pointers without setting their values to `NULL` after freeing them, which could cause double free errors in other test cases.

```
b54khan@ubuntu2004-010:~/se465/A2/skeleton/q2$ valgrind --leak-check=full
↪  ./sll_buggy.o < case2.txt
==180781== Memcheck, a memory error detector
==180781== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==180781== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==180781== Command: ./sll_buggy.o
==180781==
==180781== Invalid read of size 8
==180781==    at 0x109516: delete_all (sll_buggy.c:104)
==180781==    by 0x109B85: main (sll_buggy.c:325)
==180781==  Address 0x4a74520 is 16 bytes inside a block of size 24 free'd
==180781==    at 0x483CA3F: free (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
```

```
==180781==    by 0x109538: delete_all (sll_buggy.c:107)
==180781==    by 0x109B42: main (sll_buggy.c:312)
==180781==  Block was alloc'd at
==180781==    at 0x483B7F3: malloc (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x109819: append (sll_buggy.c:207)
==180781==    by 0x109A45: main (sll_buggy.c:279)
==180781==
==180781== Invalid read of size 8
==180781==    at 0x109522: delete_all (sll_buggy.c:106)
==180781==    by 0x109B85: main (sll_buggy.c:325)
==180781==  Address 0x4a74510 is 0 bytes inside a block of size 24 free'd
==180781==    at 0x483CA3F: free (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x109538: delete_all (sll_buggy.c:107)
==180781==    by 0x109B42: main (sll_buggy.c:312)
==180781==  Block was alloc'd at
==180781==    at 0x483B7F3: malloc (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x109819: append (sll_buggy.c:207)
==180781==    by 0x109A45: main (sll_buggy.c:279)
==180781==
==180781== Invalid free() / delete / delete[] / realloc()
==180781==    at 0x483CA3F: free (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x10952C: delete_all (sll_buggy.c:106)
==180781==    by 0x109B85: main (sll_buggy.c:325)
==180781==  Address 0x4a744c0 is 0 bytes inside a block of size 5 free'd
==180781==    at 0x483CA3F: free (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x10952C: delete_all (sll_buggy.c:106)
==180781==    by 0x109B42: main (sll_buggy.c:312)
==180781==  Block was alloc'd at
==180781==    at 0x483B7F3: malloc (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x109315: fgets_enhanced (sll_buggy.c:31)
==180781==    by 0x109A33: main (sll_buggy.c:279)
==180781==
==180781== Invalid free() / delete / delete[] / realloc()
==180781==    at 0x483CA3F: free (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x109538: delete_all (sll_buggy.c:107)
==180781==    by 0x109B85: main (sll_buggy.c:325)
==180781==  Address 0x4a74510 is 0 bytes inside a block of size 24 free'd
==180781==    at 0x483CA3F: free (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x109538: delete_all (sll_buggy.c:107)
==180781==    by 0x109B42: main (sll_buggy.c:312)
```

```
==180781==  Block was alloc'd at
==180781==    at 0x483B7F3: malloc (in
↪  /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==180781==    by 0x109819: append (sll_buggy.c:207)
==180781==    by 0x109A45: main (sll_buggy.c:279)
==180781==
[(i)nsert,(d)elete,delete (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:enter the
↪  tel:>enter the name:>[(i)nsert,(d)elete,delete
↪  (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:enter the tel:>enter the
↪  name:>[(i)nsert,(d)elete,delete
↪  (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:enter the tel:>enter the
↪  name:>[(i)nsert,(d)elete,delete
↪  (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:enter the old tel :>enter the
↪  new tel :>enter the new name:>[(i)nsert,(d)elete,delete
↪  (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:[(i)nsert,(d)elete,delete
↪  (a)ll,d(u)plicate,(e)dit,(p)rint,e(x)it]:bye
==180781==
==180781== HEAP SUMMARY:
==180781==     in use at exit: 0 bytes in 0 blocks
==180781==   total heap usage: 12 allocs, 18 frees, 9,335 bytes allocated
==180781==
==180781== All heap blocks were freed -- no leaks are possible
==180781==
==180781== For lists of detected and suppressed errors, rerun with: -s
==180781== ERROR SUMMARY: 12 errors from 4 contexts (suppressed: 0 from 0)
```

(c)

The error is a buffer overflow bug caused by the lines

```
long len = strlen(temp->str)>0 ? strlen(temp->str) : 0;
char *name = malloc(len);
strcpy(name, temp->str);
```

The `C` function `strlen()` returns the length of the string, but does not include the null character. This causes a buffer overflow bug when the string is copied to the `name` variable as `strcpy()` copies all bytes in the string until it hits the `NULL` character. The bug can be fixed by adding 1 to the length of the string.

```
long len = strlen(temp->str) + 1;
```

The test case:

```
i
100
Tom
u
100
```

(d)

Clang sanitizer finds the additional bug of a possible buffer overflow caused by the lines

```
char c;
scanf("%s",&c); /* ENTER A VALUE FOR SWITCH */
```

Here, `scanf()` is used to read in a string of characters (through the argument `%s`) terminated by the first whitespace character found in `stdin` to a variable of type `char`. This causes a buffer overflow bug as the variable `c` is only one byte in size. The bug will be triggered by any test case (in our case we ran it on Test Case 1) and can be fixed by changing the specifier to `%c`:

```
scanf("%c",&c); /* ENTER A VALUE FOR SWITCH */
```

Clang sanitizer output:

```
b54khan@ubuntu2004-004:~/se465/A2/skeleton/q2$ clang -fsanitize=address -g
 ↪  -O0 sll_buggy.c -o sll_buggy.o
b54khan@ubuntu2004-004:~/se465/A2/skeleton/q2$ ./sll_buggy.o < case1.txt
=================================================================
==100548==ERROR: AddressSanitizer: stack-buffer-overflow on address
 ↪  0x7ffc1dc3d161 at pc 0x55f3200af8d0 bp 0x7ffc1dc3d010 sp 0x7ffc1dc3c798
WRITE of size 2 at 0x7ffc1dc3d161 thread T0
    #0 0x55f3200af8cf in scanf_common(void*, int, bool, char const*,
     ↪  __va_list_tag*) asan_interceptors.cpp.o
    #1 0x55f3200b0316 in __interceptor___isoc99_scanf
     ↪  (/u4/b54khan/se465/A2/skeleton/q2/sll_buggy.o+0x3f316) (BuildId:
     ↪  e032d66b62155579589f141b07ee2666029bea04)
    #2 0x55f32014d6a0 in main
     ↪  /u4/b54khan/se465/A2/skeleton/q2/sll_buggy.c:268:9
    #3 0x7f6b74050082 in __libc_start_main
     ↪  /build/glibc-SzIz7B/glibc-2.31/csu/../csu/libc-start.c:308:16
    #4 0x55f32008f32d in _start
     ↪  (/u4/b54khan/se465/A2/skeleton/q2/sll_buggy.o+0x1e32d) (BuildId:
     ↪  e032d66b62155579589f141b07ee2666029bea04)

Address 0x7ffc1dc3d161 is located in stack of thread T0 at offset 33 in frame
    #0 0x55f32014d54f in main
     ↪  /u4/b54khan/se465/A2/skeleton/q2/sll_buggy.c:260

  This frame has 6 object(s):
    [32, 33) 'c' (line 267) <== Memory access at offset 33 overflows this
     ↪  variable
    [48, 56) 'num' (line 274)
    [80, 88) 'num9' (line 284)
    [112, 120) 'old_num' (line 292)
    [144, 152) 'new_num' (line 292)
    [176, 184) 'num24' (line 317)
HINT: this may be a false positive if your program uses some custom stack
 ↪  unwind mechanism, swapcontext or vfork
```

```
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow asan_interceptors.cpp.o in
↪  scanf_common(void*, int, bool, char const*, __va_list_tag*)
Shadow bytes around the buggy address:
  0x100003b7f9d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100003b7f9e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100003b7f9f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100003b7fa00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100003b7fa10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x100003b7fa20: 00 00 00 00 00 00 00 00 f1 f1 f1 f1[01]f2 f8 f2
  0x100003b7fa30: f2 f2 f8 f2 f2 f2 f8 f2 f2 f2 f8 f2 f2 f2 f8 f3
  0x100003b7fa40: f3 f3 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
  0x100003b7fa50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100003b7fa60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100003b7fa70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==100548==ABORTING
```

After fixing this, we can find the same bugs from parts (a) - (c) using clang sanitizer.

(e)

Valgrind seems to have less features than clang sanitizer expecially when it comes to explaining to the user what type of error is occuring. Valgrind usually gives you a line number and a vague category of either X bytes lost or illegal read of size X. Clang sanitizer returns a more detailed error message with the same information but also has a wider category of error messages it displays. Finally, Clang sanitizer also picks up on the possible stack overflow error that Valgrind does not.

Question 3

We propose two mutants:

```
int list_has_cycle_mutant_1(NODE *list)
{
        NODE *fast=list;
        while(1) {
                if(!(fast=fast->next)) return 1;
                if(fast==list) return 0;
                if(!(fast=fast->next)) return 0;
                if(fast==list) return 0;
                list=list->next;
        }
        return 0;
}

int list_has_cycle_mutant_2(NODE *list)
{
        NODE *fast=list;
        while(1) {
                if(!(fast=fast->next)) return 0;
                if(fast!=list) return 1;
                if(!(fast=fast->next)) return 0;
                if(fast==list) return 1;
                list=list->next;
        }
        return 0;
}
```

We use the following test suite:

```
int main()
{
        NODE n1, n2, n3, n4, n5;

        n1.next=&n2;
        n2.next=&n3;
        n3.next=&n4;
        n4.next=&n5;
        n5.next=NULL;

        printf("Test without cycle: ");
        if(list_has_cycle(&n1)) printf("cycle\n");
        else printf("no cycle\n");

        n1.next=&n1;

        printf("Test with cycle (n1 -> n1 -> n1): ");
```

```
        if(list_has_cycle(&n1)) printf("cycle\n");
        else printf("no cycle\n");

        return 0;
}
```

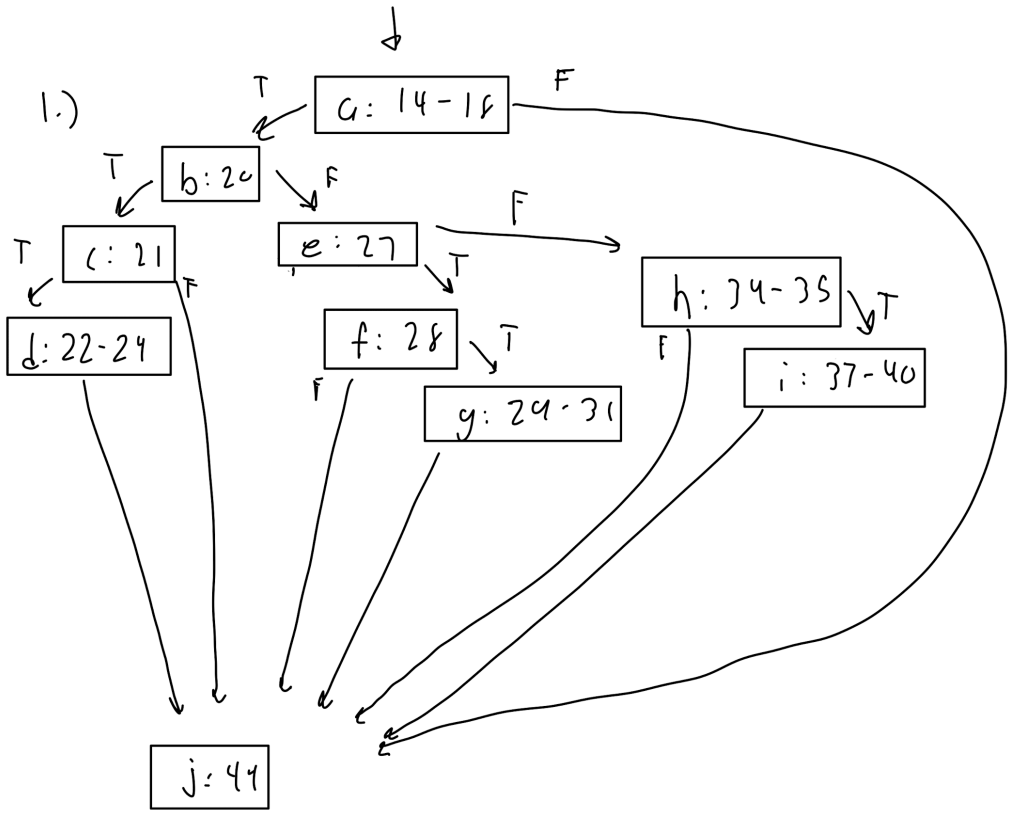
We run the test suite on the original code and the two mutants. The results are as follows:

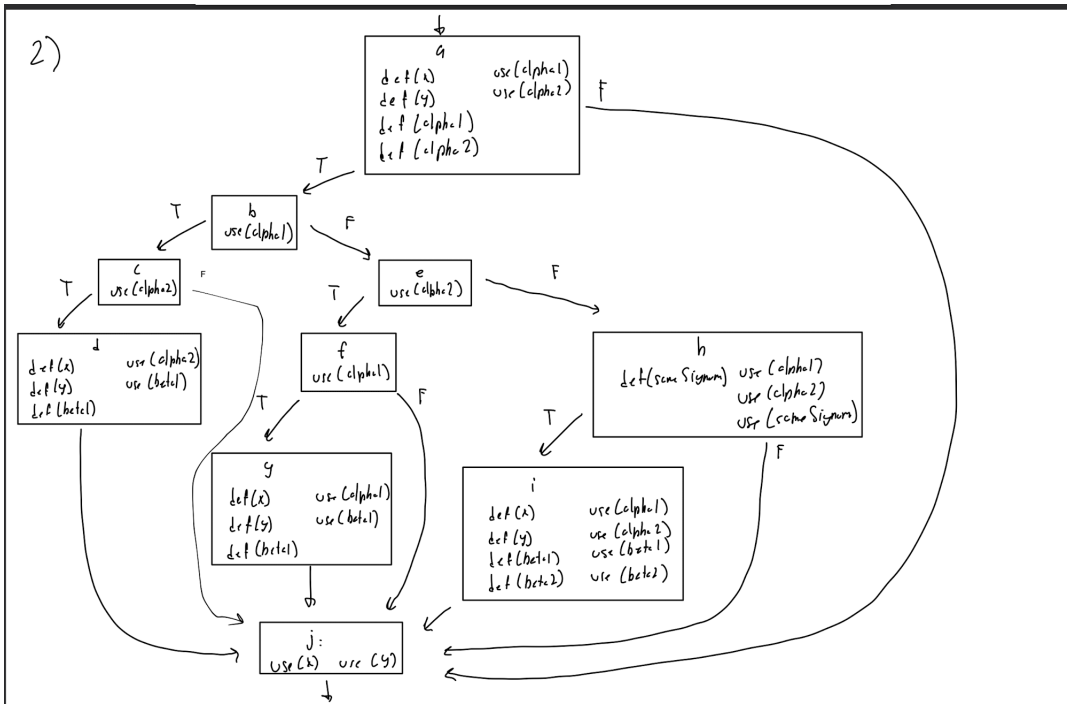
```
b54khan@ubuntu2004-004:~/se465/A2/skeleton/q3$ ./cycle-finder_original.o
Test without cycle: no cycle
Test with cycle (n1 -> n1 -> n1): cycle
b54khan@ubuntu2004-004:~/se465/A2/skeleton/q3$ ./cycle-finder_mutant_1.o
Test without cycle: cycle
Test with cycle (n1 -> n1 -> n1): no cycle
b54khan@ubuntu2004-004:~/se465/A2/skeleton/q3$ ./cycle-finder_mutant_2.o
Test without cycle: cycle
Test with cycle (n1 -> n1 -> n1): cycle
```

The two mutants are not stillborn as they compile. They are not equivalent as they produce different results on the second test. We have that the first case will strongly kill the first mutant. We have that the second test case will strongly kill the second mutant.

Question 4

(a)


1.)
a: 14-16
b: 20
c: 21
d: 22-24
e: 27
f: 28
g: 29-31
h: 34-35
i: 37-40
j: 44
T
F


2)
a
def(x)
def(y)
def(alpha1)
def(alpha2)
use(alpha1)
use(alpha2)
b
use(alpha1)
c
use(alpha2)
d
def(x)
def(y)
def(beta1)
use(alpha2)
use(beta1)
e
use(alpha2)
f
use(alpha1)
g
def(x)
def(y)
def(beta1)
use(alpha1)
use(beta1)
h
def(some Signum)
use(alpha1)
use(alpha2)
use(some Signum)
i
def(x)
def(y)
def(beta1)
def(beta2)
use(alpha1)
use(alpha2)
use(beta1)
use(beta2)
j:
use(x) use(y)
T
F

(b)

$$PPC = \{[a,b,c,d,j],[a,b,c,j],[a,b,e,f,j],[a,b,e,f,g,j],[a,b,e,h,j],[a,b,e,h,i,j],[a,j]\}$$

(c)

$$ADUPC_x = \{[a,b,c,j],[a,b,e,f,j],[a,b,e,h,j],[a,j],[d,j],[g,j],[i,j]\}$$
$$ADUPC_y = \{[a,b,c,j],[a,b,e,f,j],[a,b,e,h,j],[a,j],[d,j],[g,j],[i,j]\}$$
$$ADUPC_{\alpha_1} = \{[a,b],[a,b,e,f],[a,b,e,f,g],[a,b,e,h],[a,b,e,h,i]\}$$
$$ADUPC_{\alpha_2} = \{[a,b,c],[a,b,c,d],[a,b,e],[a,b,e,h],[a,b,e,h,i]\}$$

(d)

First off, PPC subsumes ADUPC (as all paths in the CFG are simple). ADUPC has the advantage it can be used to test the program for the (much fewer than PPC) important cases for when variables are read and assigned to, but it doesn't give you the same guarantees of testing all possible paths through the program as PPC. PPC also has the advantage of requiring fewer test cases to cover the same amount of paths as ADUPC, which in my opinion makes it a better choice for testing.

(e)

[a, b, c, d, j]:

```
point1 = [0, 0]
point2 = [1, 5000]
point3 = [0, 0]
point4 = [1, 1]
sameSignum = NULL (we dont reach this line)
alpha_1 = (5000 - 0) / (1 - 0) = 5000
alpha_2 = (1 - 0) / (1 - 0) = 1
beta_1 = NULL (we dont reach this line)
beta_2 = 1 - 1 * 1 = 0
x = 0
y = 1 * 0 + 0 = 0
```

[a, b, c, j]:

```
point1 = [0, 0]
point2 = [1, 5000]
point3 = [0, 0]
point4 = [1, 6000]
sameSignum = NULL (we dont reach this line)
alpha_1 = (5000 - 0) / (1 - 0) = 5000
alpha_2 = (6000 - 0) / (1 - 0) = 6000
beta_1 = NULL (we dont reach this line)
beta_2 = NULL (we dont reach this line)
x = 1
y = 5000
```

[a, b, e, f, j]:

```
point1 = [0, 0]
point2 = [1, 4000]
point3 = [0, 0]
point4 = [1, 6000]
sameSignum = NULL (we dont reach this line)
alpha_1 = (4000 - 0) / (1 - 0) = 4000
alpha_2 = (6000 - 0) / (1 - 0) = 6000
beta_1 = NULL (we dont reach this line)
beta_2 = NULL (we dont reach this line)
x = 1
y = 4000
```

[a, b, e, f, g, j]:

```
point1 = [0, 0]
point2 = [1, 1]
point3 = [0, 0]
point4 = [1, 6000]
sameSignum = NULL (we dont reach this line)
alpha_1 = (1 - 0) / (1 - 0) = 1
alpha_2 = (6000 - 0) / (1 - 0) = 6000
beta_1 = 1 - 1 * 1 = 0
beta_2 = NULL (we dont reach this line)
x = 0
y = 1 * 0 + 0 = 0
```

[a, b, e, h, j]:

```
point1 = [0, 0]
point2 = [1, -1]
point3 = [0, 0]
point4 = [1, 1]
sameSignum = False
alpha_1 = (-1 - 0) / (1 - 0) = -1
alpha_2 = (1 - 0) / (1 - 0) = 1
beta_1 = NULL (we dont reach this line)
beta_2 = NULL (we dont reach this line)
x = 0
y = -1
```

[a, b, e, h, i, j]:

```
point1 = [0, 0]
point2 = [1, 2]
point3 = [0, 0]
point4 = [1, 1]
sameSignum = True
```

```
alpha_1 = (2 - 0) / (1 - 0) = 2
alpha_2 = (1 - 0) / (1 - 0) = 1
beta_1 = 2 - 2 * 1 = 0
beta_2 = 1 - 1 * 1 = 0
x = (0 - 0) / (2 - 1) = 0
y = 2 * 0 + 0 = 0
```

[a, j]:

```
point1 = [0, 0]
point2 = [1, 1]
point3 = [0, 0]
point4 = [1, 1]
sameSignum = NULL (we dont reach this line)
alpha_1 = (1 - 0) / (1 - 0) = 1
alpha_2 = (1 - 0) / (1 - 0) = 1
beta_1 = NULL (we dont reach this line)
beta_2 = NULL (we dont reach this line)
x = 1
y = 1
```