David Dae Ho Kim
20415096
CS 447 - 001
dhdkim@uwaterloo.ca

Question 1
For an acyclic graph, Prime Path Coverage (PPC) does subsume Complete Path Coverage (CPC).

Proof:

First, we must prove that any test set that satisfies PPC covers all simple paths. PPC requires that the TR must contain all prime paths. We will prove this by contradiction. Let's assume that a test set T satisfies PPC but does not cover all simple paths. We will say that T does not cover the simple path P1. Path P1 must either be a prime path or a proper subpath of a prime path. This means that T must also cover P1 since P1 is a proper subpath of P2. By definition T must satisfy all prime paths so it will cover P1. If P1 is a proper subpath of P2, then T will cover P2 and thus also cover P1. This is a contradiction. Therefore T covers P1 and covers all simple paths.

Second, we must prove that all paths in an acyclic graph, all paths are simple. In an acyclic graph, no path contains a cycle. This means that no nodes will appear more than once. By definition of a simple path, this is a simple path. Therefore all paths in an acyclic graph are simple.

QED.

Question 2

a)
We know that the structure of the class *node* is has a *char** to hold the name. Running:

valgrind --leak-check=full -v

reveals that 9 bytes are still un-freed with 5 allocs and 4 frees. Valgrind also reports that the error exists in the function *fgets_enhanced ln:47* initiated by *main ln:279*. From this, we gather the information that it must be something to do with the names. After doing a comparison between *delete_node* and *delete_all,* I have noticed that *delete_node* does not free up the *char * str* when deleting the node. So this was the fault, the name of each node was not being freed. The line *free(temp->str)* before the line *free(temp)* will fix the memory leak.

b)
When running the test through valgrind, the output showed us that there were 10 allocs, but 16 frees. This tells us that there were too many free calls happening. To investigate, I checked the test case and the source code again and I noticed that *delete_all* was being called in succession. I quickly investigated this issue because I thought this was the problem. The problem was that p was not set to *NULL* after *delete_all* was called, so on the second *delete_all*

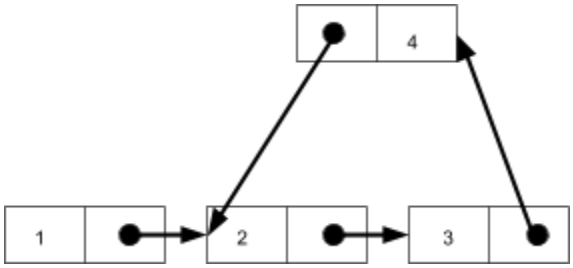
call, we tried to free up memory that was already freed because of the *while(temp!=NULL)* condition. If we set *p = NULL* at the end of each *delete_all* call, we will get the correct number of free calls.

c)
Inserting a new node, then trying to duplicate the node causes a buffer overflow bug. This is because the function *strcpy* must have its destination size large enough to copy the C-string of the source string (including the null terminator). So adding +1 to the length when the string is non-zero will solve this issue.

Question 3

Mutant 1:
Insert *list=fast* after line 33 before line 34.

Test case to strongly kill the mutant:



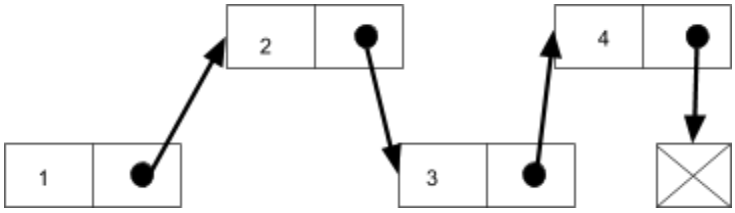
Given the above test case, this is the output when node with data 1 is passed:

Expected Output: 1
Mutant Output: *Infinite Loop*
Actual Output: *Infinite Loop*

This is because after all of the *if* checks are finished, we are letting list equal to fast. This will cause the slower pointer to catch up to the fast pointer after each iteration. Therefore this will cause the loop to be executed infinitely.

Mutant 2:
Insert *list=fast* after line 29 before line 30.

Test case to strongly kill the mutant:



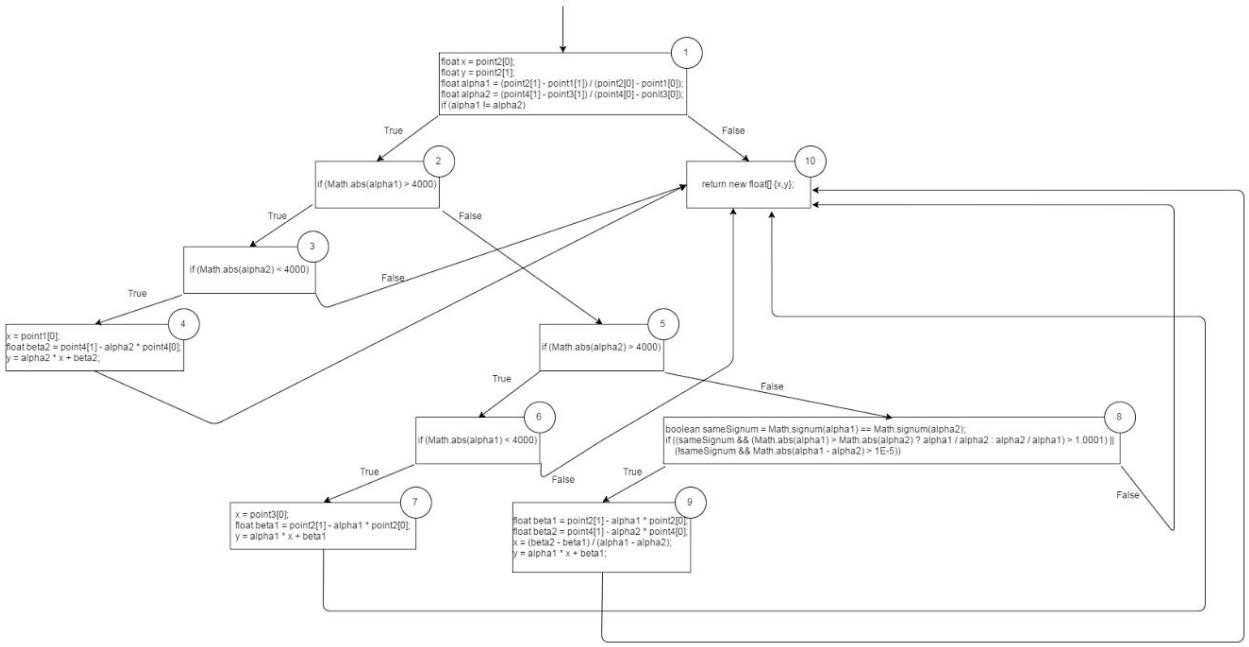
Given the above test case, this is the output when node with data 1 is passed:

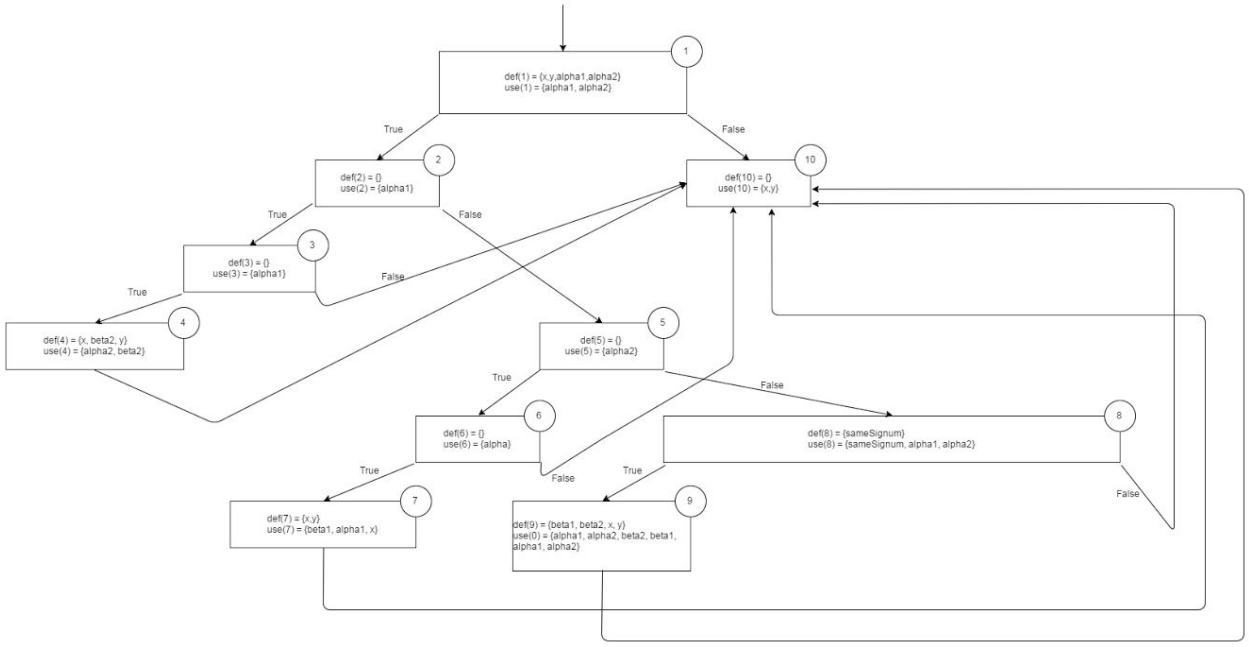
Expected Output: 0
Mutant Output: 1
Actual Output: 1

This is because on the second *if* case, there is a check for *fast* == *list*. Right before this *if* is executed, we force list to equal fast so this *if* statement will always resolve to *true*.

Question 4

a)


1
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)
True
False
2
if (Math.abs(alpha1) > 4000)
10
return new float[] {x,y};
True
False
3
if (Math.abs(alpha2) < 4000)
True
False
4
x = point1[0];
float beta2 = point4[1] - alpha2 * point4[0];
y = alpha2 * x + beta2;
5
if (Math.abs(alpha2) > 4000)
True
False
6
if (Math.abs(alpha1) < 4000)
8
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))
True
False
True
False
7
x = point3[0];
float beta1 = point2[1] - alpha1 * point2[0];
y = alpha1 * x + beta1
9
float beta1 = point2[1] - alpha1 * point2[0];
float beta2 = point4[1] - alpha2 * point4[0];
x = (beta2 - beta1) / (alpha1 - alpha2);
y = alpha1 * x + beta1;


1
def(1) = {x,y,alpha1,alpha2}
use(1) = {alpha1, alpha2}
True
False
2
def(2) = {}
use(2) = {alpha1}
10
def(10) = {}
use(10) = {x,y}
True
False
3
def(3) = {}
use(3) = {alpha1}
False
True
4
def(4) = {x, beta2, y}
use(4) = {alpha2, beta2}
5
def(5) = {}
use(5) = {alpha2}
True
False
6
def(6) = {}
use(6) = {alpha}
8
def(8) = {sameSignum}
use(8) = {sameSignum, alpha1, alpha2}
True
False
True
False
7
def(7) = {x,y}
use(7) = {beta1, alpha1, x}
9
def(9) = {beta1, beta2, x, y}
use(0) = {alpha1, alpha2, beta2, beta1,
alpha1, alpha2}

b)

TR_{ppc} = { [1,10], [1,2,3,10], [1,2,3,4,10], [1,2,5,6,10], [1,2,5,6,7,10], [1,2,5,8,10], [1,2,5,8,9,10] }

c)
TR_{ADUPC} = {
[1,2,],[1,2,5,6],[1,2,5,6,7],
[1,2,5,8],[1,2,5,8,9],[1,2,3],
[1,2,3,4],[1,2,5],[1,10],
[4,10],[7,10],[9,10],
[1,2,3,10],[1,2,5,6,10],[1,2,5,8,10]
}

Work:
du(1,alpha1) = {[1,2], [1,2,5,6], [1,2,5,6,7], [1,2,5,8], [1,2,5,8,9]}
du(1,alpha2) = {[1,2,3], [1,2,3,4], [1,2,5], [1,2,5,8], [1,2,5,8,9]}
du(1,x) = {[1,10], [1,2,3,10], [1,2,5,6,10], [1,2,5,8,10]}
du(1,y) = {[1,10], [1,2,3,10], [1,2,5,6,10], [1,2,5,8,10]}
du(4,x) = {[4,10]}
du(4,y) = {[4,10]}
du(7,x) = {[7,10]}
du(7,y) = {[7,10]}
du(9,x) = {[9,10]}
du(9,y) = {[9,10]}

d)
Strengths:

ADUPC contains very detailed information on the definitions and the uses of all the variables and the data that the program is using. It is possible to find anomalies such as feasible paths to a use of a certain variable that contains no other definition of the variable.

PPC allows us to test the program since PPC contains only prime paths. By definition, these will not be subpaths and these are also maximal simple paths. It is easy to to test the functionalities of a program with PPC.

Weaknesses:

ADUPC requires more effort to generate for all definitions and uses. There needs to be a reference between calls from other functions that passes variables and uses variables that were passed from other sources.

PPC is time consuming to find all the prime path test cases for the entire program. There also lacks the ability to identify problems with the variables themselves.

It depends on the type of program and the complexity of the program to determine if additional

number of test cases is worth it. If the time and effort that is consumed can outweigh the weaknesses of a criterion, then it will be worth it in that case. For complex programs, it may not be worth it since the time consumed will likely not be able to outweigh the strengths.

We do know that PPC subsumes ADUPC. This means that every set of test cases that satisfies PPC also satisfies ADUPC. If we have PPC, then we do not need additional test cases to satisfy ADUPC.

e)
TR_{ppc} = {[1,10], [1,2,3,10], [1,2,3,4,10], [1,2,5,6,10], [1,2,5,6,7,10], [1,2,5,8,10], [1,2,5,8,9,10]}

	point1	point 2	point 3	point 4	alpha 1	alpha 2	sameSignum	Expected output
[1,10]	1,1	2,2	1,1	2,2	1	1	/	[2.0,2.0]
[1,2,3,10]	0,1	1, 5000	0,1	1,5003	4999	5002	/	[1.0, 5000.0]
[1,2,3,4,10]	0,1	1, 5001	0,1	1,51	5000	50	/	[0.0, 1.0]
[1,2,5,6,10]	0,1	1,4001	0,1	1,5001	4000	5000	/	[1.0, 4001.0]
[1,2,5,6,7,10]	0,1	1,2001	0,1	1,7001	2000	7000	/	[0.0, 1.0]
[1,2,5,8,10]	2,1	95000, 95000	0,0	1,1	1.0000105	1	true	[95000.0, 95000.0]
[1,2,5,8,9,10]	0,1	1,503	0,603	1,1	502	-602	false	[0.55, 274.74]