Christian Ayala
Algorithms HW3

Question 1 Page 1

To build the data structure, we must save two pieces of information: the start time of an event, and the stop time. We need to be sure that we maintain the proper order of events as well. With that in mind, we create two lists using the following algorithm:

```
Initialize 2 lists, S and F
for each item E in Event List
    if E not in S                       <- E is the first occurence
                                           of the interval
        add E_t to end of S
    else                                <- 2nd occurence of E
        add E_t to end of F, link F and S
    end if
end for
```

Since we only do a single traversal of the event list, we see that this algorithm is $O(n)$

With the event list: ZWXTZVYYXQWUVSUTRSRQ we can build the data structure as such:

S: Z_0 W_1 X_2 T_3 V_4 Y_5 Q_6 U_7 S_8 R_9

F: Z_3 Y_6 X_6 W_7 V_7 U_8 T_8 S_9 R_{10} Q_{10}

Where the sequence in S defines the order in which they start, and F defines the order in which they end. Note that they are not in the same order, and I have arbitrarily chosen t times such that $t = 0 \dots n$, and an event does not start and stop at the same time t_x.

Question 1 Page 2

Now that we have a completed data structure, we can use our event list data structure to determine coloring using the following algorithm:

```
Initialize a set of Colors, C
Initialize i = 1, j = 1
For t = 0, .., n
    While (F_j.time == t)
        Restore F_j.color to set C
        j <- j + 1
    End While
    While (S_i.time == t)
        C_k = First available color of C
        Apply color C_k to S_i and corresponding finish time
        Remove C_k
        i <- i + 1
    End While
End For
```

In this case, we are doing a complete traversal of both the starting list and the finish list. Since each list contains n elements, we have $2n$ operations, or $O(n)$. In addition, we only apply color information to the n start times, which is $O(n)$ additional space.

Question 1 Page 3

We will now use the algorithm to apply colors to:

$S: Z_0 \; W_1 \; X_2 \; T_3 \; V_4 \; Y_5 \; Q_6 \; U_7 \; S_8 \; R_9$

$F: Z_3 \; Y_6 \; X_6 \; W_7 \; V_7 \; U_8 \; T_8 \; S_9 \; R_{10} \; Q_{10}$

We start at $t=0$. We see that F_1.time is 3, so we don't restore any colors. However, S_1.time is $\emptyset$, so we apply the first available color to S_1. (Red)

$S: Z_0 \; W_1 \; X_2 \; T_3 \; V_4 \; Y_5 \; Q_6 \; U_7 \; S_8 \; R_9$

$F: Z_3 \; Y_6 \; X_6 \; W_7 \; V_7 \; U_8 \; T_8 \; S_9 \; R_{10} \; Q_{10}$

Continuing through $t=1$ and $t=2$, we apply new colors to S_2 and S_3 (blue) and (green)

$S: Z_0 \; W_1 \; X_2 \; T_3 \; V_4 \; Y_5 \; Q_6 \; U_7 \; S_8 \; R_9$

$F: Z_3 \; Y_6 \; X_6 \; W_7 \; V_7 \; U_8 \; T_8 \; S_9 \; R_{10} \; Q_{10}$

When $t=3$, j is still 1, and we see that F_1.time is 3. We then restore the first used color, which was red. We then apply it to S_4

$S: Z_0 \; W_1 \; X_2 \; T_3 \; V_4 \; Y_5 \; Q_6 \; U_7 \; S_8 \; R_9$

$F: Z_3 \; Y_6 \; X_6 \; W_7 \; V_7 \; U_8 \; T_8 \; S_9 \; R_{10} \; Q_{10}$

With $t=4$ and $T=5$, we continue to use new colors, pink and yellow

$S: Z_0 \; W_1 \; X_2 \; T_3 \; V_4 \; Y_5 \; Q_6 \; U_7 \; S_8 \; R_9$

$F: Z_3 \; Y_6 \; X_6 \; W_7 \; V_7 \; U_8 \; T_8 \; S_9 \; R_{10} \; Q_{10}$

For $t=6$, we restore green and yellow, and apply green

$S: Z_0 \; W_1 \; X_2 \; T_3 \; V_4 \; Y_5 \; Q_6 \; U_7 \; S_8 \; R_9$

$F: Z_3 \; Y_6 \; X_6 \; W_7 \; V_7 \; U_8 \; T_8 \; S_9 \; R_{10} \; Q_{10}$

Question 1 Page 4

$t=7$, so we restore blue and pink, and apply blue

$S:\ Z_0\ W_1\ X_2\ T_3\ V_4\ Y_5\ Q_6\ U_7\ S_8\ R_9$

$F:\ Z_3\ Y_6\ X_6\ W_7\ V_7\ U_8\ T_8\ S_9\ R_{10}\ Q_{10}$

$t=8$, we restore blue and red, and apply red

$S:\ Z_0\ W_1\ X_2\ T_3\ V_4\ Y_5\ Q_6\ U_7\ S_8\ R_9$

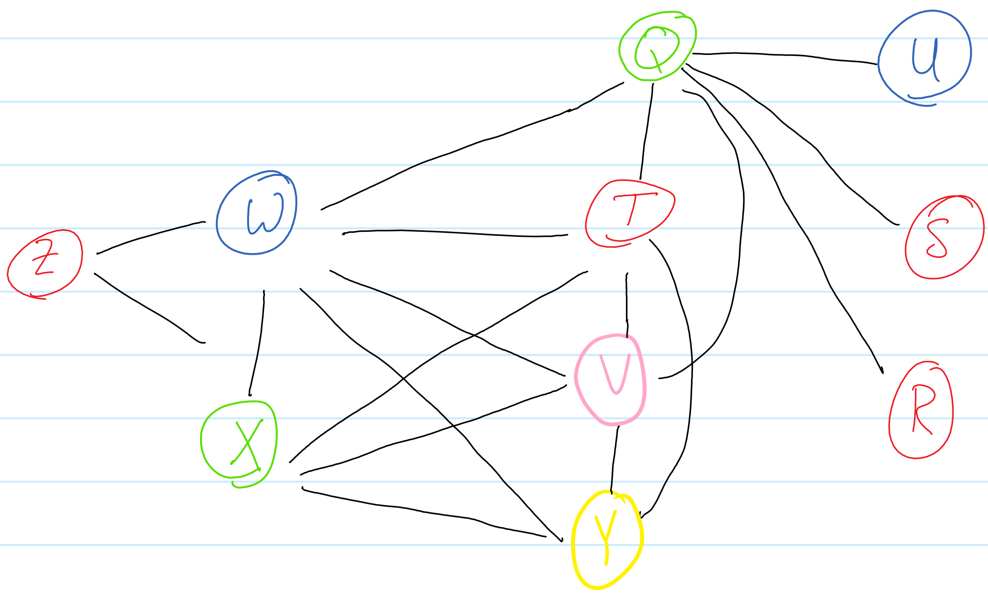
$F:\ Z_3\ Y_6\ X_6\ W_7\ V_7\ U_8\ T_8\ S_9\ R_{10}\ Q_{10}$

$t=9$, restore red and apply red

$S:\ Z_0\ W_1\ X_2\ T_3\ V_4\ Y_5\ Q_6\ U_7\ S_8\ R_9$

$F:\ Z_3\ Y_6\ X_6\ W_7\ V_7\ U_8\ T_8\ S_9\ R_{10}\ Q_{10}$

We have now colored the start times, and we required 5 distinct colors. We can show that these 5 colors are non-overlapping by looking at the interval graph: Since no edge connects two nodes of the same color, our algorithm has produced a valid interval graph:



Question 1 Page 5

To find the degree of all vertices, we use our structure to examine 2 values: The number of nodes that arrive before a node leaves, and the number of nodes that have left before it arrives. We set up two additional storage units of size $|V|$, one for each of the above values. Then, we perform the following algorithm:

Initialize $A \leftarrow 0$
For $i = 1 .. 2n$
    $E_i.A \leftarrow A$
    If (E_i is start time)
        $A \leftarrow A+1$
    End if
End For

With our event list, we have the following structure:

E:	Z	W	X	T	Z	V	Y	Y	X	Q	W	U	V	S	U	T	R	S	R	Q
A:	0	1	2	3	3	4	5	5	5	6	6	7	7	8	8	8	9	9	9	9

Then, we subtract start time's A from the end time's A for each vertex pair.

Question 2 Page 1

Question 2a)

For this question, one can see that with 4 binary digits, there are $2^4 = 16$ possible strings that can be generated. The goal to get the maximum length is therefore achieved if each substring $(b_i b_{i+1} b_{i+2} b_{i+3})$ represents a unique binary string of 4 digits. Since there are 16 combinations, $i = 16$, and the max length string therefore contains $i+3 = 16+3 = \underline{19}$ digits.

In fact, an example string that satisfies this constraint is shown below:

1111 0110 0101 0000 111

And we can demonstrate this by iterating over $i = 1 \ldots 16$, and showing that this string contains all 16 combinations of length 4 digits, but no more:

i	Sequence	i	Sequence	Sequence	i	Sequence	i
1	1111	9	0101	0000	13	1000	12
2	1110	10	1010	0001	14	1001	7
3	1101	11	0100	0010	8	1010	10
4	1011	12	1000	0011	15	1011	4
5	0110	13	0000	0100	11	1100	6
6	1100	14	0001	0101	9	1101	3
7	1001	15	0011	0110	5	1110	2
8	0010	16	0111	0111	16	1111	1

Adding an extra digit anywhere in the string will create a matching substring, so the max length is $\underline{19}$

Question 2ii)

For this problem, I assume that we already have a suffix tree built for string B, and the algorithm will only consider the complexity of finding the longest matching substring in B with the given tree.

We can define a maximal pair in B as b_i and b_j, such that the characters to the immediate left/right of b_i are different than those surrounding b_j. If we were to extend both b_i or b_j in either direction, they would no longer be equal.

To define the algorithm, we first define the concept of left-diverse nodes. Given a tree T, an internal node v is said to be left-diverse if at least 2 leaves in v's subtree have different left-most characters. We assume that leaf nodes are not left-diverse because they have no subtrees. If a node is left diverse, then the diversity propagates upward to its parent node in tree T. If a node is left-diverse, then the path-label to that node is a maximal pair.

Once we have found the set of all maximal pairs, then we simply find the node with the longest path-label length, and that is our longest matching substring.

Question 2 Page 3

Now that we have defined the necessary terminology, the algorith for finding all left-diverse nodes is as follows:

```
Traverse T bottom-up            //a.k.a. starting at leaves
    If node v is a leaf
        Record its left-most character
    Else if node v is an internal node
        If any child of v is left-diverse
            Mark v as left diverse
            Prepend v to the longest left-diverse child
            Save previous concatination in v
        Else if all children have a common left char x
            Record x for v
        Else
            Mark v as left-diverse
        End if
    End if
End Traversal
```

Building the suffix tree can be done in $O(n)$ time (I will not go over the algorithm here, but an algorithm like Ukkonen's Algorithm has been proven to build a tree in $O(n)$ time and space). The bottom-up traversal of the tree is also $O(n)$. We can observe that, for a given string, T can have at most n internal nodes. T has n leaves (one for each possible suffix), and since each internal node has ≥ 2 children, we know that the upper limit of internal nodes is n. Therefore, a bottom-up traversal is $O(n)$. Saving the running longest substring is also $O(n)$.

To demonstrate the algorithm, consider the number π.

$\pi = 3.11037552421026430215_8$

This number has 49 binary bits, as such:

$\pi = 11.001001000011111101101010100010001000010110100011$

0123456789 0123456789 0123456789 0123456789 012345678

Indices +(0*10) +(1*10) +(2*10) +(3*10) +(4*10)

Because it is such a large number with so many prefixes, the suffix tree has been programmatically generated, and is shown in the next two pages

Question 2 Page 5

Left half of the suffix tree.

```
└── (0)
    ├── (5) 0 # 010001000
    │   ├── (15) 0 # 001000
    │   │   ├── (17) 0 # 01000
    │   │   │   ├── (67) 01 # 01
    │   │   │   │   ├── (68) 0110100011$
    │   │   │   │   └── (16) 111101101010100010001000010110100011$
    │   │   │   └── (51) 1 # 1000
    │   │   │       ├── (69) 0 # 000
    │   │   │       │   ├── (59) 00 # 00
    │   │   │       │   │   ├── (60) 010110100011$
    │   │   │       │   │   └── (52) 1000010110100011$
    │   │   │       │   └── (70) 110100011$
    │   │   │       └── (87) 1 # 1
    │   │   │           ├── (88) $
    │   │   │           └── (18) 11101101010100010001000010110100011$
    │   │   └── (19) 1 # 100001
    │   │       ├── (71) 0 # 00001
    │   │       │   ├── (9) 0 # 0001
    │   │       │   │   ├── (53) 0 # 001
    │   │       │   │   │   ├── (61) 01 # 01
    │   │       │   │   │   │   ├── (62) 0110100011$
    │   │       │   │   │   │   └── (10) 111101101010100010001000010110100011$
    │   │       │   │   │   └── (54) 1000010110100011$
    │   │       │   │   └── (4) 100001111101101010100010001000010110100011$
    │   │       │   └── (72) 110100011$
    │   │       └── (89) 1 # 1
    │   │           ├── (90) $
    │   │           └── (20) 11101101010100010001000010110100011$
    │   └── (21) 1 # 10001000
    │       ├── (39) 0 # 0001000
    │       │   ├── (11) 0 # 001000
    │       │   │   ├── (47) 0 # 01000
    │       │   │   │   ├── (63) 01 # 01
    │       │   │   │   │   ├── (64) 0110100011$
    │       │   │   │   │   └── (12) 111101101010100010001000010110100011$
    │       │   │   │   └── (83) 1 # 1000
    │       │   │   │       ├── (55) 000 # 000
    │       │   │   │       │   ├── (56) 010110100011$
    │       │   │   │       │   └── (48) 1000010110100011$
    │       │   │   │       └── (84) 1$
    │       │   │   └── (6) 100001111101101010100010001000010110100011$
    │       │   └── (73) 1 # 10
    │       │       ├── (43) 0 # 0
    │       │       │   ├── (44) 0010001000010110100011$
    │       │       │   └── (40) 100010001000010110100011$
    │       │       └── (74) 10100011$
    │       └── (33) 1 # 1010
    │           ├── (77) 010 # 010
    │           │   ├── (78) 0011$
    │           │   └── (34) 10100010001000010110100011$
    │           └── (22) 11101101010100010001000010110100011$
```

Question 2 Page 6

Right half of suffix tree

```
└─(2) 1 # 10001000
  ├─(31) 0 # 0001000
  |  ├─(13) 0 # 001000
  |  |  ├─(49) 0 # 01000
  |  |  |  ├─(65) 01 # 01
  |  |  |  |  ├─(66) 0110100011$
  |  |  |  |  └─(14) 111101101010100010001000010110100011$
  |  |  |  └─(85) 1 # 1000
  |  |  |     ├─(57) 000 # 000
  |  |  |     |  ├─(58) 010110100011$
  |  |  |     |  └─(50) 1000010110100011$
  |  |  |     └─(86) 1$
  |  |  └─(7) 100 # 100
  |  |     ├─(8) 001111101101010100010001000010110100011$
  |  |     └─(3) 100001111101101010100010001000010110100011$
  |  └─(37) 1 # 10001
  |     ├─(45) 0 # 0001
  |     |  ├─(81) 001 # 001
  |     |  |  ├─(46) 0001000010110100011$
  |     |  |  └─(82) 1$
  |     |  └─(41) 10 # 10
  |     |     ├─(42) 0010001000010110100011$
  |     |     └─(38) 100010001000010110100011$
  |     └─(75) 1010 # 1010
  |        ├─(76) 0011$
  |        └─(32) 10100010001000010110100011$
  └─(23) 1 # 1010
     ├─(29) 0 # 010
     |  ├─(1) 0100100001111101101010100010001000010110100011$
     |  └─(35) 1 # 10
     |     ├─(79) 0 # 0
     |     |  ├─(80) 0011$
     |     |  └─(36) 10100010001000010110100011$
     |     └─(30) 101010100010001000010110100011$
     └─(27) 1 # 11
        ├─(28) 01101010100010001000010110100011$
        └─(25) 1 # 1
           ├─(26) 01101010100010001000010110100011$
           └─(24) 101101010100010001000010110100011$
```

Question 2 Page 7

In looking at the algorithm, the first step is to start at the leaves of the tree, and mark them all as not left-diverse. These are marked with a green arrow (⟵). Next, we move up the tree to the internal nodes, and begin marking nodes that are left-diverse. These are marked with red pound signs (#). As it turns out, since this is a binary string with only 2 possible characters, every internal node is marked as left-diverse. At this point, we simply find the path with the most left-diverse nodes, and the longest concatination of vertex edge labels is our longest matching substring. Each # has the concatination of all vertex edge labels, leading up to that vertex, marked in blue text.

What we find is that, according to our suffix tree, our longest matching substring is 010001000 , or nine characters long. Indeed, we can find this string starting at index 24 and index 28, and indeed we have no longer matching substrings. In the example above, we require $O(n^2)$ space to store the running longest substrings at each internal node. However, we can optimize this to $O(n)$ by the following logic: if we need to prepend the current node to the current longest substring, then append it, store it in the node, and delete the current longest substring from each child node. At the end of the algorithm, only the root node will contain a subsequence, bounded in space by $O(n)$. Also, we have seen that time complexity is $O(n)$ as well, requiring a single traversal of all n nodes in the tree.

Question 3 Page 1

Question 3i)

In the case of a generic graph, this problem is actually NP-Complete. However, since we are finding the maximum independent set (MIS) of a tree, we can make the following assumptions:

1) Our tree is directed
2) The tree does not contain any cycles
3) Except for the root node, a node has exactly 1 parent, but can have ≥ 0 children.

With that in mind, we make the assertion that, since a node has only a single parent, there are exactly 2 independent sets that contain all the nodes in the tree, and one of these 2 sets is the MIS. The strategy is such that we alternate colors when going from root to child.

We can determine colors using the algorithm on the next page, which a modification on a standard breadth-first search.

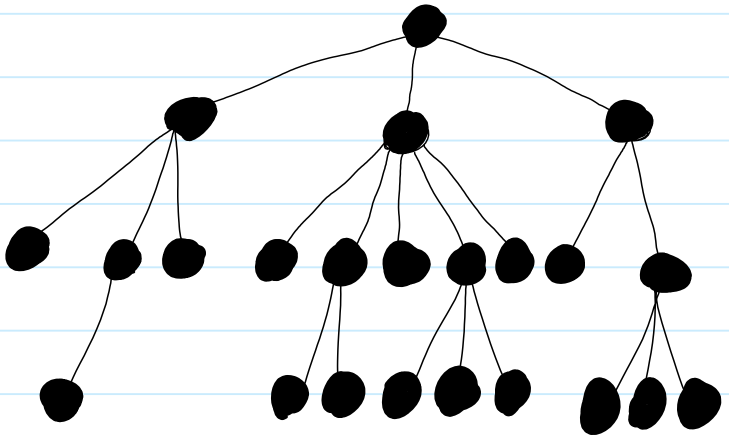
Question 3 Page 2

```
Initialize queue Q
Initialize 2 sets, Red and Blue
Enqueue root node onto Q
While Q is not empty
    v ← Q.dequeue
    If v is the root
        Color v red
        Add v to Red
    Else if v.parent.color is red
        Color v blue
        Add v to Blue
    Else
        Color v red
        Add v to Red
    End If
    For all children C in v.children
        Enqueue C onto Q
    End For
End While
If Red.Count > Blue.Count
    MaxSet ← Red
Else
    MaxSet ← Blue
Return MaxSet
```

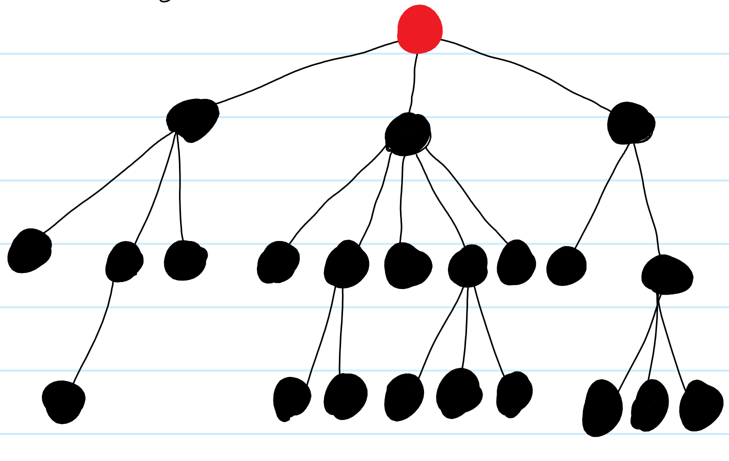
Question 3 Page 3

When the algorithm finishes, MaxSet contains the maximum independent set in the tree. Since a node cannot have the same color as its single parent, we know that the MaxSet contents are indeed independent. In the worst case the queue Q will require $O(|V|)$ space, since each vertex is visited once. In addition, the two sets Red and Blue requires a combined space of $|V|$ elements, which means the algorithm requires a total $O(|V|)$ space. Since all vertices in V are visited exactly once, this algorithm operates in $O(|V|)$ time.

We can demonstrate the algorithm with the following tree:

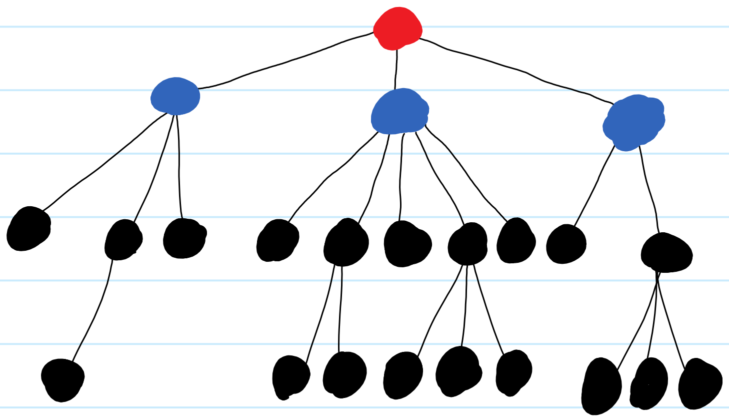


Starting the algorithm, we color the root node red:

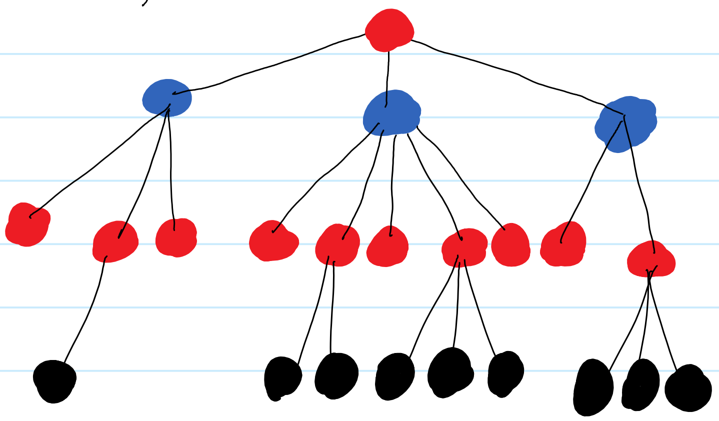


Question 3 Page 4

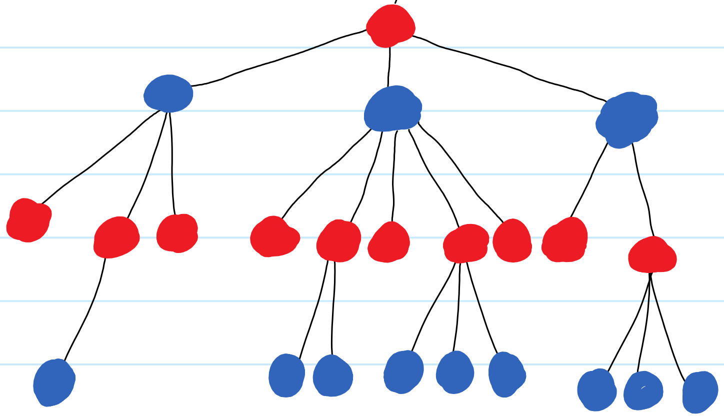
Next, we enqueue all the children of the root node. Each of the children get colored with blue, because the parent node is red:



Similarly, for the next row of children, we color them red because the parent is blue



Finally, the bottom row is painted blue:



Question 3 Page 5

Now that the queue is empty, we have traversed the tree and all nodes belong to either the Red or Blue set. We see that $|R| = 11$, and $|B| = 12$, so our maximum independent set is all nodes in the tree colored blue (which are returned in MaxSet)

Question 4 Page 1

Part a) Euclid's Algorithm

The idea of the algorithm is that you start with two positive integers ($a = 40902$, $b = 24140$). Repeatedly create new pairs consisting of the smaller number, and the remainder of the larger number divided by the smaller number. We repeat until a remainder comes up as Ø. At the k'th step, we are finding a quotient q_k and remainder r_k such that:

$$r_{k-2} = q_k r_{k-1} + r_k$$

Step	Results
0	$r_{-2} = 40902$, $r_{-1} = 24140$, $q_0 = 1$, $r_0 = 16762$ $\Rightarrow 40902 = (1 * 24140) + 16762$
1	$r_{-1} = 24140$, $r_0 = 16762$, $q_1 = 1$, $r_1 = 7378$ $\Rightarrow 24140 = (1 * 16762) + 7378$
2	$r_0 = 16762$, $r_1 = 7378$, $q_2 = 2$, $r_2 = 2006$ $\Rightarrow 16762 = (2 * 7378) + 2006$
3	$r_1 = 7378$, $r_2 = 2006$, $q_3 = 3$, $r_3 = 1360$ $\Rightarrow 7378 = (3 * 2006) + 1360$
4	$r_2 = 2006$, $r_3 = 1360$, $q_4 = 1$, $r_4 = 646$ $\Rightarrow 2006 = (1 * 1360) + 646$
5	$r_3 = 1360$, $r_4 = 646$, $q_5 = 2$, $r_5 = 68$ $\Rightarrow 1360 = (2 * 646) + 68$
6	$r_4 = 646$, $r_5 = 68$, $q_6 = 9$, $r_6 = 34$ $\Rightarrow 646 = (9 * 68) + 34$
7	$r_5 = 68$, $r_6 = 34$, $q_7 = 2$, $r_0 = Ø$ $\Rightarrow 68 = (2 * 34) + Ø$

Therefore, the $GCD(40902, 24140) = 34$

Question 4 Page 2

Part b) Binary Normalization Shift-and-Subtract

40902_{10} = 1001 1111 1100 0110, 24140 = 0101 1110 0100 1100

This algorithm operates on the following identities:

1) $gcd(u, 0) = u$, $gcd(0, u) = u$, $gcd(0, 0) = 0$
2) If u and v are both even, then $gcd(u,v) = gcd(u/2, v/2)$
3) If u is even and v is odd, then $gcd(u,v) = gcd(u/2, v)$

3b) If u is add and v is even, then $gcd(u,v) = gcd(u, v/2)$

4) If u and v are odd, then $gcd(u,v) = gcd((u-v)/2, v)$

Repeat 2-4 until u = v or u = 0, and the GCD = $2^k v$, where K is the number of times 2) was satisfied.

Step	u	v	
0	1001 1111 1100 0110	0101 1110 0100 1100	2
1	0100 1111 1110 0011	0010 1111 0010 0110	3b
2	0100 1111 1110 0011	0001 0111 1001 0011	4
3	0001 1100 0010 1000	0001 0111 1001 0011	3a
4	0000 1110 0001 0100	0001 0111 1001 0011	3a
5	0000 0111 0000 1010	0001 0111 1001 0011	3a
6	0000 0011 1000 0101	0001 0111 1001 0011	4
7	0000 1010 0000 0111	0000 0011 1000 0101	4
8	0000 0011 0100 0001	0000 0011 1000 0101	4
9	0000 0000 0010 0010	0000 0011 0100 0001	3a
10	0000 0000 0001 0001	0000 0011 0100 0001	4
11	0000 0001 1001 1000	0000 0000 0001 0001	3a
12	0000 0000 1100 1100	0000 0000 0001 0001	3a
13	0000 0000 0110 0110	0000 0000 0001 0001	3a
14	0000 0000 0011 0011	0000 0000 0001 0001	4
15	0000 0000 0001 0001	0000 0000 0001 0001	= 17

Since 2 was satisfied once, $gcd(40902, 24140) = 2^1 * 17 = 34$

Question 4 Page 3

Question 3iii) Binary Parity "Right-Normalize" and Subtract.
This algorithm is similar to the previous one, except we shift right when a number has a zero in the LSB, until the LSB is a 1. Do this for both numbers, then subtract the two, keeping the two smallest numbers.

	u	V	
Shift right 1	1001 1111 1100 0110	0101 1110 0100 1100	Shift right 2
Subtract	0100 1111 1110 0011	0001 0111 1001 0011	
Shift right 4	0011 1000 0101 0000	0001 0111 1001 0011	
	0000 0011 1000 0101	0001 0111 1001 0011	Swap with u
Shift right 1	0001 0100 0000 1110	0000 0011 1000 0101	
Subtract	0000 1010 0000 0111	0000 0011 1000 0101	
Shift right 1	0000 0110 1000 0010	0000 0011 1000 0101	
	0000 0011 0100 0001	0000 0011 1000 0101	
Subtract	0000 0011 0100 0001	0000 0000 0100 0100	Shift right 2
Subtract	0000 0011 0100 0001	0000 0000 0001 0001	
Shift right 4	0000 0011 0011 0000	0000 0000 0001 0001	
	0000 0000 0011 0011	0000 0000 0001 0001	
Subtract	0000 0000 0010 0010	0000 0000 0001 0001	
Shift 1	0000 0000 0001 0001	0000 0000 0001 0001	= 17

Once again, since there was one common factor of 2 in the first step, we multiply 17*2, for a gcd of 34.

Question 4 Page 4

Question 4) Ternary Parity Right Normalize and subtract.
Once again, the process is similar to the previous algorithm, only in base 3 instead of 2. We still search for 1 or 2 in the LSB position

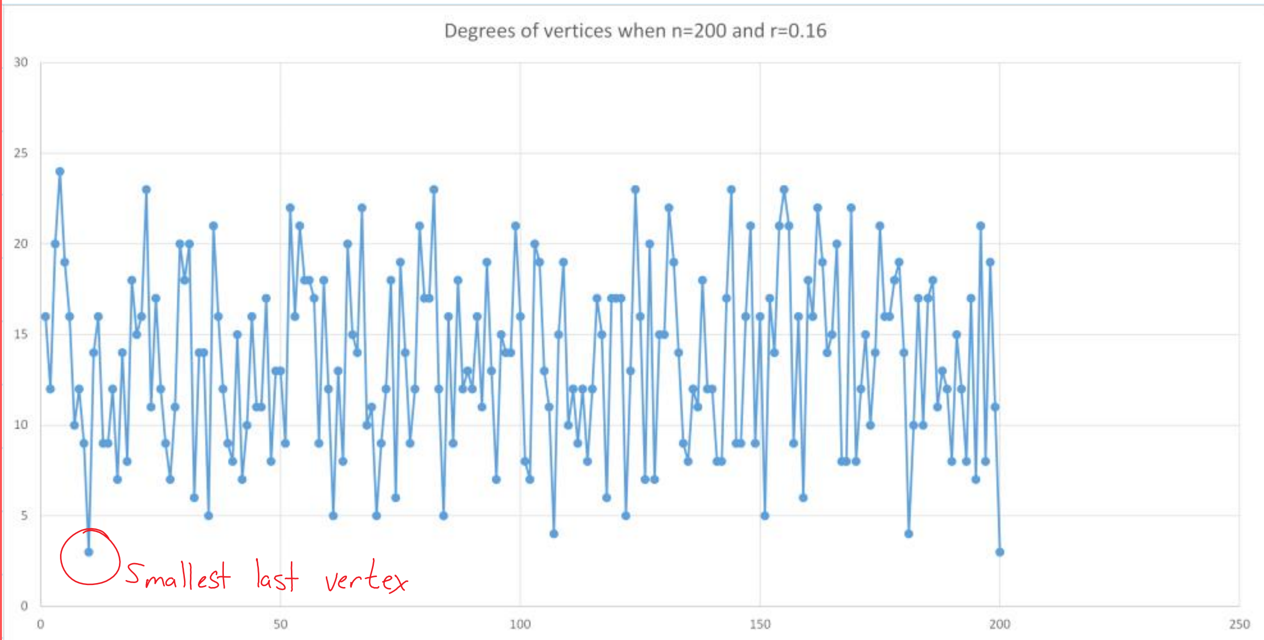
	u	v	
	2002002220	1020010002	
Shift right	200200222	1020010002	
Subtract	2002 0022 2	1121 0201 0	Shift right
	2002 0022 2	1121 0201	
Subtract	1112 2002 1	1121 0201	
Subtract	1000 0212 0	1121 0201	
Shift	1000 0212	1121 0201	
Subtract	1000 0212	1202 212	
subtract	1202212	1021 000	Shift
	1202212	1021	
Subtract	1201121	1021	
Subtract	1200100	1021	
Shift	12001	1021	
Subtract	10210	1021	
Shift	1021	1021	

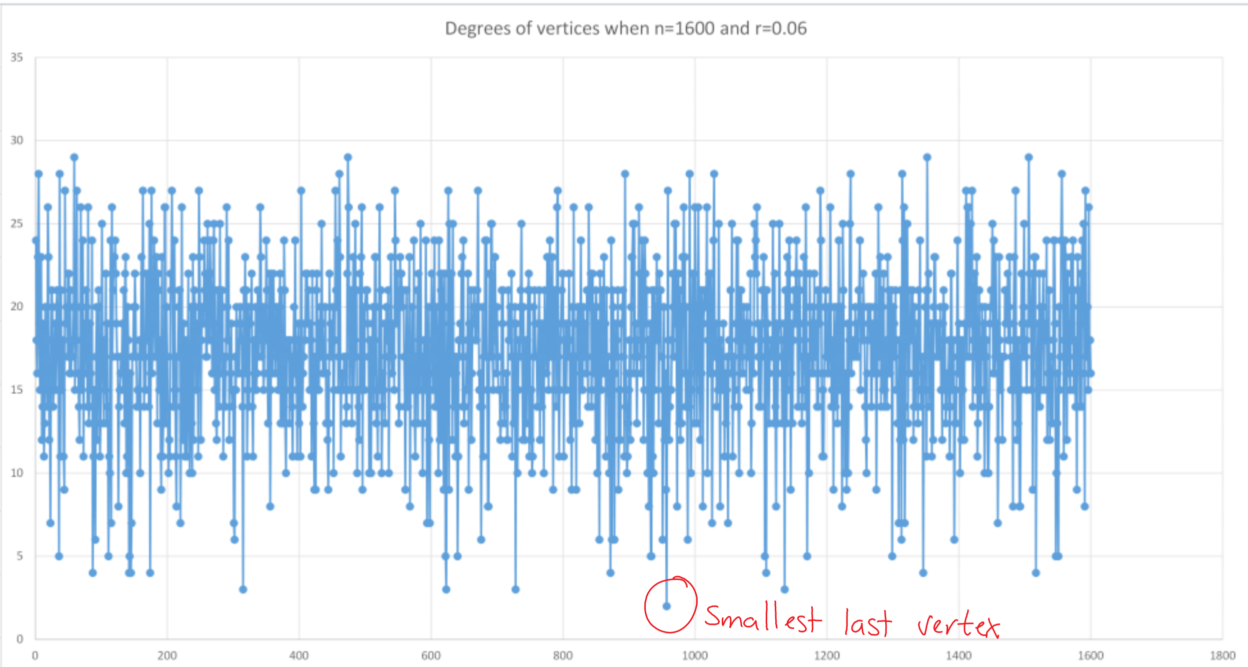
Now that u and v are equal, we have found our gcd of $1021_3 = 34_{10}$.

Question 6 Page 1

Question 6i)

In each of the graphs, the X-axis represents the point number (in order of when they were generated). The y axis represents the degree of that vertex.

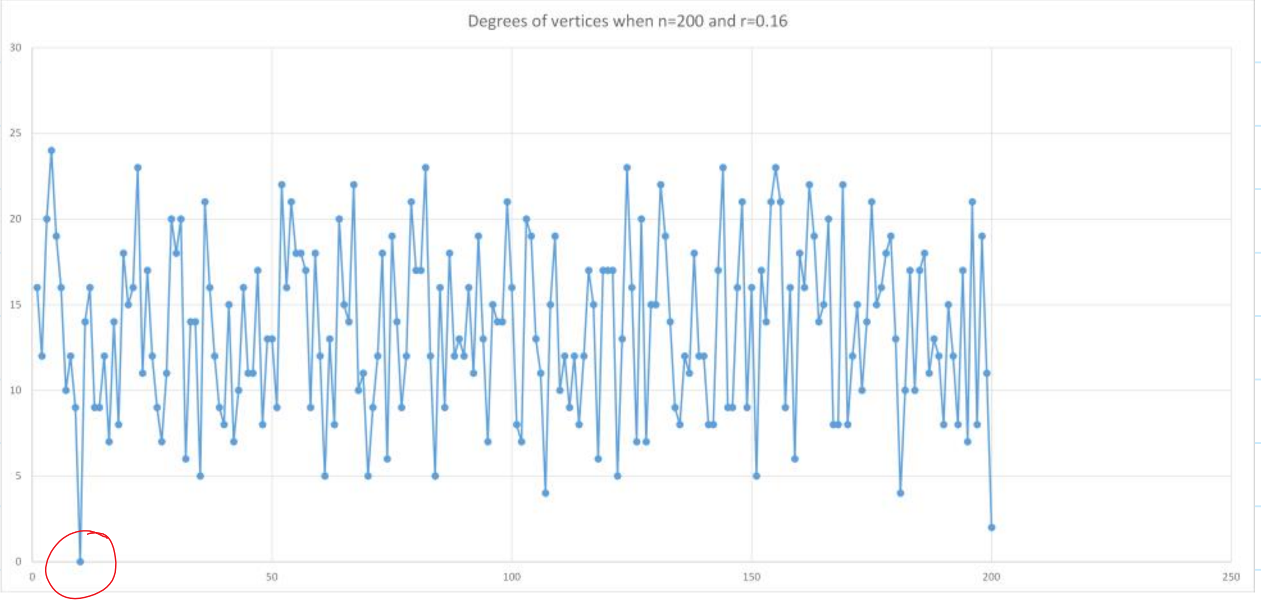


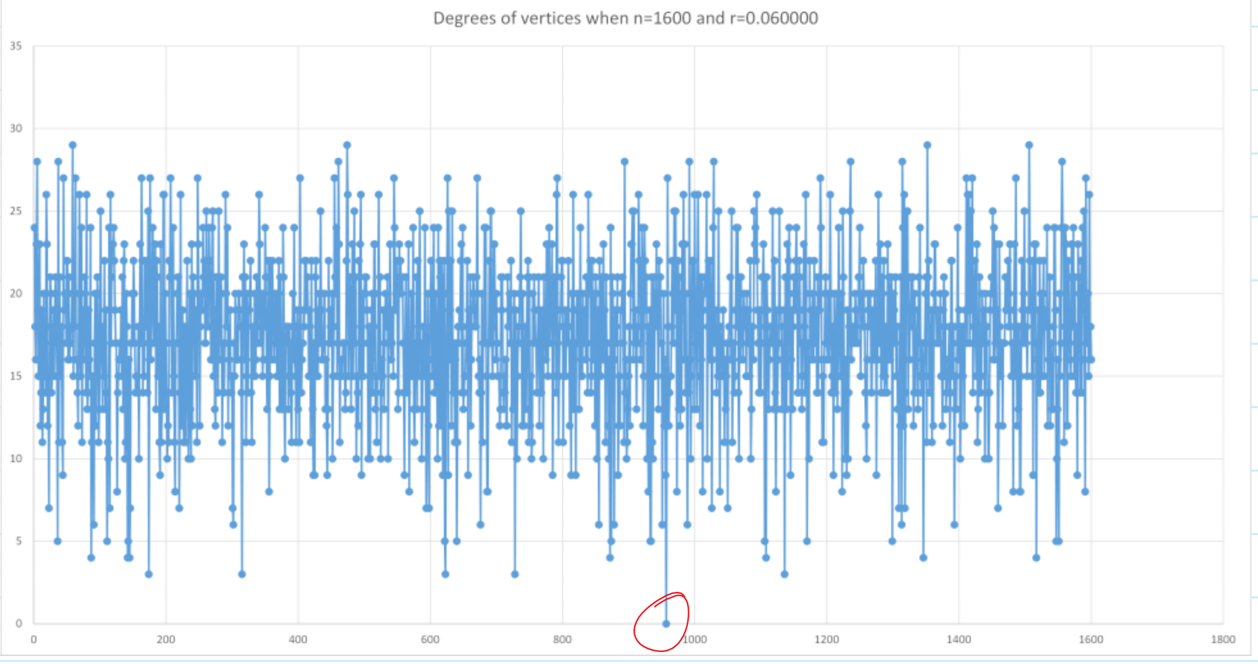


Question 6 Page 2

Question 6ii)

Below are the graphs when the smallest last order vertex is removed. Note that its degree is $\emptyset$.





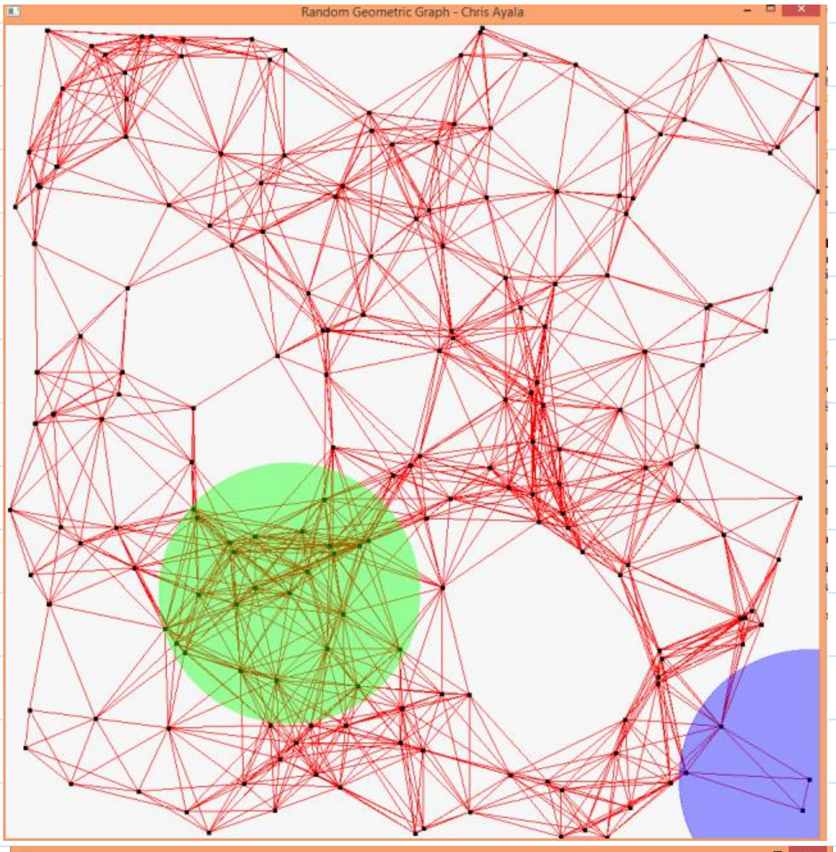
Question 6 Page 3

Question 6iii)

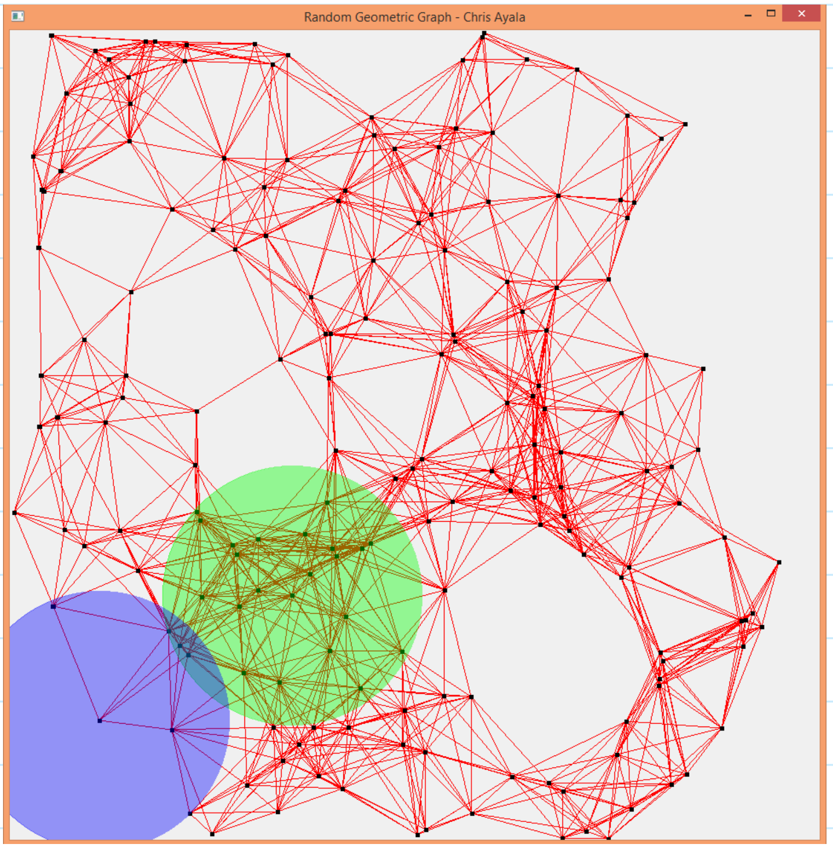
For the following, the graphs are shown in their original form, followed by -10%, -20%, -50%, -80%, and -90%

Original:

Note: In all graphs, the vertex with the highest degree is marked in green. The lowest-degree vertex is marked in blue.

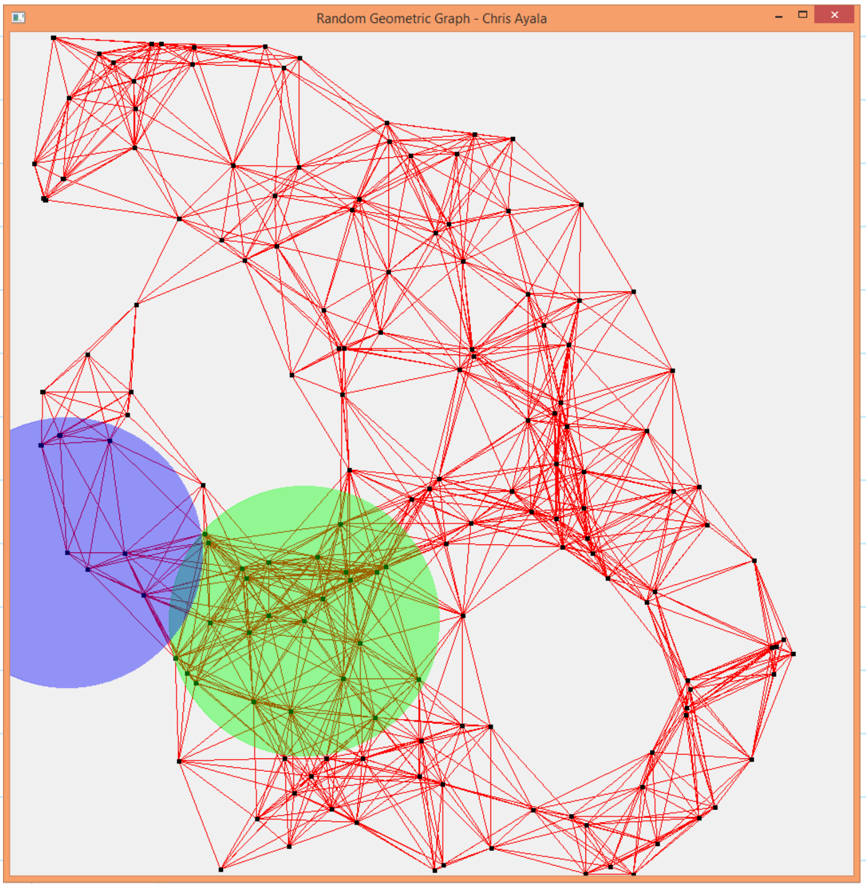


-10%

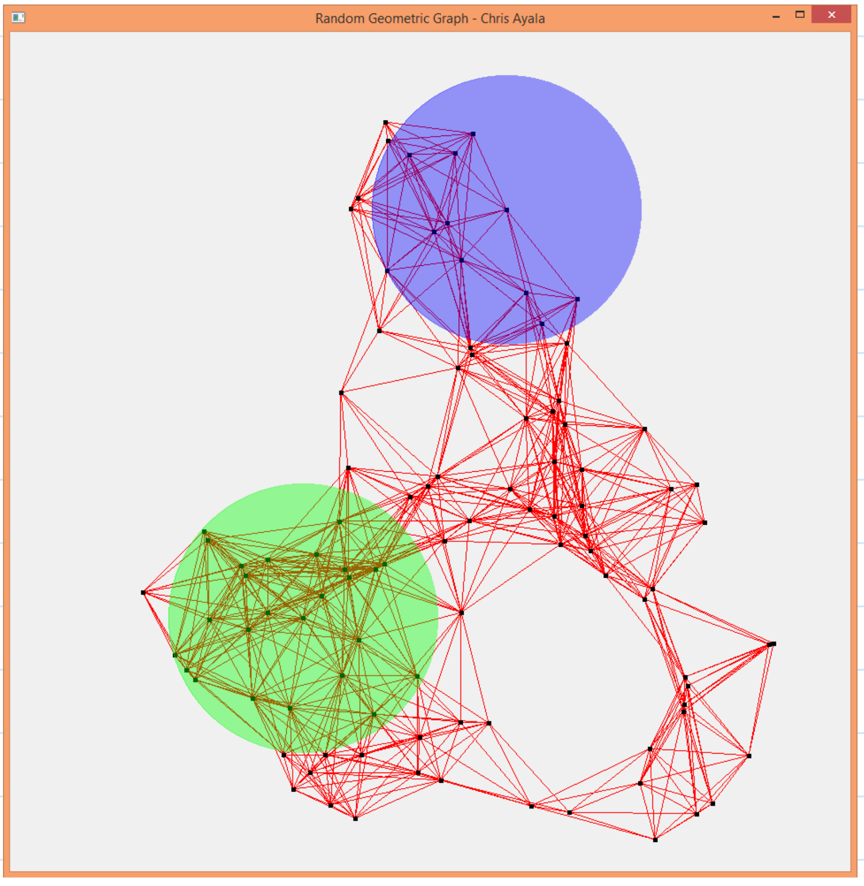


Question 6 Page 4

-20%

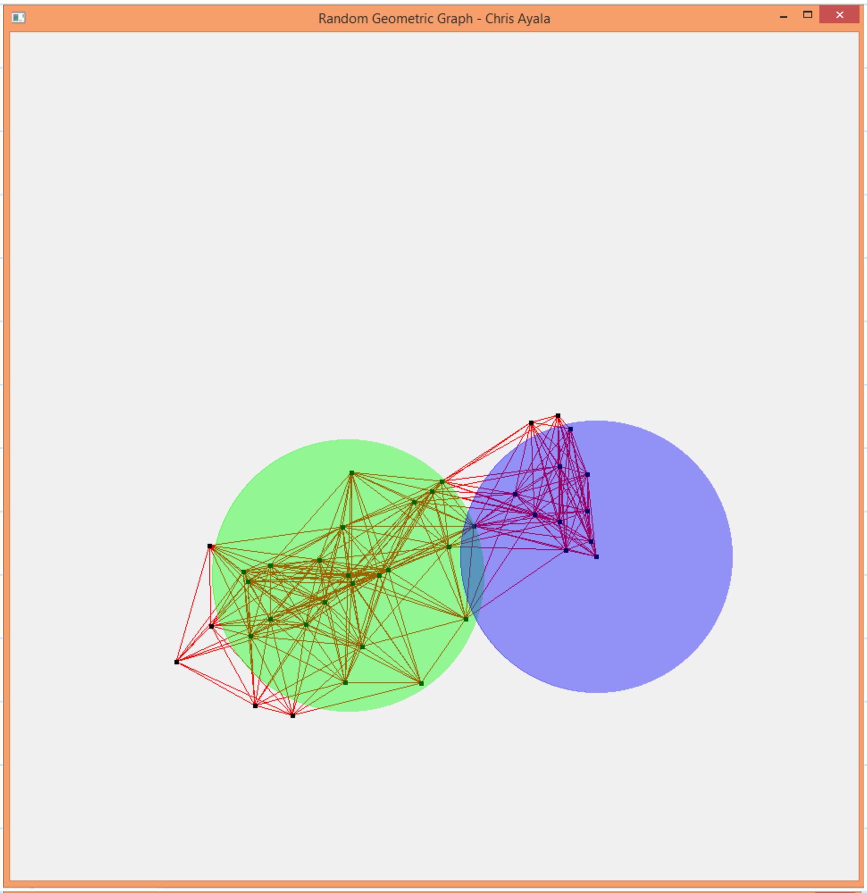


-50%

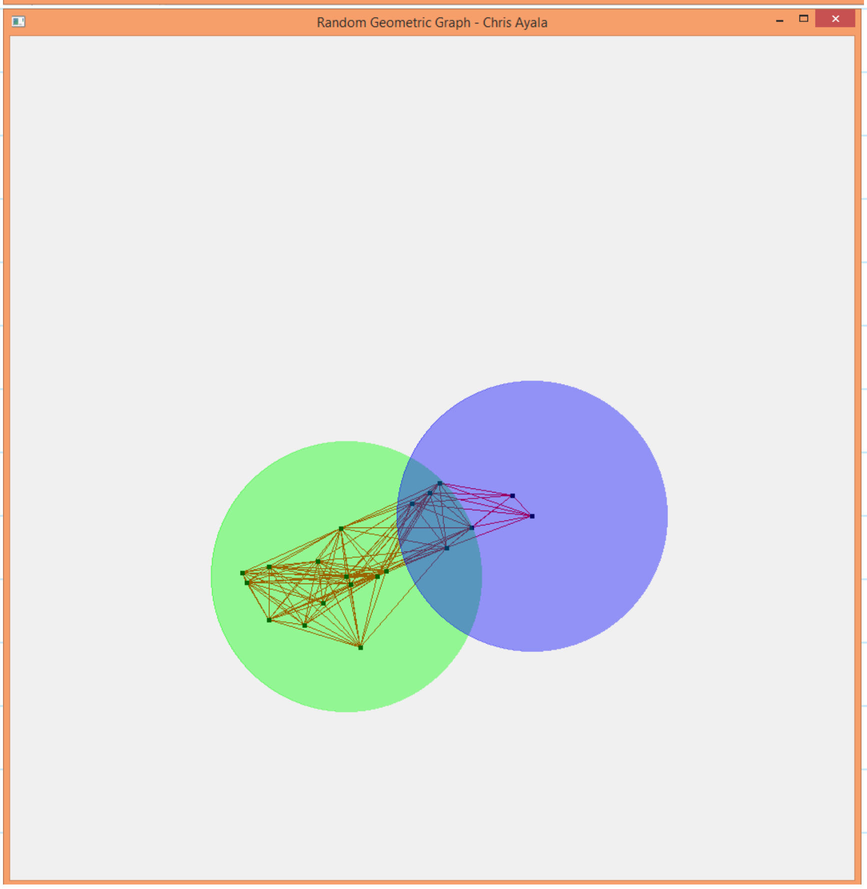


Question 6 Page 5

-80%

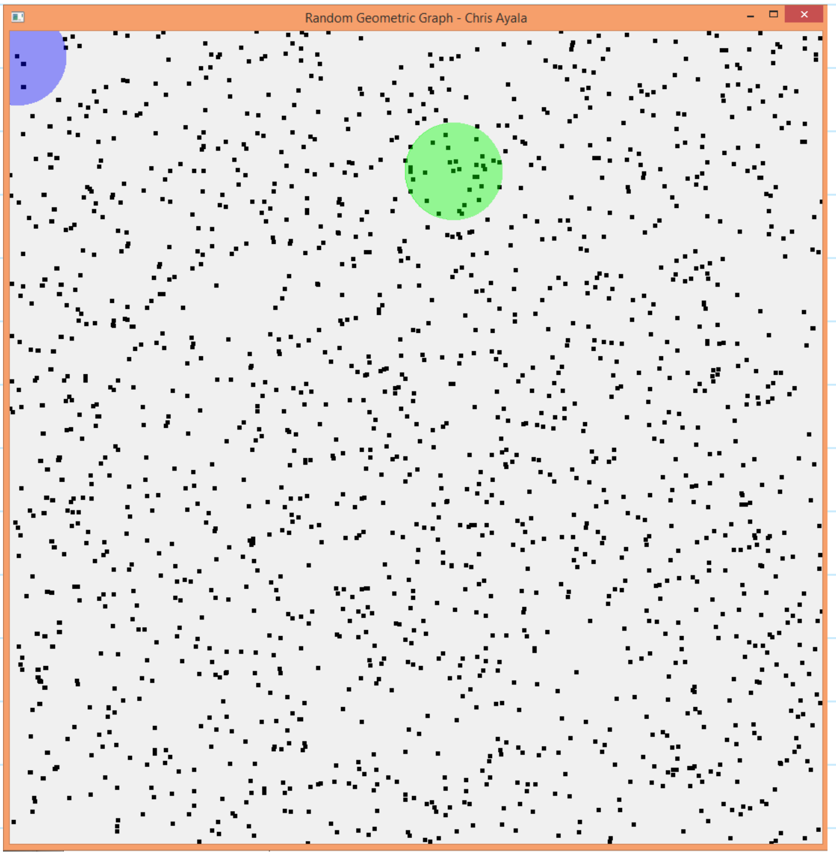


-90%

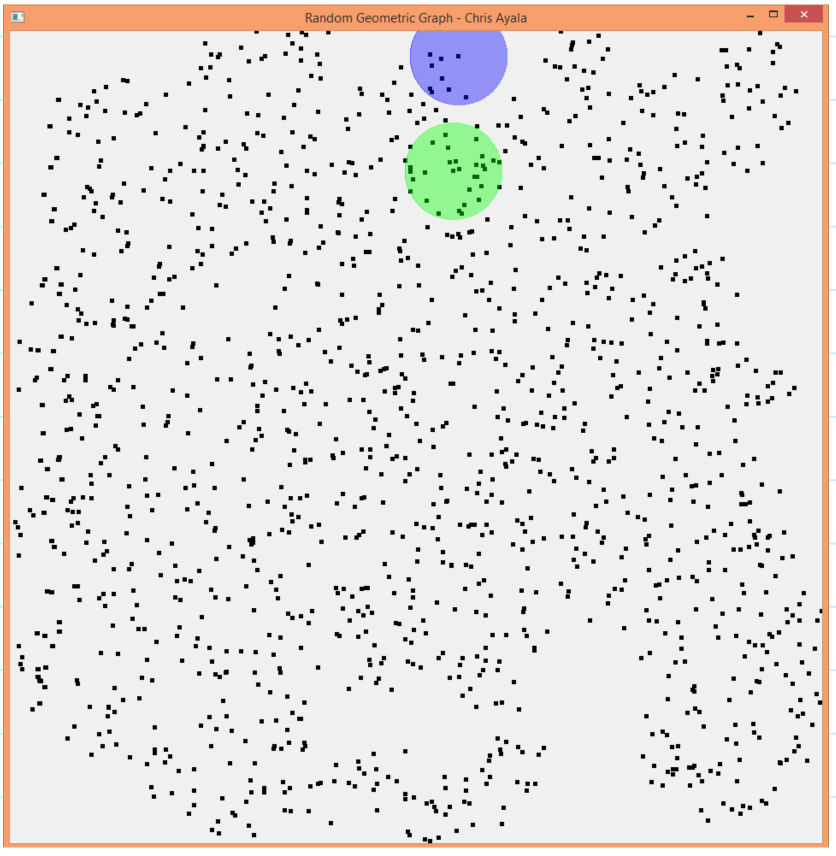


Question 6 Page 6

Original :

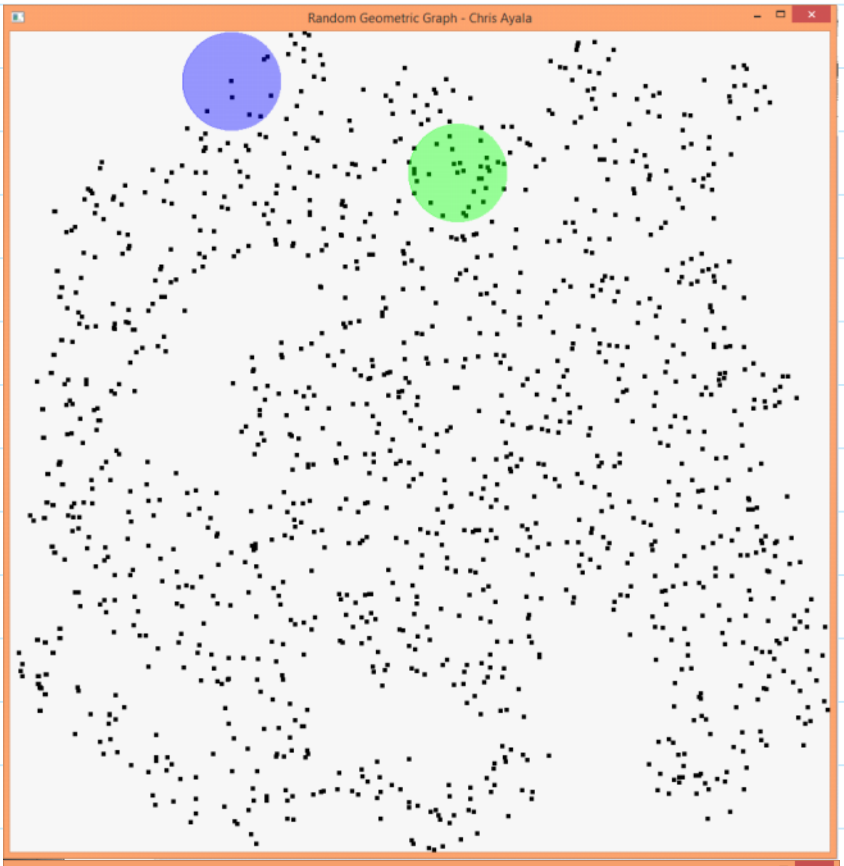


- 10%

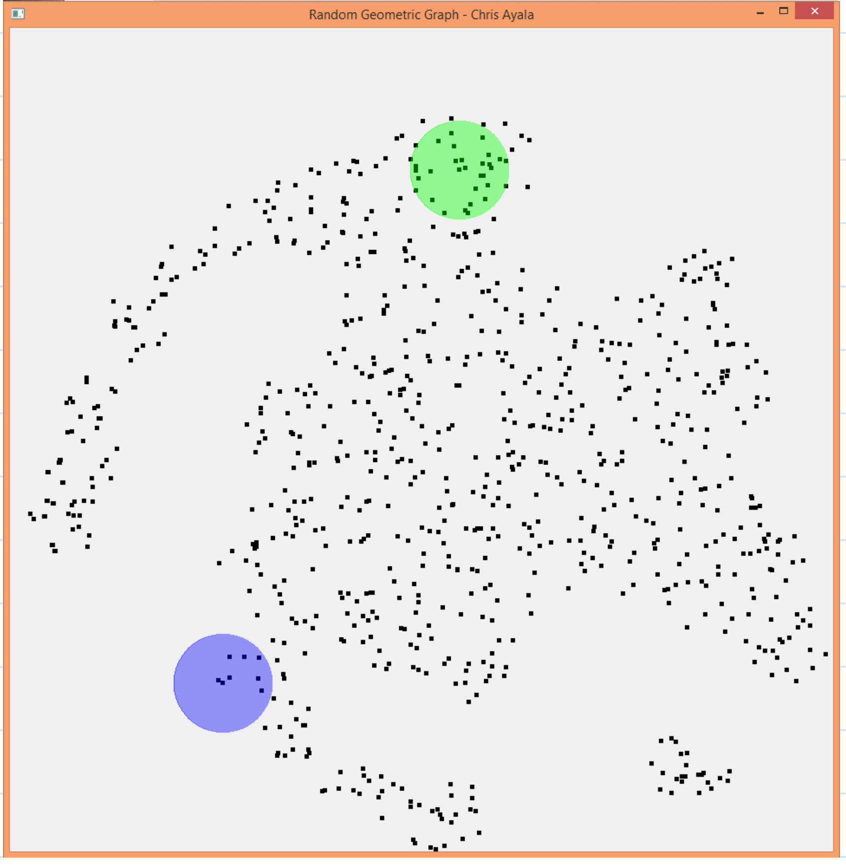


Question 6 Page 7

- 20%

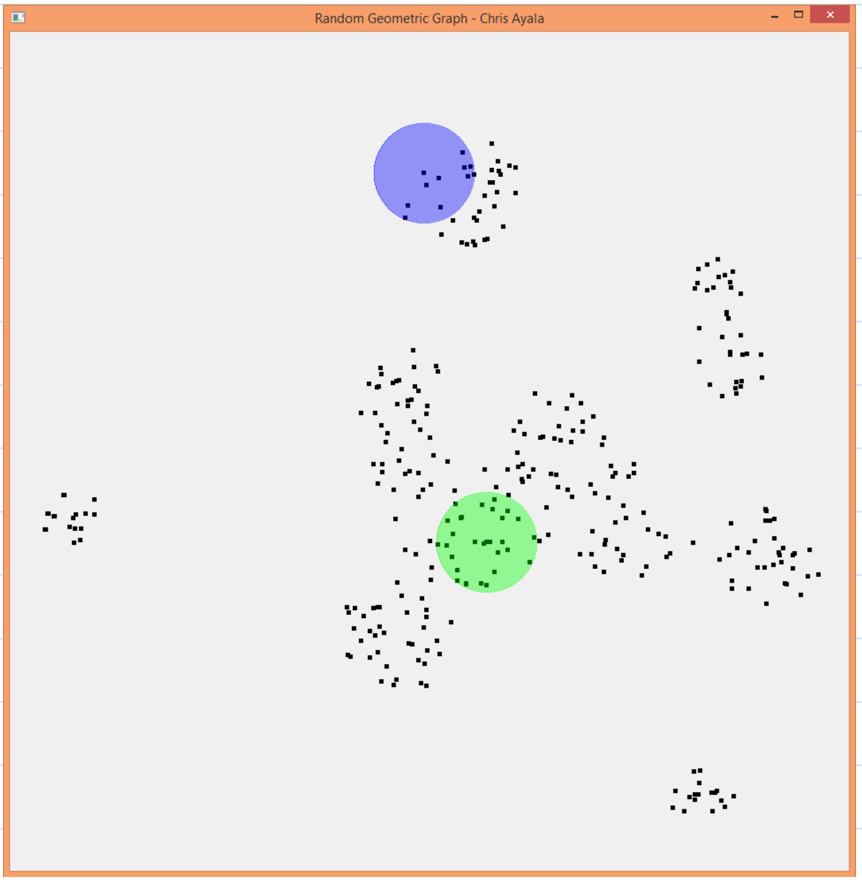


- 50%



Question 6 Page 8

-80%



-90%