CSE 7350/5350 D.W. Matula	Homework Set # 3	Due date: 03 November 2015

Total Points: 65

Greedy Paradigm
Applications

Grading Policy:
Unless otherwise stated all problems are 10 points each. Partial credit is available on all problems. Homework submitted after the due date may incur a 10 point penalty on that homework set. (Distance students should observe the same deadline.)

---

1. [FIFO Scheduling](15 points) Consider the greedy first-come first-serve coloring of interval graphs, where the n intervals all start and stop at integral times 0, 1, 2, …, t for t ≤ n. Intervals that overlap only at end points are *not* to be considered to yield edges in the interval graph. Describe an O (n) procedure for first building the *event list data structure* from the list of intervals assumed given by start and stop times. The event list is a temporally ordered sequence with each interval occurring twice. The first occurrence of each interval is determined by its start time and the second occurrence by its stop time. Then describe and analyze an implementation of the coloring procedure employing the event list data structure that operates in O(n) time and space that colors with a minimum number of colors. Provide a walk-through of your coloring algorithm for the graph with the event list **ABETAVEQMMBUVHUJTJHQ**.

Provide an algorithm determining the degrees of all vertices in O(n) time from the event list for the interval graph. Walkthrough your algorithm for the above graph (provide a drawing of the graph).

Reference: Text Problem 16.1-4, p. 422

First, we must create an event list data structure in O(n) time. To accomplish this we will traverse the list once, leading to an n traversal resulting in O(n). The traversal will basically determine wether or the letter instance is a START or END instance. The procedure for each list will be described below.

```
//original event list ordering
E = {ABETAVEQMMBUVHUJTJHQ}

//starting time ordering list
eventList = {}

//iterate over each letter in E
For each element, defined as letter, in E:
      //eventList may be defined as a hash map, achieving O(1)
      //    seek time
      if letter is not yet in eventList:
            //append the letter with a START indicator
            append {letter: letter, type: START} to eventList
      else:
            //append the letter with a END indicator
            append {letter: letter, type: END} to eventList
```

We may then take this data and transform it into an array, another O(n) procedure that looks as follows:

```
eventList = {[A, START] [B, START] [E, START] [T, START] [A, END]
      [V, START] [E, END  ] [Q, START] [M, START] [M, END]
      [B, END  ] [U, START] [V, END  ] [H, START] [U, END]
      [J, START] [T, END  ] [J, END  ] [H, END  ] [Q, END]}
```

My algorithm for coloring uses a greedy approach which chooses the soonest occurring events for optimal performance (minimum colors). Since our event list is already sorted in a ‘first in’ order then our greedy algorithm’s choice is simple. To achieve a O(n) we must optimize or removal method form the coloring we are building.

```
//starting time ordering list
eventList = {…}
//we have a generated list of unique colors in a stack
colorBuffer = []
letterColors = {}
eventBuffer = {}
//iterate over each element in the eventList
For each element, defined as event, in eventList:
      if event.type == “START”:
            append {event.letter} to eventBuffer

            //for a new letter, we must either find a recycled
            //    color, or generate a new one
            if colorBuffer.size == 0:
                  color = generateUniqueColor()
            else:
                  color = colorBuffer[0]
                  colorBuffer[0].remove()

            //we may now append the letter and color into a final
            //    dictionary data structure
            append {
                  letter: event.letter,
                  color: color}
            to letterColors


      else:
            //we may now add the color used back into the color
            //    buffer to be used once again
            colorBuffer.append(letterColors[event.letter])
            remove {event.letter} from eventBuffer
return letterColors
```

Algorithm Walkthrough:
The buffers and output array color definitions below refer to the diagrams at the end of the question

Iteration 1

To add ‘A’ to our event list we will follow to algorithm above:
We first check if A is a starting incident. Since it is, we may add A to the eventBuffer list. We must now check to colorBuffer to see if we can recycle any colors. Since we may not, we generate a new colors for A. We now append A and its defined color to our output array. The same thing happens while there are no ending incidents called.

Iteration 2

Iteration 3

Iteration 4

Iteration 5

Now we must perform iteration n = 5. This is our first ending incident, therefore we follow the else structure in our first if statement. We we now recycle our color, A’s case we recycle blue, we now remove this element, A, from the eventBuffer.

Iteration 6

Iteration 7

Iteration 8

Iteration 9

Iteration 10

Iteration 11

Iteration 13

Iteration 14

Iteration 15

Iteration 16

Iteration 17

Iteration 18

Iteration 19

Iteration 20

Buffer and Appended Outputs Table

n	buffer	output	color
1	A	A	BLUE
2	AB	B	GREEN
3	ABE	E	YELLOW
4	ABET	T	ORANGE
5	BET		
6	BETV	V	BLUE
7	BTV		
8	BTVQ	Q	YELLOW
9	BTVQ	M	PURPLE
10	BTVQ		
11	TVQ		
12	TVQU	U	PURPLE
13	TQU		
14	TQUH	H	GREEN
15	TQH		
16	TQHJ	J	BLUE
17	QHJ		
18	QH		
19	Q		

Final Color Output Table

A	BLUE
B	GREEN
E	YELLOW
T	ORANGE
V	BLUE
Q	YELLOW
M	PURPLE
U	PURPLE
H	GREEN
J	BLUE

2. [Graph Degree Structures] Describe data and record structures for vertex ordering and vertex or edge coloring (or labeling) and a suitably greedy graph search algorithm to solve each of the following problems in the time bound indicated. Illustrate each algorithm with vertex or edge coloring (or labeling) on a graph or tree designed to teach your algorithm. The graph (or tree) should have at least 20 vertices and a maximum degree of at least 5. The graph should be connected with a minimum degree 3.

i) Find a maximum independent set in a tree in time O (|V|).

i. A maximal independent set is simply defined as a set of element for which each element is not a sibling of any other element. To determine a maximum independent in O(|V|) it is obvious that we must use an algorithm which traverses each vertices once. The following pseudo code explains the algorithm:

```
Vertices = |V|
independent_set = {}
while Vertices is not {}:
      v = Vertices(randomElement)
      indepedent_set.append(v)
      for each s in v.siblings
            V.remove(s)
return indepedent_set
```

The algorithm begins determining maximal sets at the first random element is finds, therefore this maximal set may not be maximal set for the random element chosen, while it will not find the MIS with the greatest cardinality. To find the MIS with the greatest cardinality we must take a greedy approach to choosing our initial vertices to add to our independent set.

1. Perform bucket sort with buckets from the Minimal Vertices Degree to the Maximum Vertices Degree (defined as 1 to 5 in the above question).

1. We first make the greedy decision my gathering all the leaves (or lowest vertices count) from the graph. This inherently forms an independent.

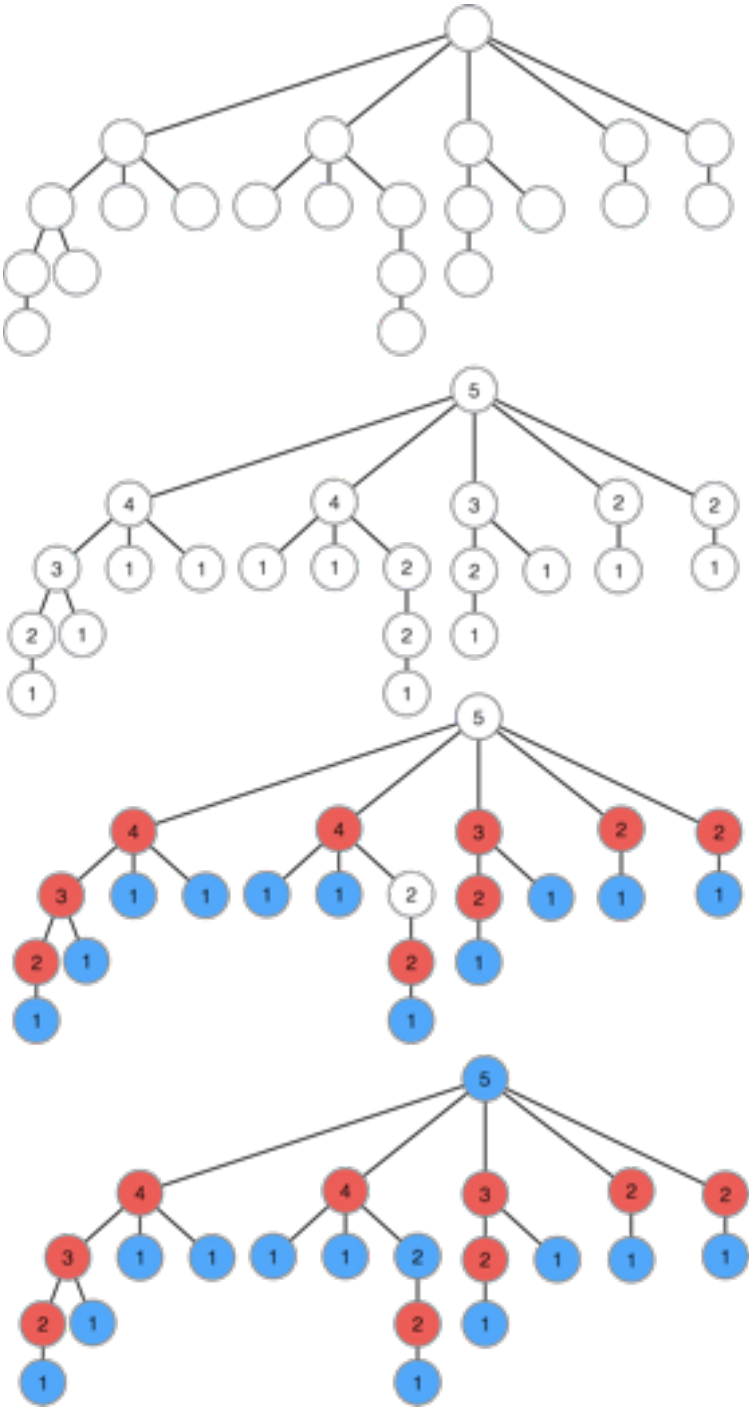
2. Now we may remove all of the parents, since these are obviously not part of the independent set.

3. Next we will take the remaining vertices and repeat step one, iteratively, until we have no more vertices left.

This is a good candidate for the MIS, though it this the subset with the highest cardinality? Although this is our local maximum for the sets we begins with, is it, globally, the MIS? If you deconstruct the problem to a parent and child relationship, we can understand that the if we add the parent to the independent

set, we eliminate two possibilities form the independent set. Therefore, if we are attempted to reach the MIS with the highest cardinality, it would be in our interest to add both children to the independent set. Let's do an example for further proof.

Graph Example:



In step 1, we begin with a blank graph. To get to step 2, we must use bucket sort on each vertices degree. The smallest vertices degree are considered the elements which should be added to the graph first (typically the leaves). In step 3 we color each minimal degree with blue (indicating we have added the vertices to our independent set), and then each of the parents with red (indicated we have removed them from the next step in processing and they they may NOT be part of our independent set). In step 4 we repeat this process until we have no more vertices in our buffer (meaning each vertices has either been eliminated because it is parent or added the independent set because it a of the lowest degree of that round).

ii) Find some “k-connected” pair of vertices u, v in the graph having k = min {degree (u), degree (v)} edge disjoint distinctly colored (labeled) paths between u and v in time O (|V|+ |E|) using maximum adjacency search. Identify all the sets of pairwise k-edge-connected vertices using the "k-1 cycle" observation discussed in class.

Reference: For (ii) see web page Project Description for Maximum Adjacency Search and the example walkthrough.

3.[Matching Substrings] A matching substring pair of length k in a binary bit string $b_1, b_2...b_n$ is a pair $b_ib_{i+1}... b_{i+k-1} = b_jb_{j+1}...b_{j+k-1}$, with i not equal to j, of identical k-bit substrings. Determine a maximum length binary string with no matching substrings of length 4.

Reference: See notes on web page.

First, we begin by determining the total number of permutations a substring may take on. Since the substring is binary, we can identify the two binary indicators as a set of {A, B}. Since we are checking for substring of length 4, we may now create a set of all possible substring from these binary indicator set as defined in the table below.

Permutations Table

AAAA	AAAB	AABA	AABB
ABAA	ABAB	ABBA	ABBB
BAAA	BAAB	BABA	BABB
BBAA	BBAB	BBBA	BBBB

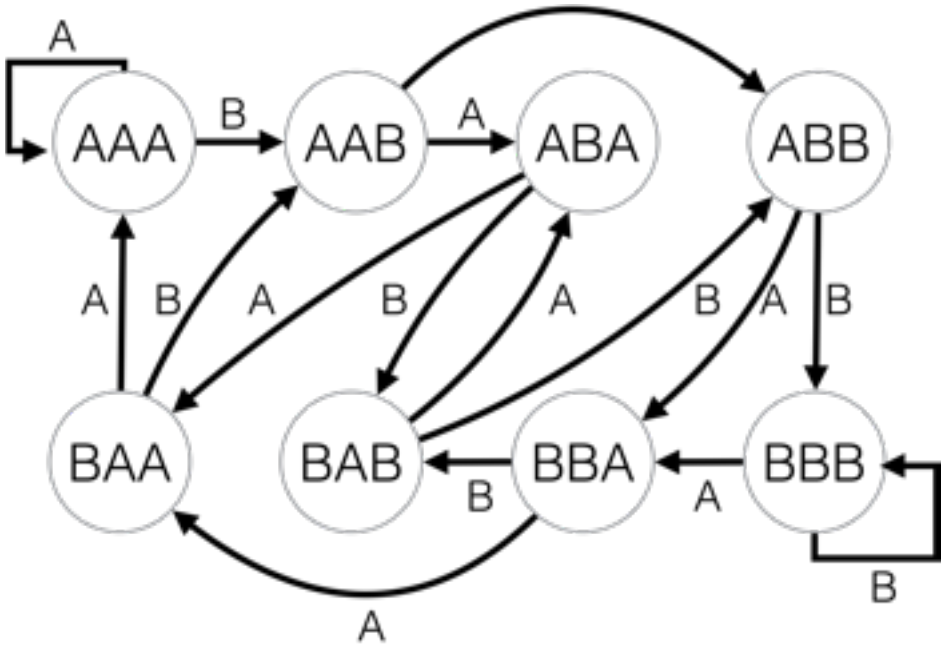
We may now verify our permutation count by theoretically calculating it as 2, since we have a binary set of possible values for each digits, to the power of 4, since we have 4 digits total ($2^4 = 16$). We have a total of 16 possible combinations.

We may now determine the maximum string by combining all of these elements. No matter where we start, we will want to go through all of these element to get through the maximum amount of characters, plus 3 for the last three characters. This gives us 19 as K.

We may now reduce our sequence defined combinatorial problem by generate a state matrix to define our unique transitions between each of our combination defined above.

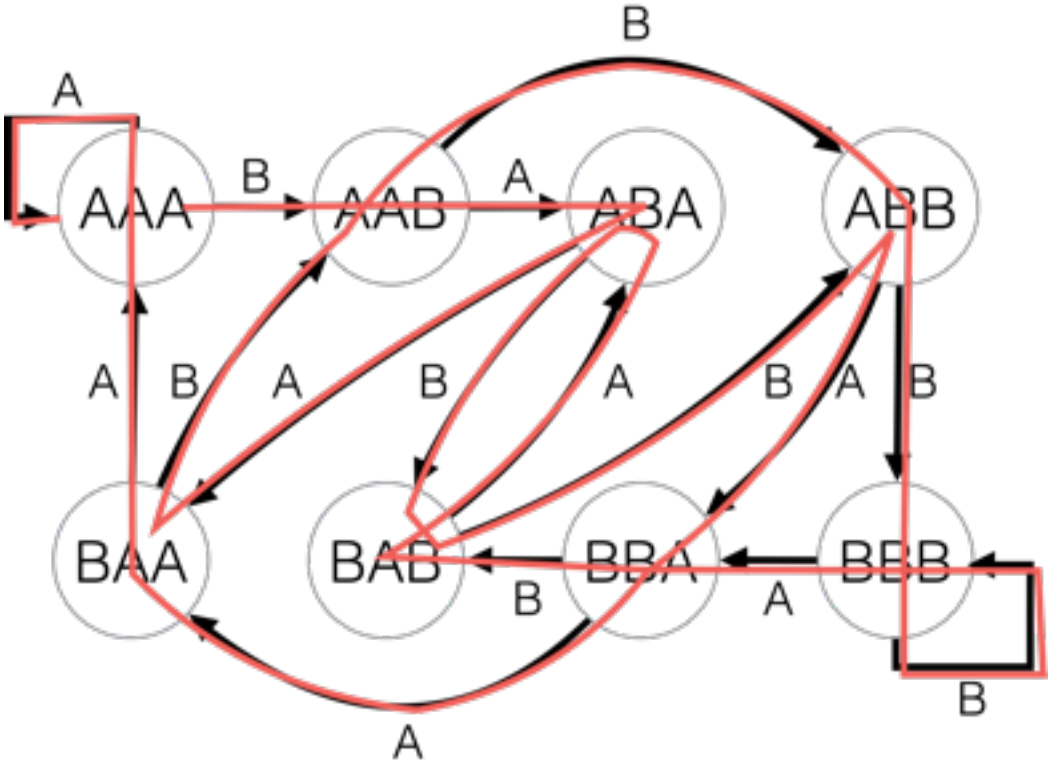
The way this string is formed are by determining which substring of length 4 comes next. We may now determine each state for when we will choose each of the combinations determined above in a State Transition Table.

State Transition Table



We may now use the State Transition table to generate the maximum Tour (the key difference between a maximum adjacency tour and a maximum adjacency path maximum adjacency tour may repeat vertices while a maximum adjacency path may not). We always want to traverse to a node with as many out nodes still available to avoid getting stuck in a node and not completing a full tour of the state transition table. We may perform Euler's Tour. This is where our greedy algorithm is applied. It allows for each node jump to be chosen simple base on outward pointing arrows from a single node. We want to jump to the one s with the most to ensure that we do not get stuck until we reach the starting node once again.

Complete Tour



The Tour defined above by our greedy algorithm generate the string: {AAABAABBBABABBAAAA} which is exactly 19 characters as predicted above.

A Maximum Length Binary String

{AAABAABBBABABBAAAA} OR

{1110110001010011111} IN BINARY

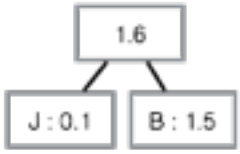
Max Length = 19 Bits

4. [Data Compression] Draw a Huffman coding tree for the following letter frequencies for the first ten letters of the English alphabet:

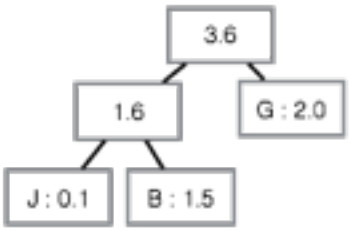
E 12.0; A 8.1; I 7.3; H 5.9; D 4.3; C 2.7; F 2.3; G 2.0; B 1.5; J 0.1

Note that the letters are arranged in decreasing frequency. What is the time complexity of building such a Huffman coding tree for an *n* letter alphabet arranged in decreasing frequency order?

letter	frequency
E	12.0
A	8.1
I	7.3
H	5.9
D	4.3
C	2.7
F	2.3
G	2.0
B	1.5
J	0.1

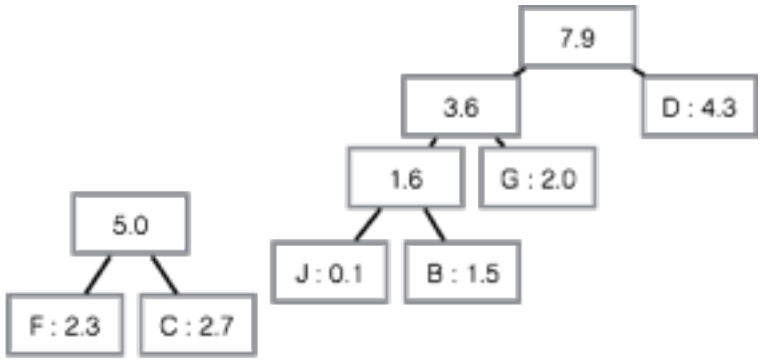


letter	frequency
E	12.0
A	8.1
I	7.3
H	5.9
D	4.3
C	2.7
F	2.3
G	2.0
*	1.6

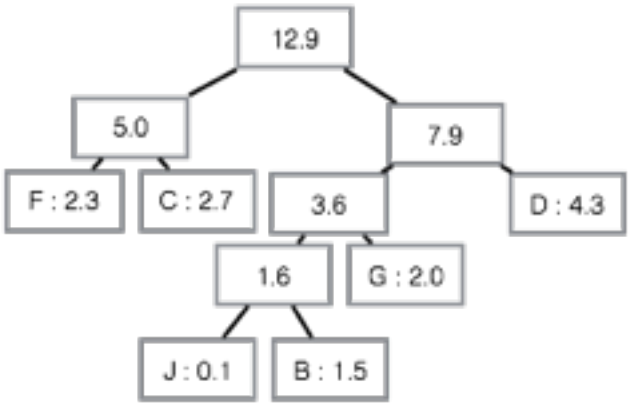


letter	frequency
E	12.0
A	8.1
I	7.3
H	5.9
D	4.3
*	3.6
C	2.7
F	2.3

letter	frequency
E	12.0
A	8.1
I	7.3
H	5.9
*	5.0
D	4.3
*	3.6

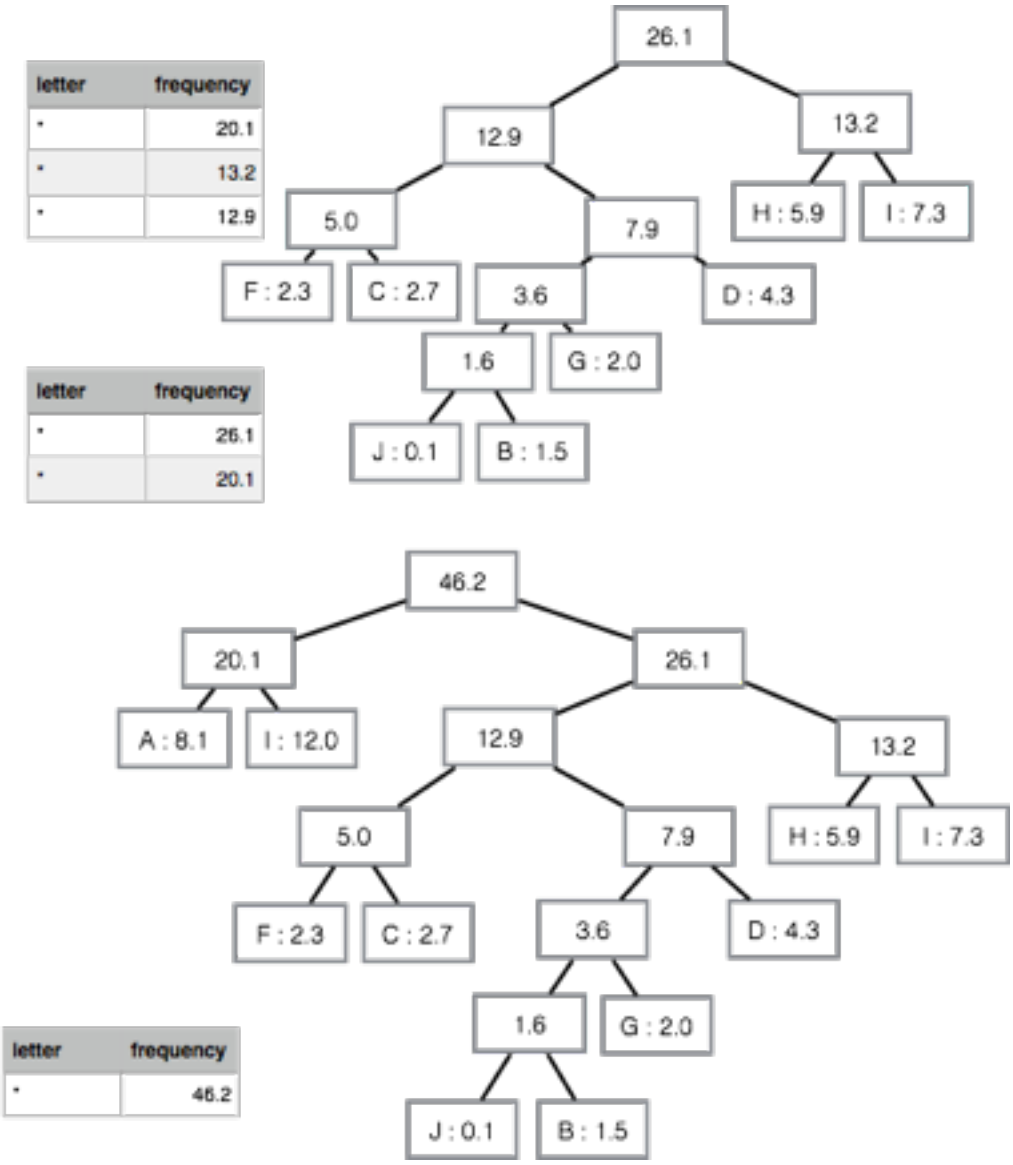


letter	frequency
E	12.0
A	8.1
I	7.3
H	5.9
*	5.0
*	7.9



letter	frequency
*	12.9
E	12.0
A	8.1
I	7.3
H	5.9

letter	frequency
*	13.2
*	12.9
E	12.0
A	8.1



For an n letter alphabet there will be a maximum of n elements in the frequency array. We must iterate a total of O(n) time to add each element to a Huffman tree. Furthermore, since we may use a greedy algorithm, which takes the two least most elements, and inserts them into a tree with out any searching required. This can be achieved because our frequency list si already sorted. Therefore, construction may be done in O(n) time complexity.

5. [Squaring Arrays] The following parts (a) and (b) refer to problems for section 4.7 on page 269 of **Finite Precision Number Systems and Arithmetic.**

(a) Do problem 4.7.1. Include a brief explanation of your product array formation. (Illustrate the consolidated partial product array employing radix-2 squaring for the 16-bit square of an eight-bit operand.)

(b) Do problem 4.7.2. You may delete low order insignificant zeros in the argument and consolidate the rows. Show the array before and after consolidation. (For the 16-bit normalization binary operand x = 1.1010 1110 0101 100, illustrate the left-to-right radix-4 recoded partial square array as in Example 4.7.1, extending the array formation to include the needed 2's complement bits.)

6. [Maximal Independent Set in an RGG] A maximal independent set in a graph is a subset of vertices that are pairwise non adjacent and where no other vertex can be added preserving this property.

(a) Implement a greedy maximal independent set selection algorithm for a random geometric graph (RGG) where you recursively select a vertex of minimum degree for the set and delete it and its adjacent vertices, and then repeat on the residual graph.

(b) Determine an approximate upper bound on the maximal independent set size by assuming the largest such set forms a triangular lattice. Run your program 10 times and find the average set size you obtain for RGG's on 2,000 vertices with average degree about 40, and also for 8,000 vertices with average degree about 100. Compare your obtained set size with the approximate bound.

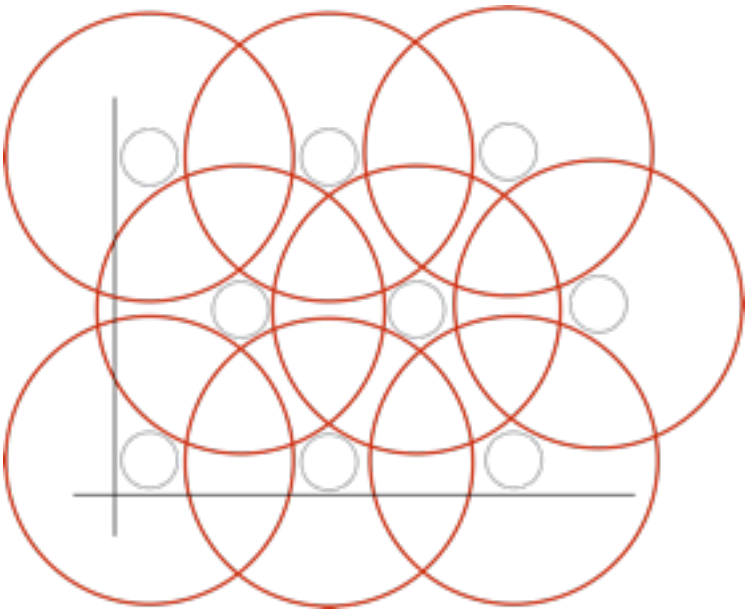
a. The RGG greedy algorithm was implement in Python using the NetworkX library for visualization. I implemented both a recursive and iterative approach as defined below in the code. The output for the following code slice is at the end of this problem. Output we split into a full RGG with white dots indicating the independent set, and an RGG with just the independent set displayed.

```
import networkx as nx
import matplotlib.pyplot as plt

nodes = 8000
G=nx.random_geometric_graph(nodes,0.0634)

# get node degrees dictionary
node_degrees = {}
for n in nx.nodes(G):
      node_degrees[n] = nx.degree(G,n)
#sort our degrees list in an ascending order
degree_sort = sorted(node_degrees.items(), key=lambda x:x[1])

#calculating the average degree for verification
avg = 0
for n in degree_sort:
      avg += n[1]
print("Average Degree: " + str(avg/nodes))

#start an independent set indicator list
indepedent_set = {}
for key in node_degrees:
      indepedent_set[key] = 1

#iterating over each node
for node in degree_sort:
      if indepedent_set[node[0]] == 1:
            #removing all neighbors which connect to the node in the ind_set
```

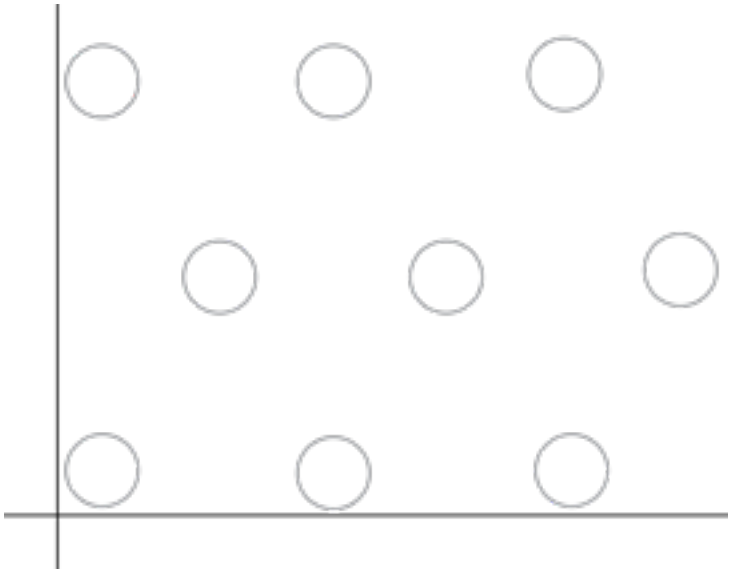
```
            neighbors = nx.all_neighbors(G,node[0])
            for child in neighbors:
                  indepedent_set[child] = 0
#recursive removal based on the lowest element as our greedy choice
def recursiveRemoval(ind):
      if len(nx.nodes(G)) == 0: return
      min = (nx.nodes(G)[0], nx.degree(G,0))
      for n in nx.nodes(G):
            n_deg = nx.degree(G,n)
            if n_deg < min[1]:
                  min = (n, n_deg)
      ind.append(min[0])
      neighbors = nx.all_neighbors(G,min[0])
      for n in neighbors:
            G.remove_node(n)
      G.remove_node(min[0])



#create network visualization
plt.figure(figsize=(12,12))
pos=nx.get_node_attributes(G,'pos')
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_nodes(G, pos, node_color='r', alpha=0.8)

#removing all nodes that are not in the Maximal Independent set
set_size = 0
for node in indepedent_set:
      if indepedent_set[node] != 1:
            G.remove_node(node)
      else:
            set_size += 1
print("SIZE: " + str(set_size))

#graphing the independent set on top of the network graph
pos=nx.get_node_attributes(G,'pos')
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_nodes(G, pos, node_color='w')
#making things look pretty
plt.xlim(-0.05,1.05)
plt.ylim(-0.05,1.05)
#plt.axis('off')
#plt.savefig('random_geometric_graph.png')
plt.show()
```

b. In order to determine an approximate upper bound in the maximal independent set size we must first visualize the problem. The following graphs describe my understand of forming tightly packed (maximal independent set) from a node set by hand.

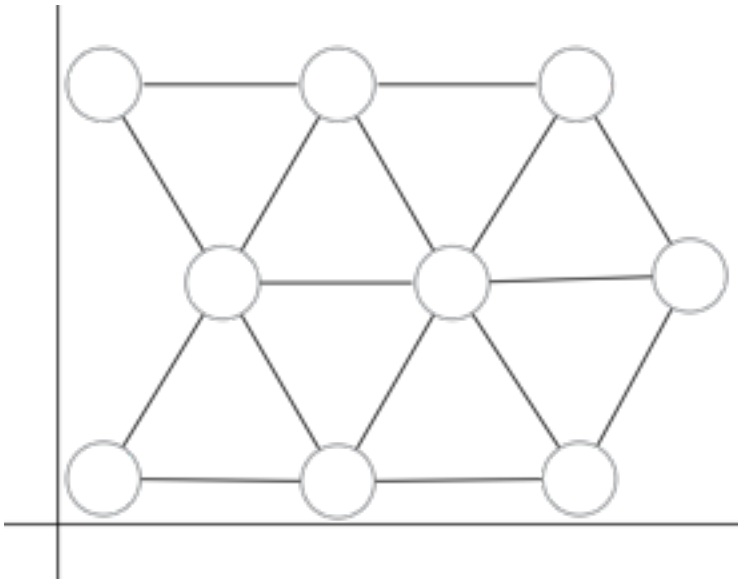
Tightly Packed Non-Connected Node Structure



Node Topology (No Boundaries)



Triangular Lattice Drawn On Top Of Node Topology



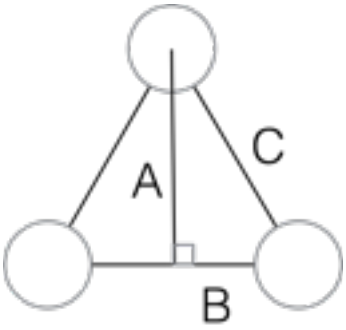
After visualizing the tightly packed triangular lattice, our maximal independent set for any graph will be based on the distance between linked nodes in a random geometric graph. This distance is often refer to as the radius. Thus, our equation for the upper bound on maximal independent sets will have R as a term, representing radius. We will use the property of an equal lateral triangle to determine our upper bound. If we start from the bottom right of the graph, we may expand our first layer of node horizontally right, incrementing by R, each time. Therefore the first each layer may have a maximum of

Odd Layers = FLOOR(1/R) vertices.

This is considering that each odd layer it the same (which the bottom left vertices is part of). Next we must calculate how many vertices are in the even layers. To perform this task we must first figure out the offset from left, that each of these layer will have. Since we are dealing with equal lateral triangles, this offset with be R/2. Therefore the equation for even layers is

Even Layers = FLOOR((1/R) - (R/2)) vertices

Next we must calculate how many horizontal layers the graph will include by recalling basic geometry. We may split an equal lateral triangle in two, making a 90 degree triangle. We way no perform Pythagorean theorem on this triangle to calculate the height.



$$Recall\ that\ A^2 + B^2 = C^2$$

$$A^2 = C^2 - B^2$$

$$A = sqrt(C^2 - B^2)\ where\ B = R/2\ and\ C = R$$

$$Height = sqrt(R^2 - (R/2)^2)$$

Therefore the number of rows we may have is equal to

$$Rows = FLOOR(1/sqrt(R^2 - (R/2)^2))$$

The final equation comes out to

$$UPPER\ BOUND = \{[CEILING(FLOOR(1/sqrt(R^2 - (R/2)^2))/2) * FLOOR(1/R)] + [FLOOR(FLOOR(1/sqrt(R^2 - (R/2)^2))/2) * FLOOR((1/R) - (R/2))]\}$$

RGG Testing

In order to get an average vertices degree of 40 and 100, we must reverse engineer a question from homework 1. This question had us estimate the degree of a vertices in an RGG with an equation. That equation was defined as so:

$$Average\ Degree = [(pi*r^2*n)/A]-1$$

Where R is the radius of connected vertices, n is the numbers of node in the RGG, A is the size of RGG (always 1). RGGs are generating using a distance value, typically defined as R, to determine connected edges. Therefore we will solve the equation above or R, instead of Average Degree.

$$R = sqrt((Average\ Degree + 1) * A/n*pi)$$

We may now apply this equation to both our sets. Sets = [(2000 nodes, Average degree of 40), (8000 nodes, Average Degree of 100)].

$$R = 0.0808\ for\ Average\ Degree = 40,\ Area = 1,\ Number\ of\ Nodes = 2{,}000$$

$$R = 0.0634\ for\ Average\ Degree = 80,\ Area = 1,\ Number\ of\ Nodes = 8{,}000$$

Now that this has been calculated, we may generate our RGG's.

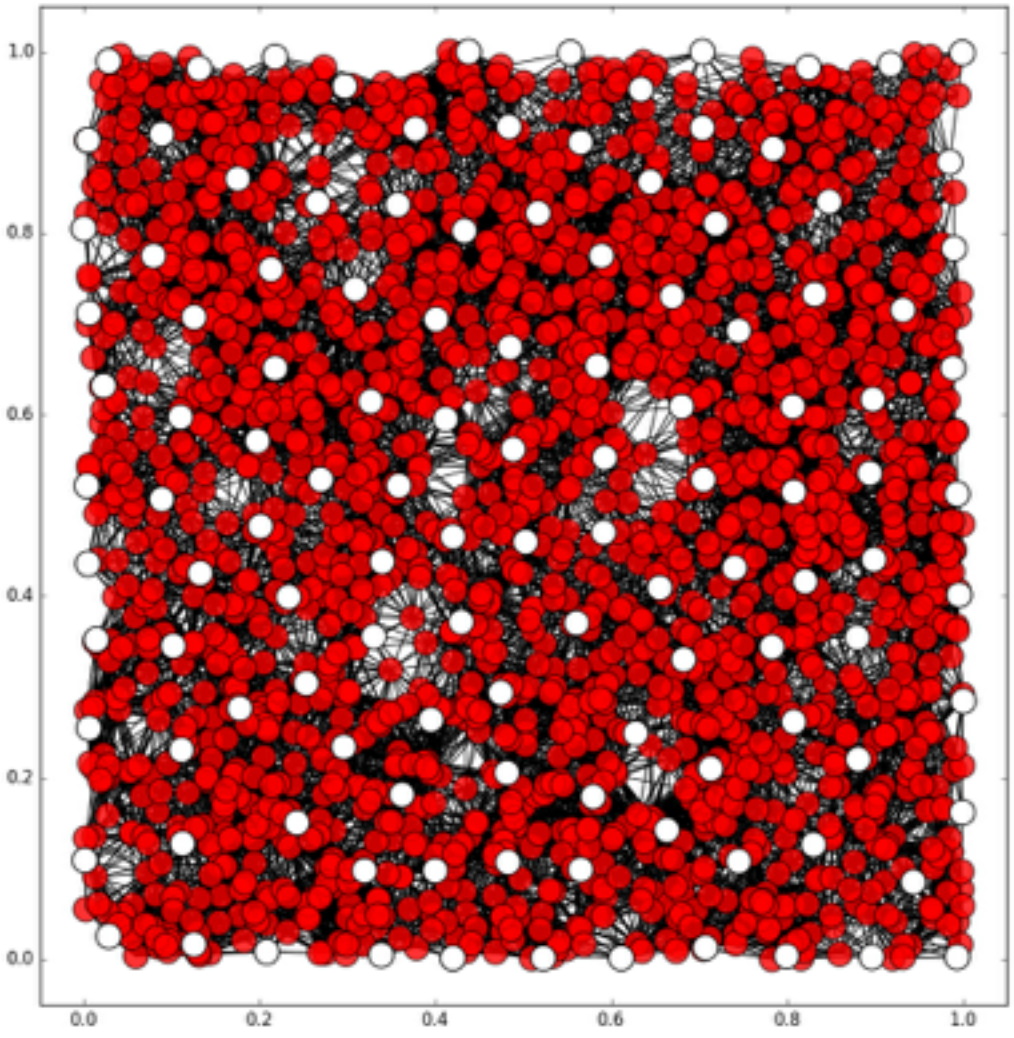
Runs	MIS Size N = 2,000	MIS Size N = 8,000
1	119	200
2	117	205
3	120	202
4	114	195
5	117	197
6	118	203
7	119	205
8	120	200
9	119	196
10	121	198
Averages	118.4	200.1
Derived Upper Bound	**168**	**270**
Deviation %	**29.5238095238095**	**25.8888888888889**

The derived upper bound from above we can see that the upper bound did hold. Looking at the actually graph below, it is evident that the triangle lattice was not tightly packed, as an ideal lattice would be. Therefore the deviation of 25-30% is expected. The deviation also decreased as the number of nodes increased, meaning that increasing the number of nodes leader to a more tightly packed triangular lattice. Over all, I would say that this upper bound is accurate.

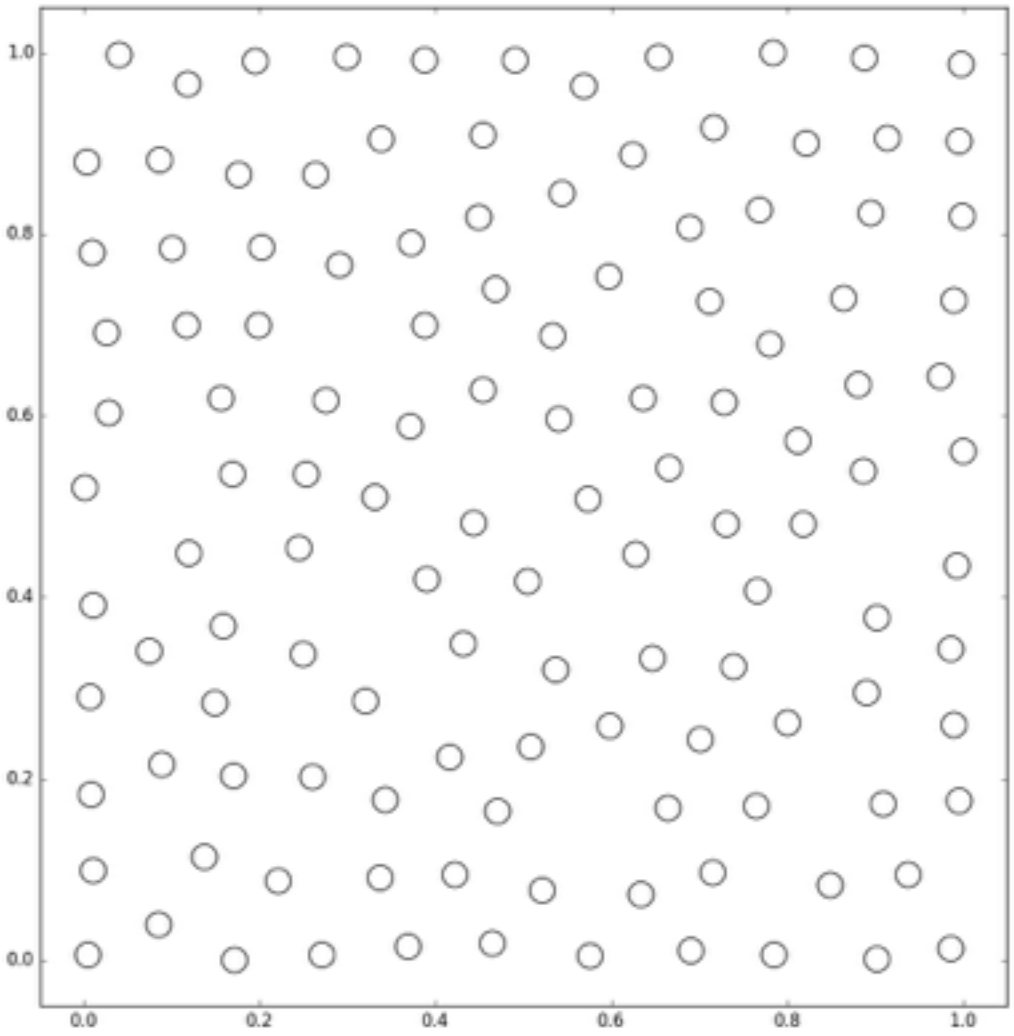
Examples of the RGG's generated

White Dot = Part of Independent Set

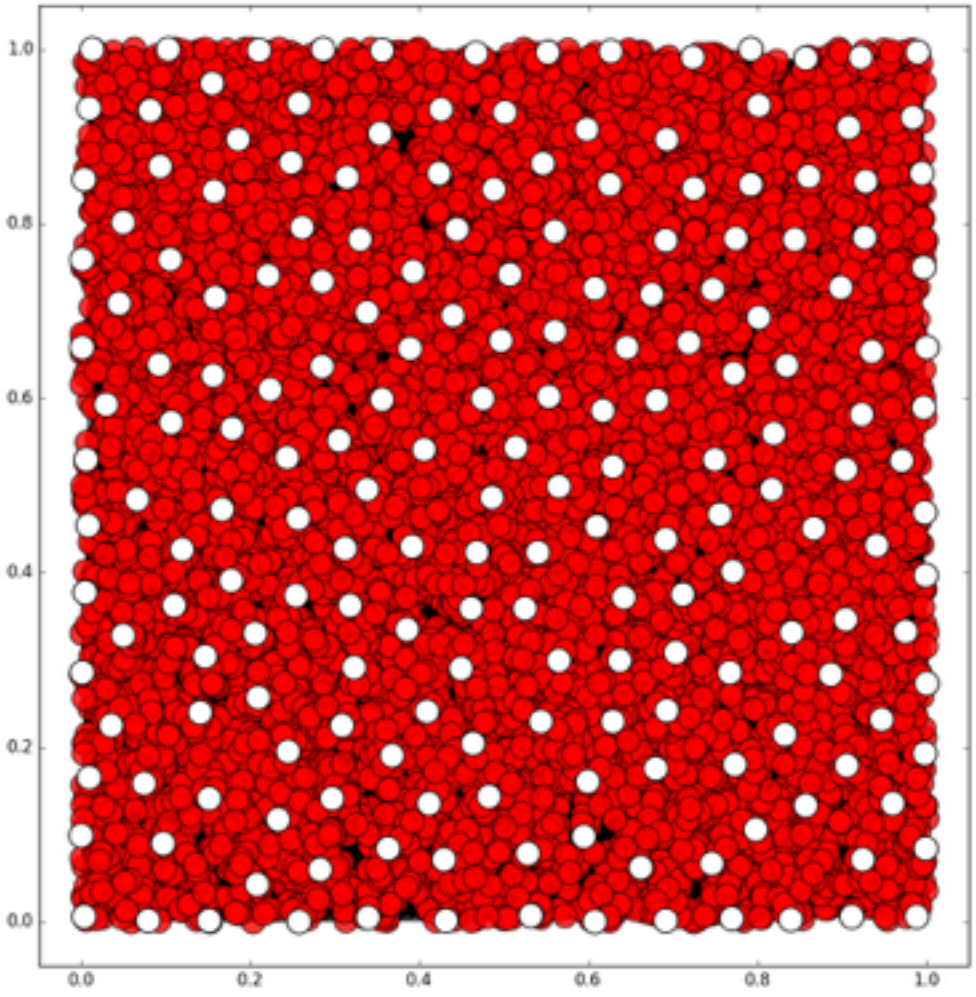
2,000 Vertices RGG with an average degree of 40 (Entire Graph)



2,000 Vertices RGG with an average degree of 40 (Independent Set Only)



8,000 Vertices RGG with an average degree of 100 (Entire Graph)



8,000 Vertices RGG with an average degree of 100 (Independent Set Only)