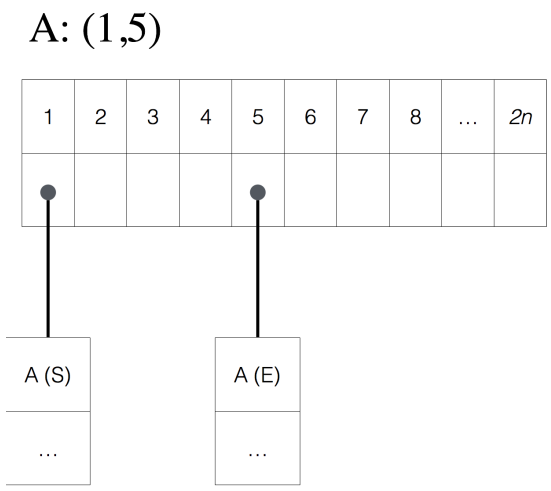
Greedy Paradigm Applications

1. [***FIFO Scheduling***]
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

Creating an Event List from an Interval List
In order to build the event list in O(n) time, a hash table of size 2*n* can be used where each index in the table points to an array of structs. A single pass through the sorted interval list will be made where two structs will be created for each event. Each struct will contain the letter value and either the start time or the end time of the event. These structs will then be pushed into the arrays located at the indices of the start and end time respectively. Finally, because each index of the hash table points to an array of structs, if multiple events occur concurrently, they will be added to the same array within the hash table. An example using the event list representation of *A* is shown in the diagram to the right.



At this point, the new data structure contains event list information in an ordered format. Finally, a pass through the 2*n* length array containing arrays of structs will be made in order to store all events in a single array of size 2*n*. Using the example data from above, this algorithm will generate the following event list, where *S* and *E* represent the event's start time and end time respectively and the numerical value represents the time of the event:

A (S1), B (S2), E (S3), T (S4), A (E5), V (S6), E (E7), Q (S8), M (S9), M (E10), B (E11), U (S12), V (E13), H (S14), U (E15), J (S16), T (E17), J (E18), H (E19), Q (E20)

Overall, because this algorithm only required a single pass through the original interval list of size *n* and a subsequent pass through the data structure of length 2*n*, the algorithm results in a time complexity of O(*n*). Furthermore, because two additional data structures of size 2*n* were required, the algorithm operates with a space complexity of O(*n*).

Coloring Events with a Minimal Number of Colors

In order to apply a coloring procedure to the event list in linear time and space, a stack of *n* colors must first be created. It is very likely that only a small portion of the *n* colors will be used, however, if all events occur concurrently, *n* colors will be needed. Next, a hash table data structure must be created to contain the list of events and their associated colors. Each index of the hash table will point to an array of structs where each struct contains event information as well as the color assigned to it.

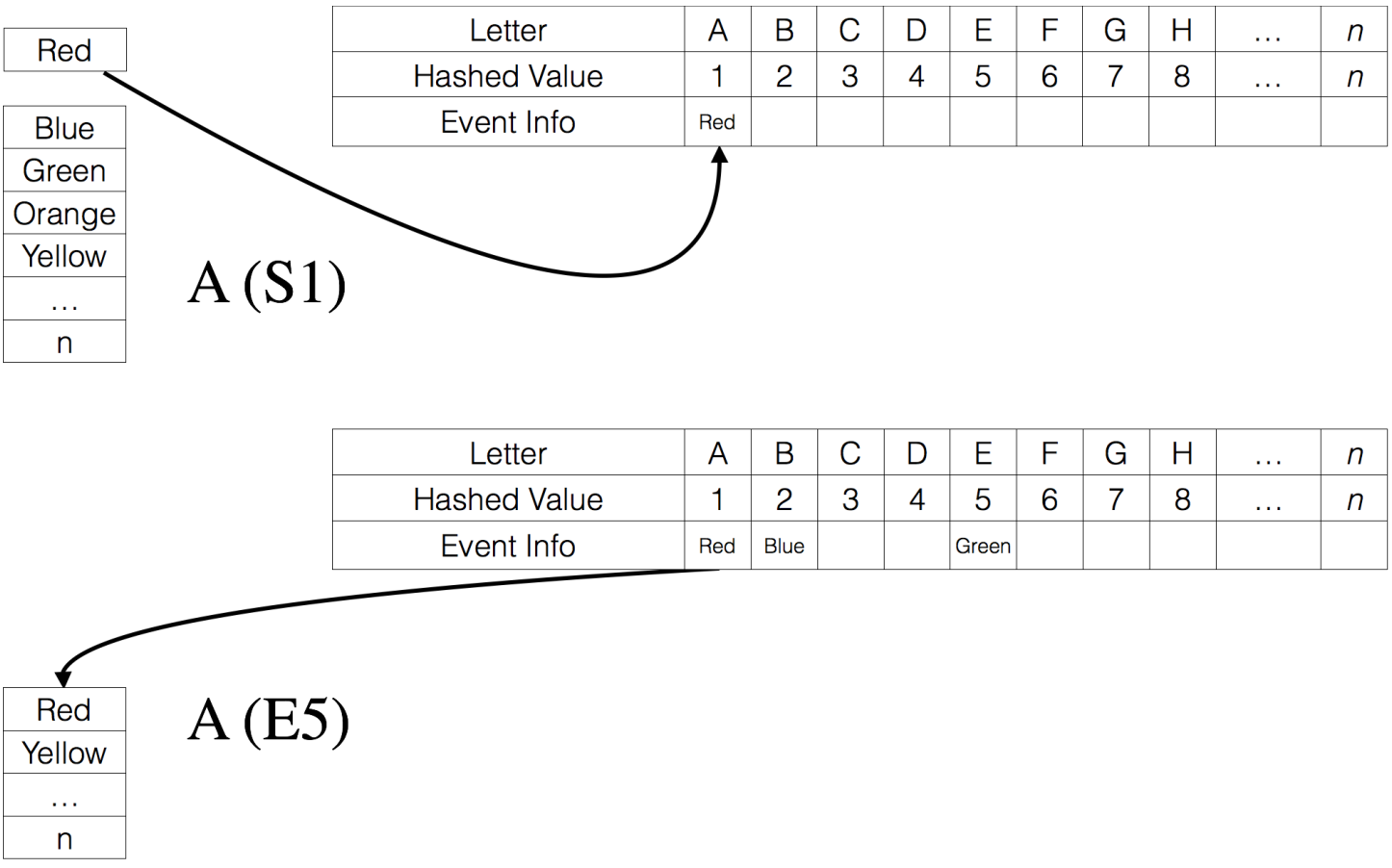
After creating the stack of colors, the coloring procedure iterates through the event list and whenever an event start is encountered, a color is popped off the stack and the event and its color are added to the hash table of length *n*.When a completion event is encountered, the color for that event is retrieved from the hash table and that color is pushed back onto the stack and made available for other events. This process only requires a single pass through the event list, which requires *n* O(1) lookups/insertions on the hash table, and only utilizes a stack of size *n*, therefore, this algorithm operates in O(*n*) time and space. Furthermore, because the algorithm utilizes a stack, each color is only reserved for as long as is necessary, resulting in a minimal number of colors used. Finally, because the events are sorted in 'first in' order, the algorithm always chooses the next occurring event, which is behavior of a greedy algorithm.

Using the colors and hash table shown on the right, the diagram below shows how the algorithm operates when the *start* and *end* events for *A* are encountered:

Red
Blue
Green
Orange
Yellow
…
n

Event Colors

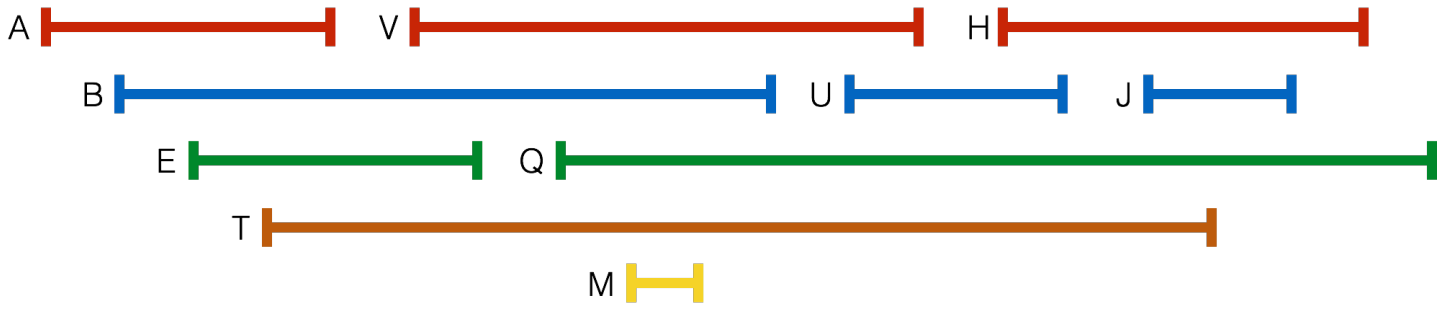
Letter	A	B	C	D	E	F	G	H	…	*n*
Hashed Value	1	2	3	4	5	6	7	8	…	*n*
Event Info										

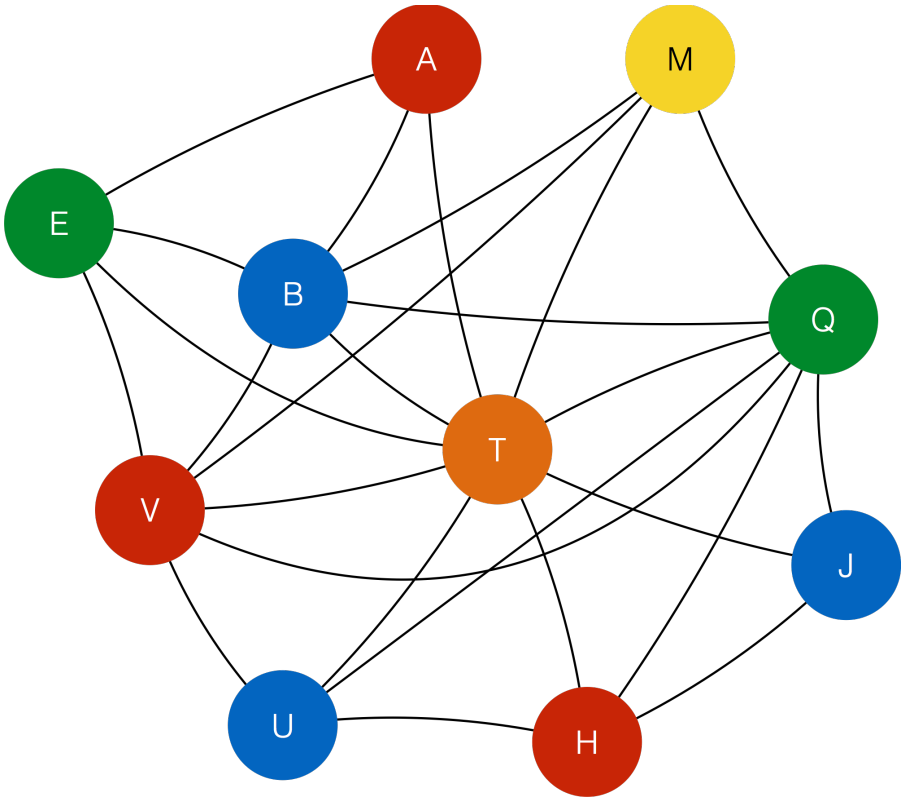


Applying this algorithm to the example event list results in the following hash table, interval graph, and colored graph:

Red
Blue
Green
Orange
Yellow
Purple

Letter	A	B	E	H	J	M	Q	T	U	V
Hashed Value	1	2	3	4	5	6	7	8	9	10
Event Info	Red	Blue	Green	Red	Blue	Yellow	Green	Orange	Blue	Red





Determining Degrees of all Vertices

To determine the degrees of all the vertices in O(*n*) time from the event list, the first step involves creating a data structure which will contain information about each event and will eventually hold information that will be used to calculate it's degree. This data structure will essentially be a hash table where each index points to an array of length 2. Each array will contain a value for the number of events that have *completed* (index 0) prior the beginning of the current event's start time and how many have *begun* (index 1) before the current event's end time. Once a given event's end time has been reached, it's degree can be calculated by taking the difference between the two values in its corresponding array in the hash table. In addition, this algorithm will utilize 2 global variables to keep track of the total number of events that have begun and the total number of events that have completed.

Once the data structure is created and the counter variables have been instantiated to 0, the algorithm will then begin iterating through the event list. Each time an event start is encountered, the letter value for the current event is hashed and the current value of the *completion counter* is added to the corresponding array the hash table points to, which represents the number of events that have finished before the current event started. Afterwards, the *start counter* is incremented by one. Each time an event end is encountered, the letter value for the event is hashed and the current value of the *start counter* is added to the corresponding array the hash table points to. This value represents the number of events that began before the current event completed. At this point, the degree of the event can be calculated by taking the difference of the number of events that began prior to completion and the number of events that completed prior to the current event's starting time.

The table below shows event list, which is created from the interval list, and it shows the values for each event's degree, the number of events that have finished prior to the current event, the number of events that began before the current event's completion, and the values of the counters at their associated points in time.

Event List:
A (S), B (S), E (S), T (S), A (E), V (S), E (E), Q (S), M (S), M (E), B (E), U (S), V (E), H (S), U (E), J (S), T (E), J (E), H (E), Q (E)

Interval List	Finished Before Starting	Began Before Completion	Degree	Completion Counter (At Start Time)	Start Counter (At Completion Time)
A: (1,5)	0	3	**3**	0	3
B: (2,11)	0	6	**6**	0	6
E: (3,7)	0	4	**4**	0	4
T: (4,17)	0	9	**9**	0	9
V: (6,13)	1	7	**6**	1	7
Q: (8,20)	2	9	**7**	2	9
M: (9,10)	2	6	**4**	2	6
U: (12,15)	4	8	**4**	4	8
H: (14,19)	5	9	**4**	5	9
J: (16, 18)	6	9	**3**	6	9

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|).
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.

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

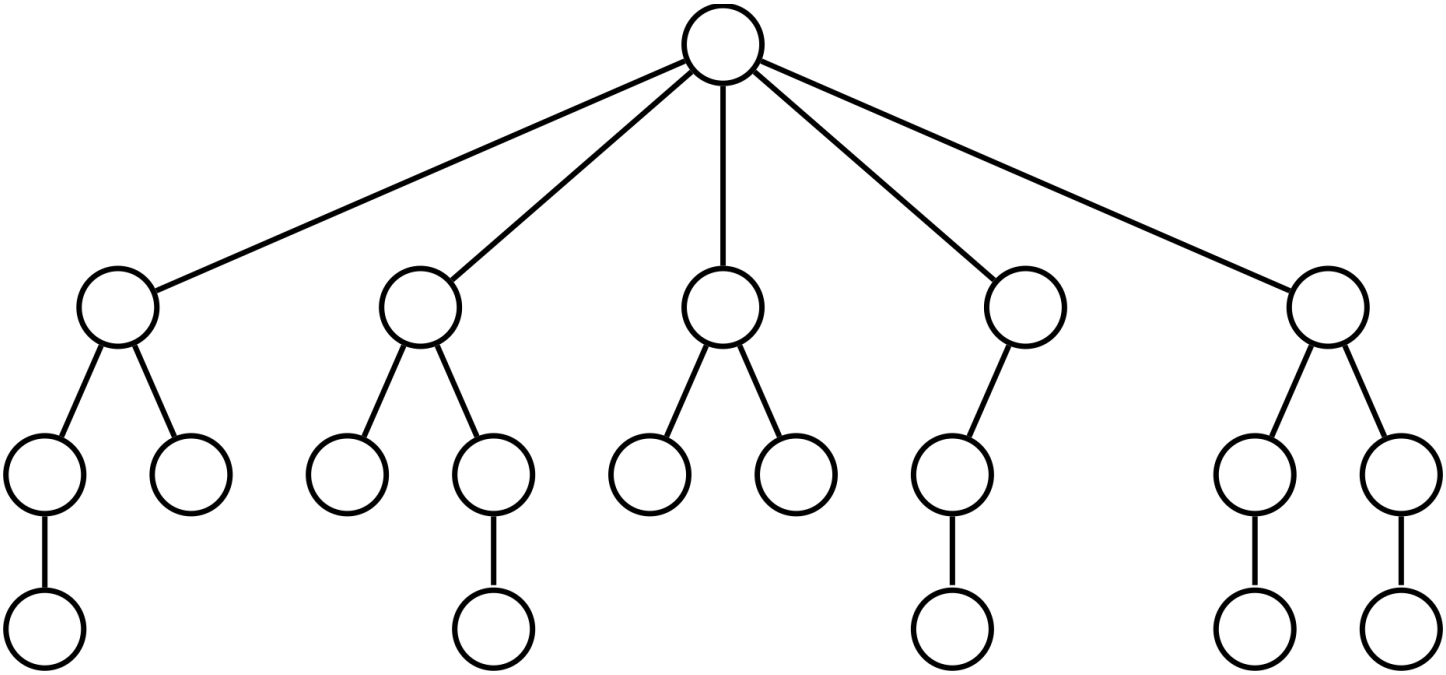
Finding a maximum independent set in a tree with a time complexity of O(|V|) sounds like a very challenging problem, however, in reality it is relatively simple. This problem may be solved through the use of a breadth first search and a greedy algorithm.

The first step in this process involves performing a breadth first search through the tree to obtain a list of nodes and their associated degrees. A breadth first search has a worst case time complexity of O(|V|), therefore, this algorithm begins by operating as efficiently as required. During the search, a hash table is used to place all nodes into a bucket of their associated degree. At this point, a single integration through the hash table is required to greedily find the first bucket containing nodes, which are the nodes with minimum degree. These nodes are then painted to indicate they have been added to the maximum independent set (MIS). Finally, parents of these nodes are painted with a different color to indicate they have been removed from the potential pool of nodes that may be included in the MIS. Both painting processes incur a time complexity of O(|V|).

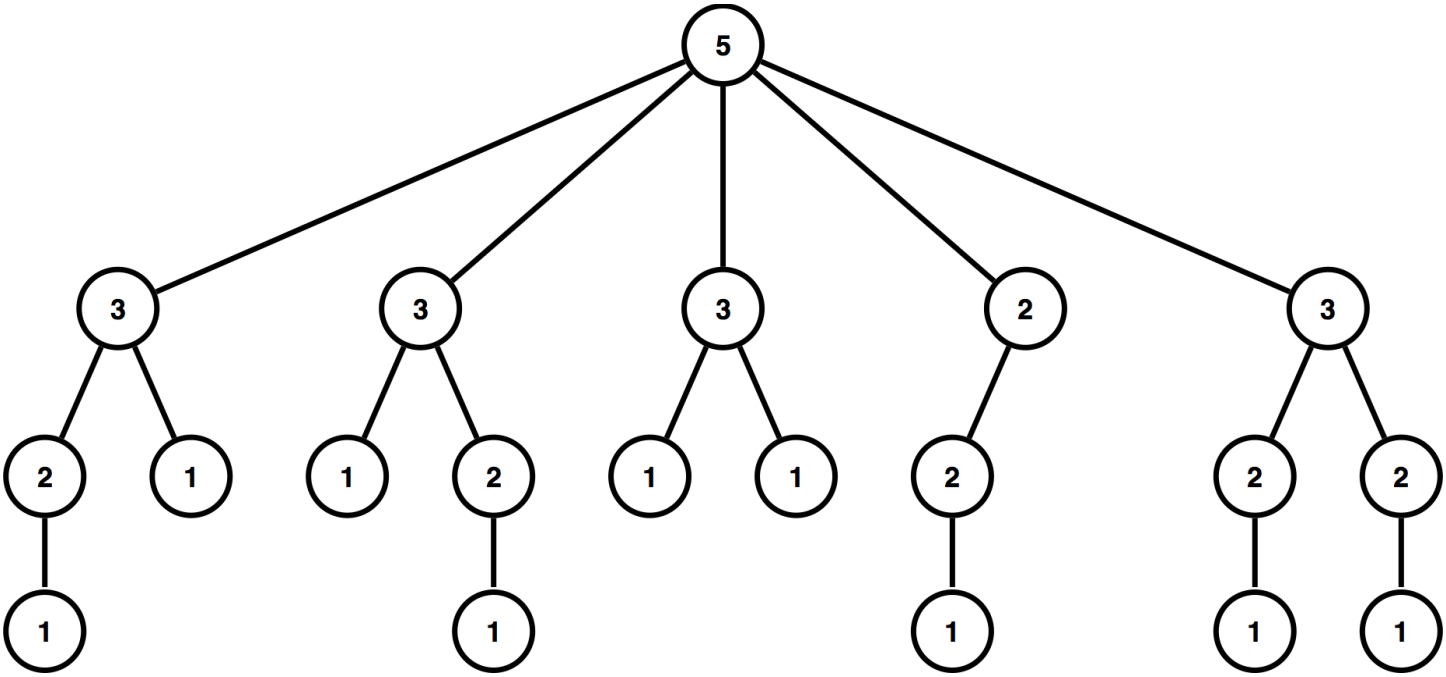
The process then continues, however, while performing subsequent breadth first searches through the tree, nodes that have been pained are not included in the hashing process. This ensures that the tree still maintains its structure and that no nodes which have been painted as *not* being a part of the MIS are not accidentally included. The algorithm continues in this manner until the hashing process yields an empty table.

Throughout this process, all components of the algorithm adhere to the time complexity restriction of O(|V|), therefore, the overall time complexity of this algorithm is O(|V|).

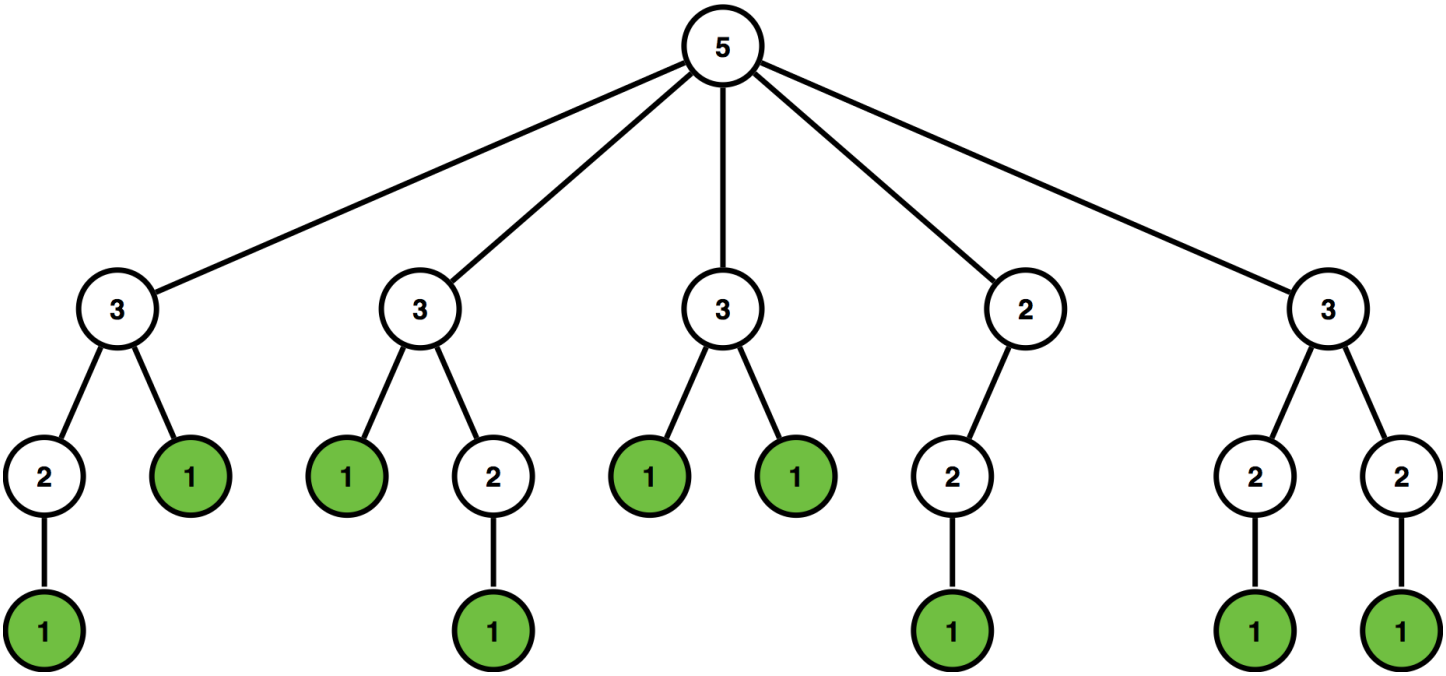
The diagrams below walkthrough the algorithm:



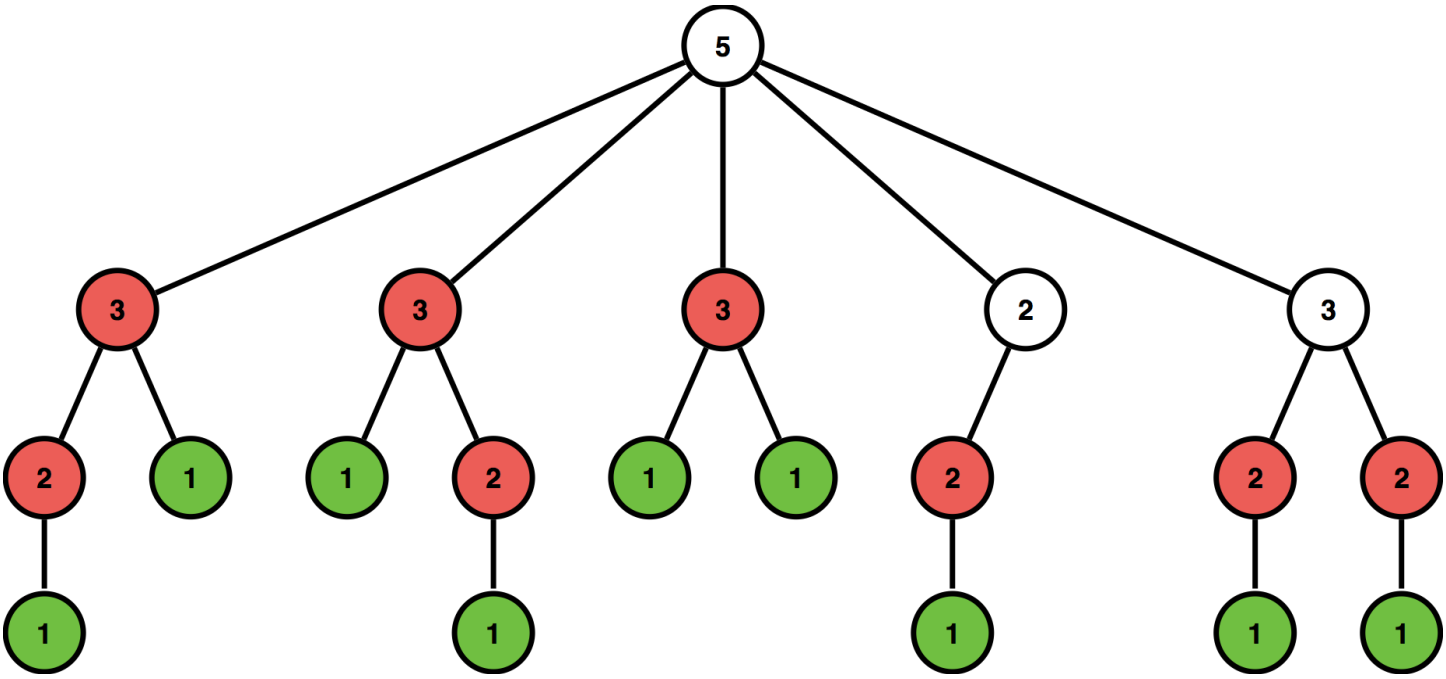
The initial structure of the graph



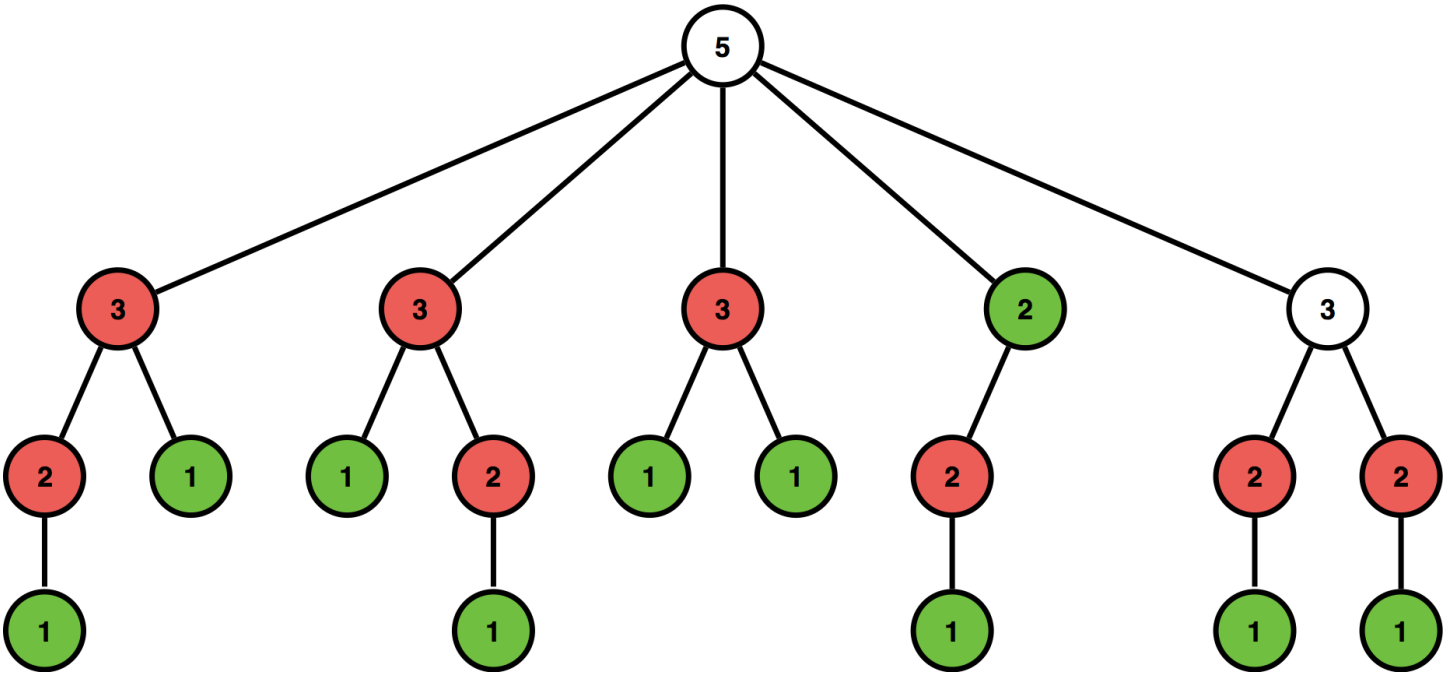
After performing the breadth first search, the degree of each node is recorded and nodes are bucketed based on this value. This diagram indicates the degree of each node, which in turn, indicates the bucket each node is placed in.



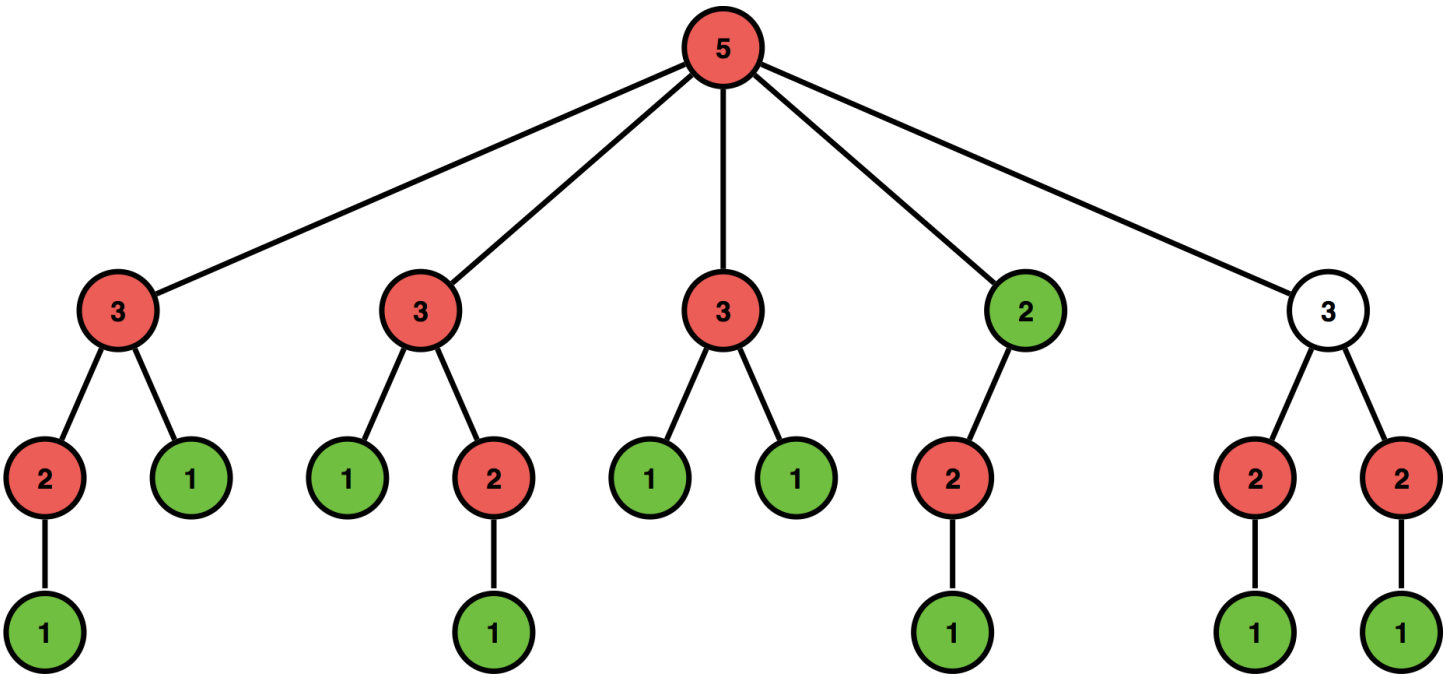
Nodes in the first bucket are retrieved and added to the maximum independent set. Each node is then painted to indicate it has been added.



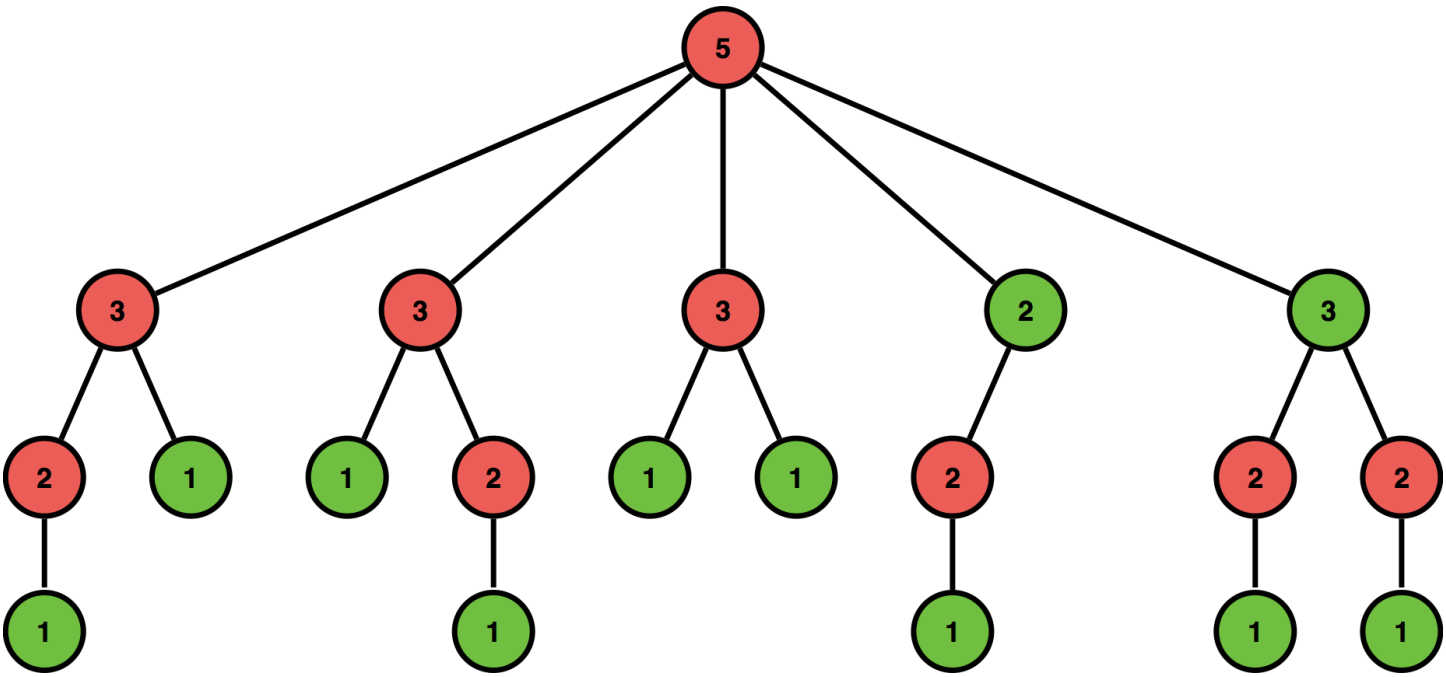
Parent nodes are painted to indicate they cannot be added to the maximum independent set.



Another breadth first search is performed. Only nodes which are *not* painted are hashed and added to the hash table. After iterating through the hash table, the smallest bucket containing nodes is then added to the MIS and painted accordingly.



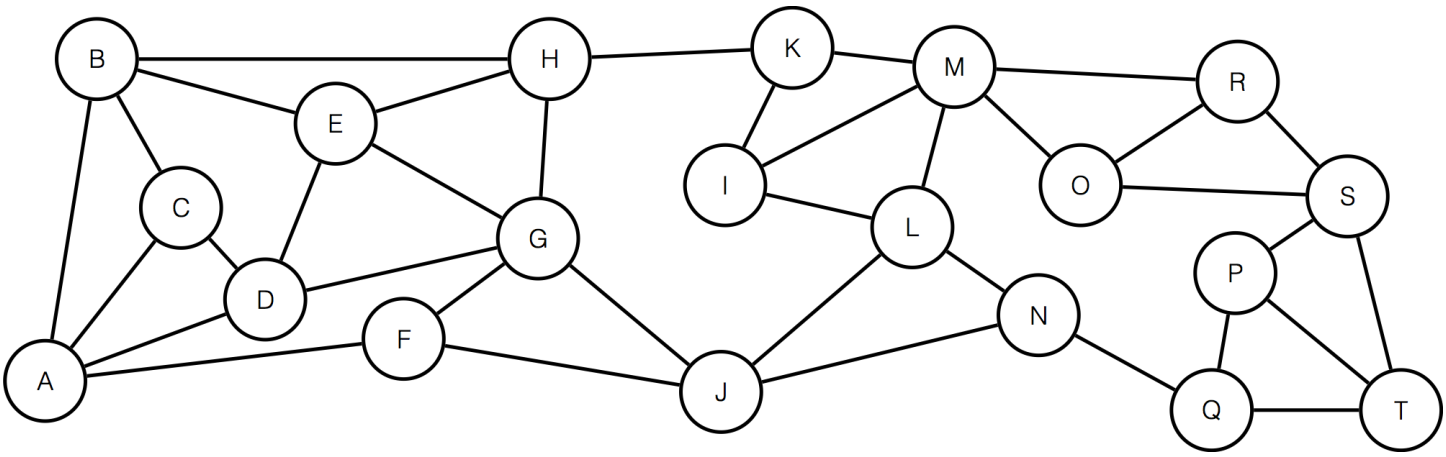
Again, parent nodes are painted to indicate they cannot be included in the hashing process and the subsequent MIS.



The process continues until the breadth first search results in an empty hash table. At this point, the maximum independent set has been created.

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.

Using maximum adjacency search and by painting both the vertices and the edges, a pair of "k-connected" vertices can be found in O(|V| + |E|). The diagram below illustrates the initial graph which contains 20 vertices with a minimum degree of 3 and a maximum degree of at least 5:



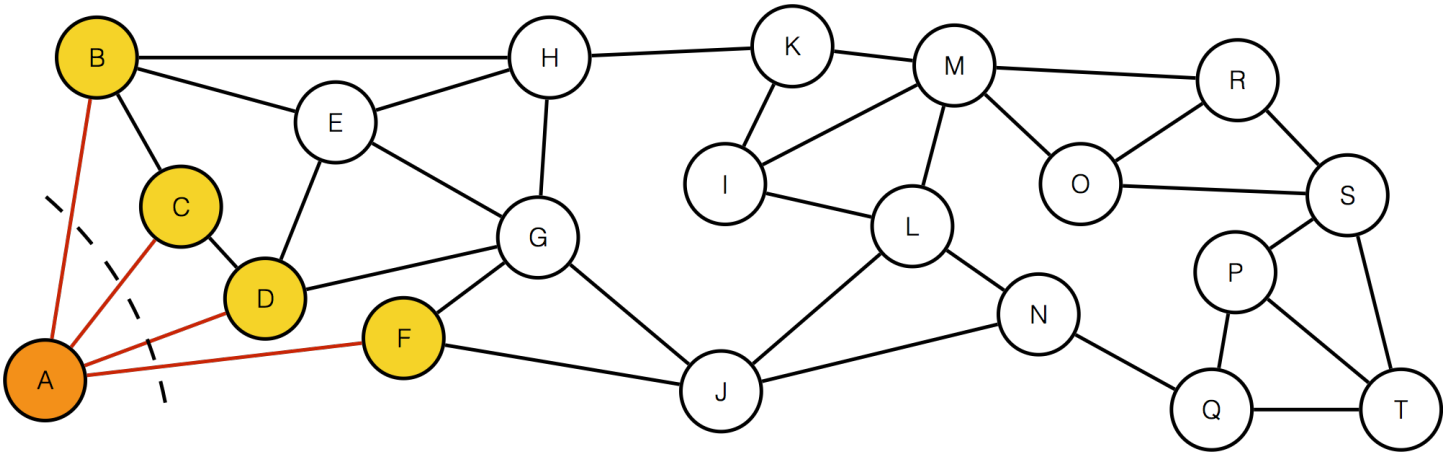
Performing a maximum adjacency search with ties broken lexicographically through this graph results in the following ordering:

A, B, C, D, E, G, H, F, J, K, I, L, M, N, O, R, S, P, Q, T

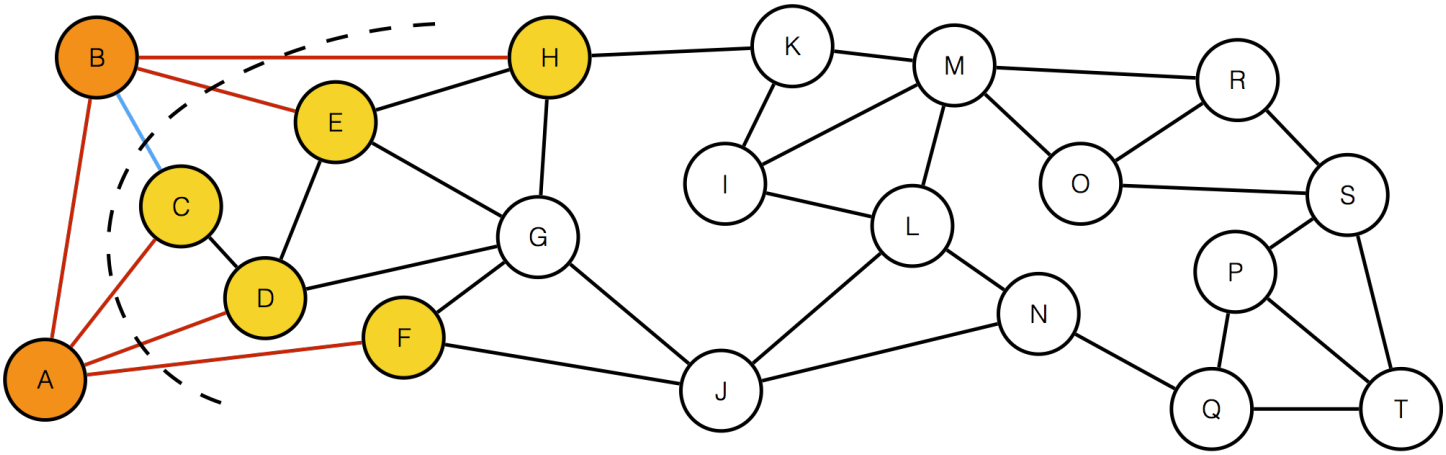
During this search, each node visited is painted to indicate that it has been visited and each node it is connected to is marked as having been reached. Furthermore, each edge connecting the current vertex and it's connected vertices is painted depending on how many times the connected vertices have been reached.

The maximum adjacency search with painting of vertices and edges continues until the final node has been reached. At this point, the number of edge-disjoint paths between nodes can be found by following the different painted edges.

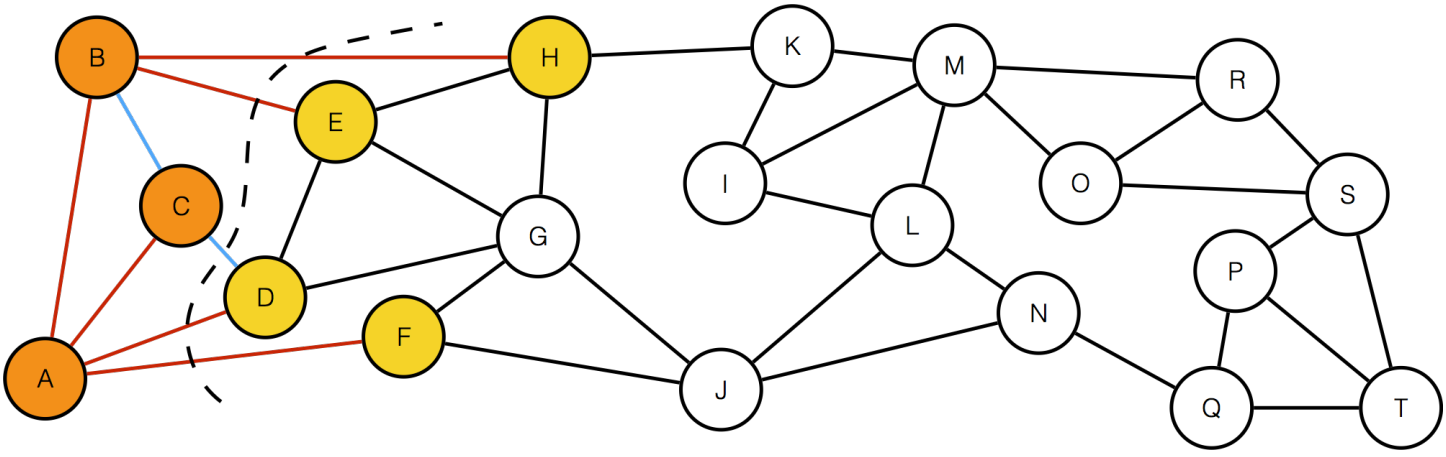
The diagrams on the proceeding pages illustrate the process of coloring the graph.

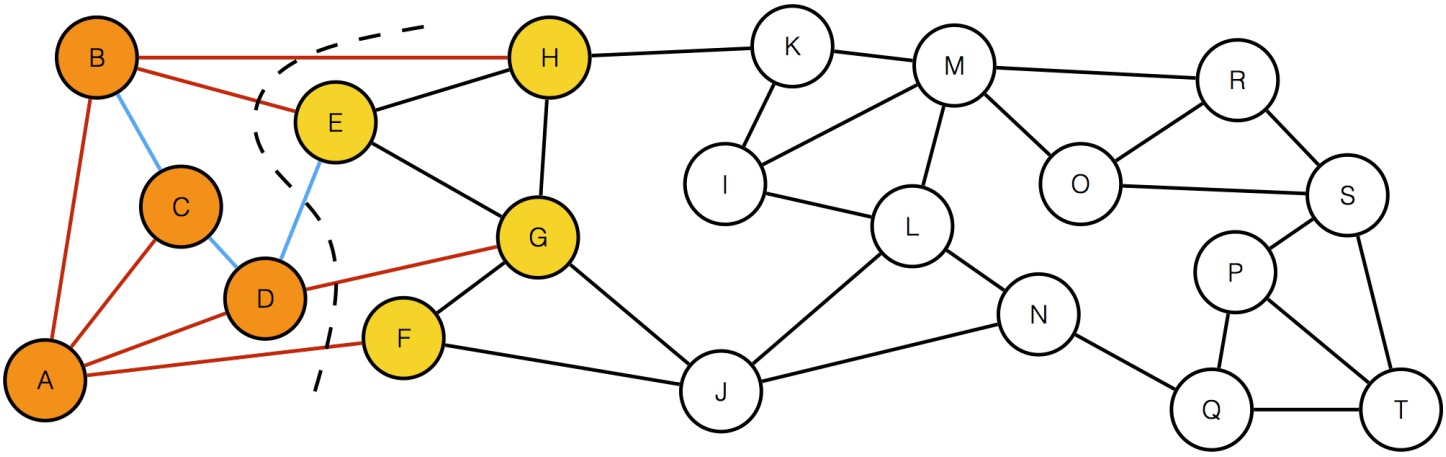


The maximum adjacency search begins with node A. A is painted orange to indicate it has been visited. Nodes B, C, D, and F are reached for the first time and are painted yellow. Because they are being reached for the first time, the edges connecting them to A are marked with the first color (red). The dashed line represents the cut.

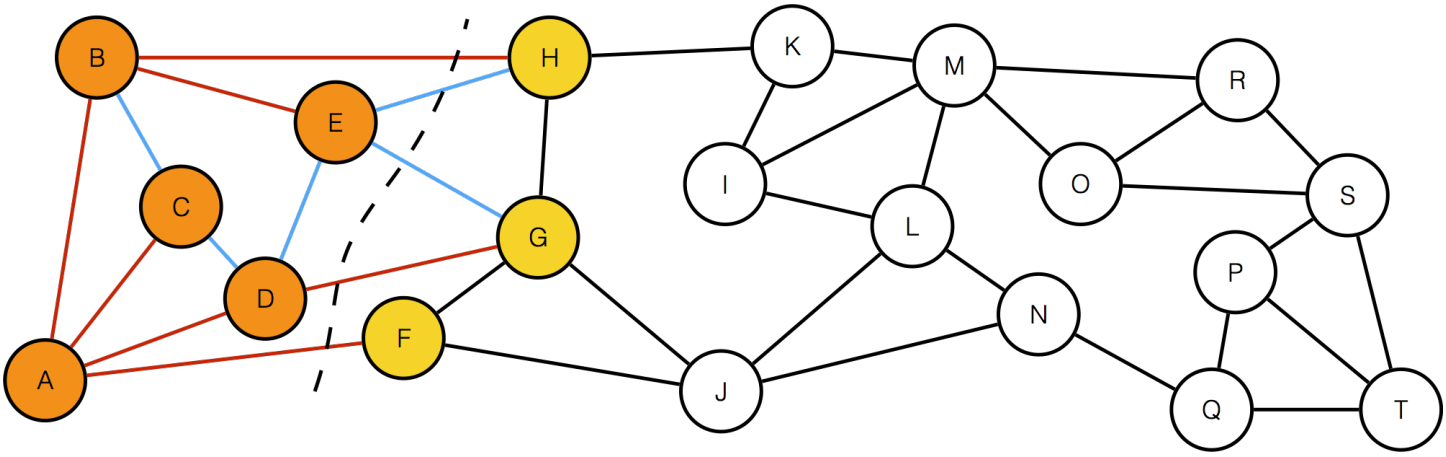


B is visited and is painted orange. C is reached for the second time, so the edge connecting B and C is painted with the next available color. E and H are reached for the first time.

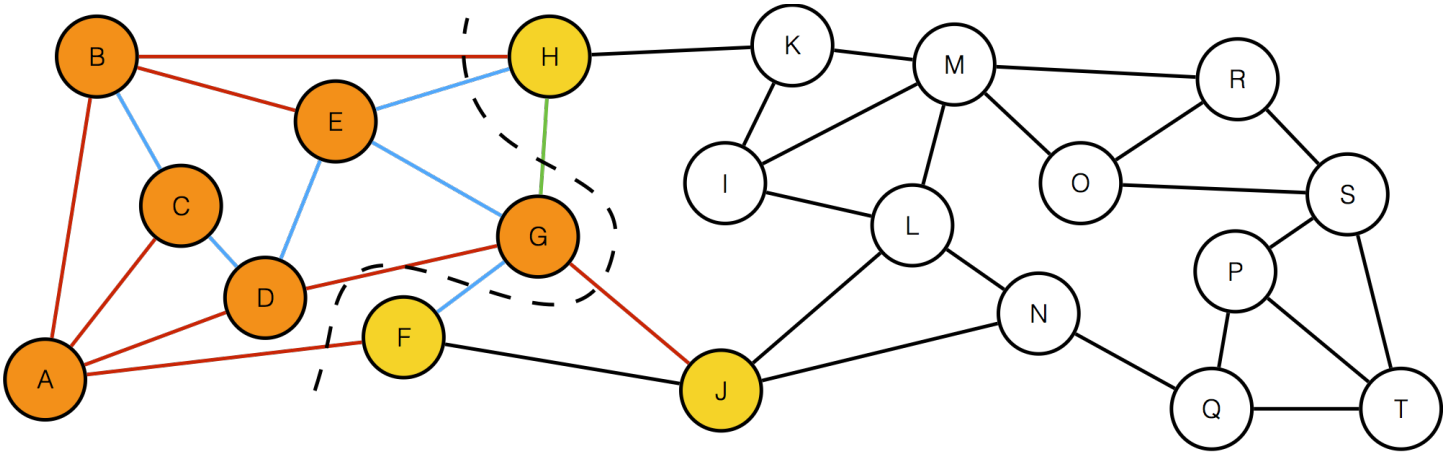




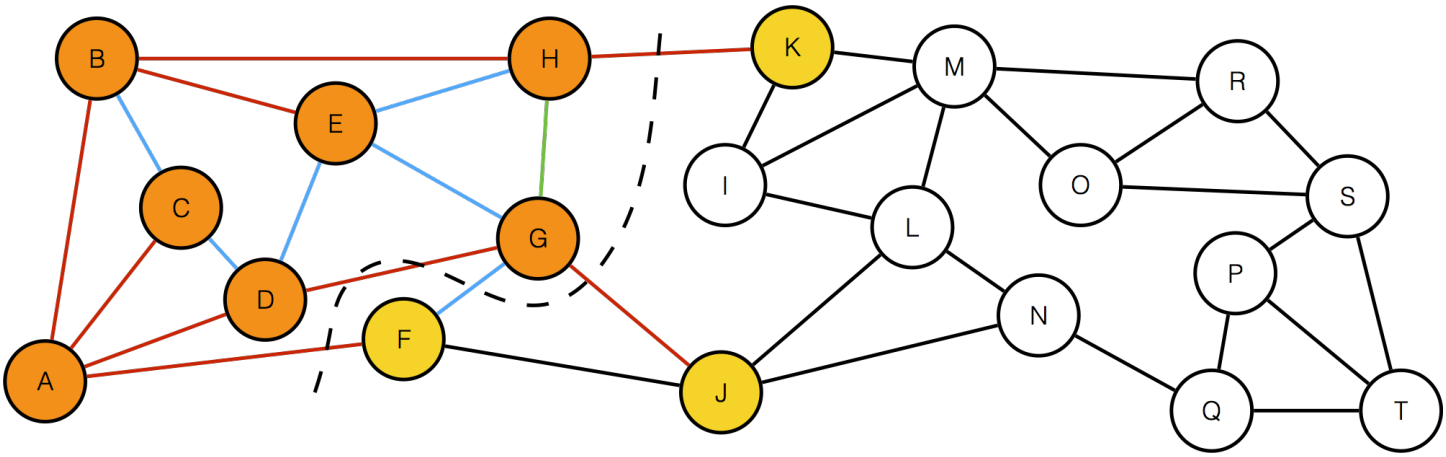
The maximum adjacency search begins with node A. A is painted orange to indicate it has been visited. Nodes B, C, D, and F are reached for the first time and are painted yellow. Because they are being reached for the first time, the edges connecting them to A are marked with the first color (red). The dashed line represents the cut.

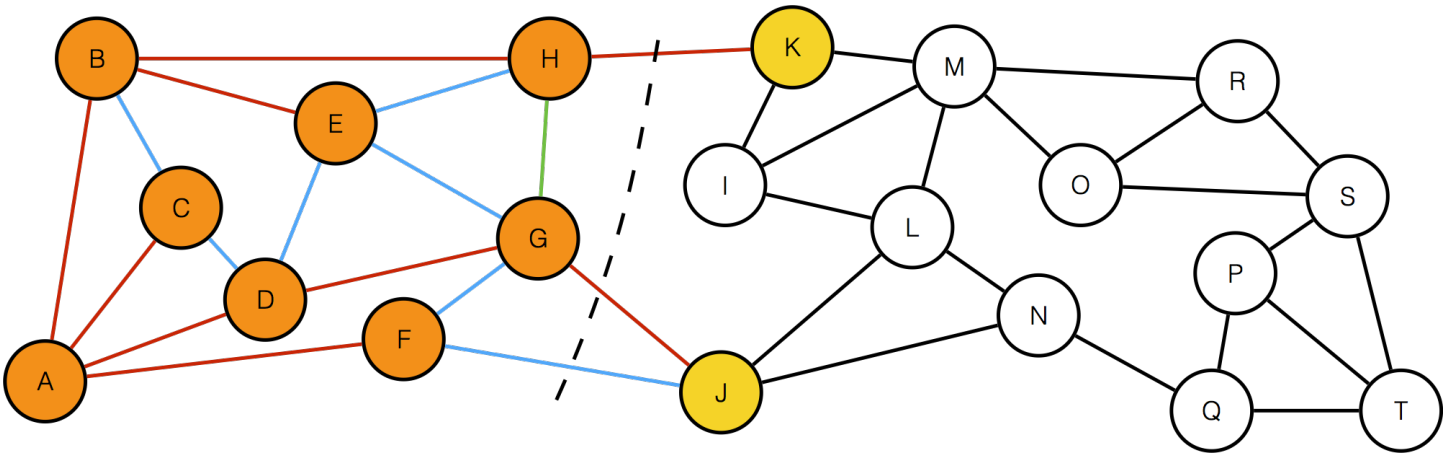


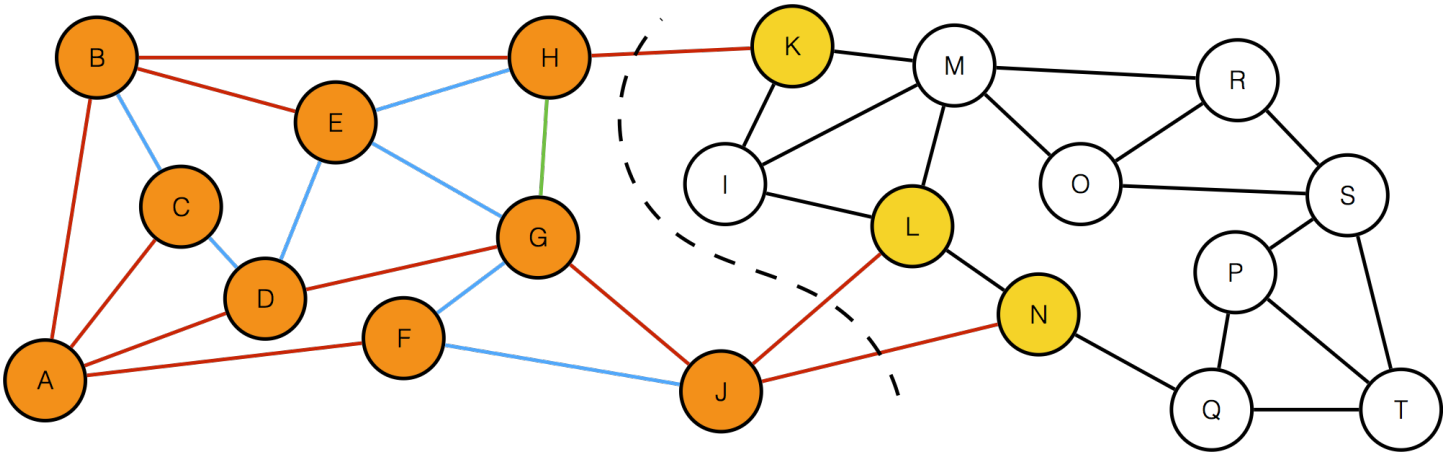
The maximum adjacency search begins with node A. A is painted orange to indicate it has been visited. Nodes B, C, D, and F are reached for the first time and are painted yellow. Because they are being reached for the first time, the edges connecting them to A are marked with the first color (red). The dashed line represents the cut.

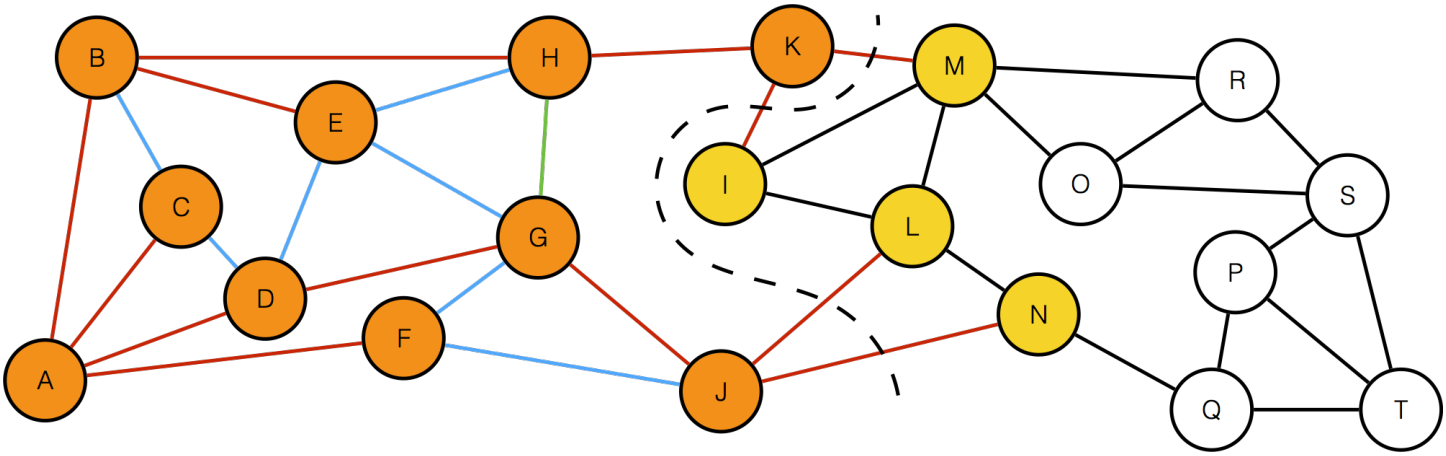


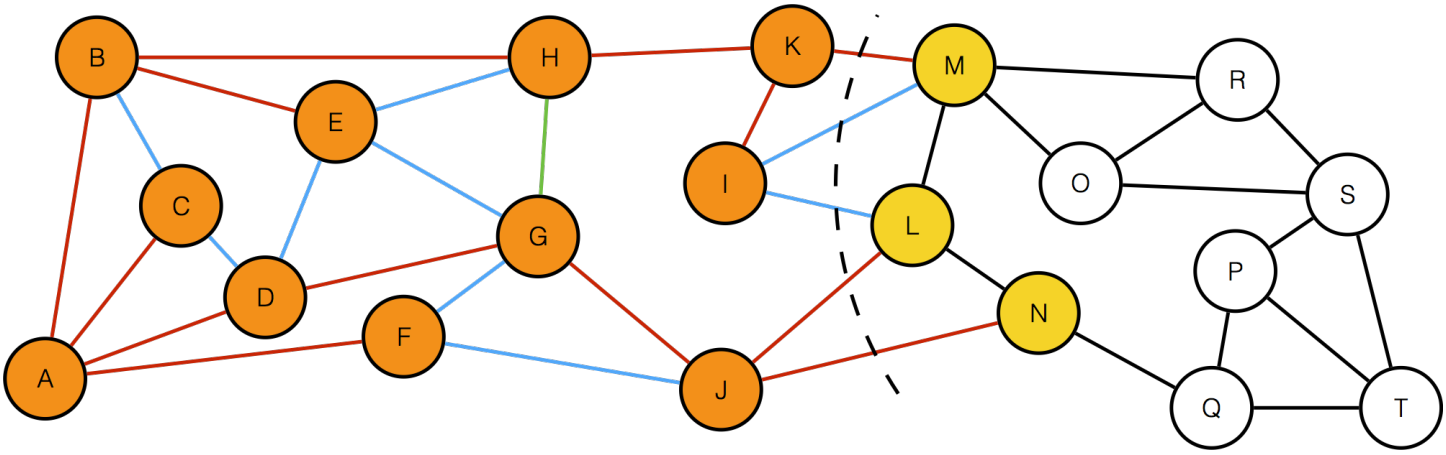
H is reached for the third time, so the edge connecting G and H is painted with the next available color.

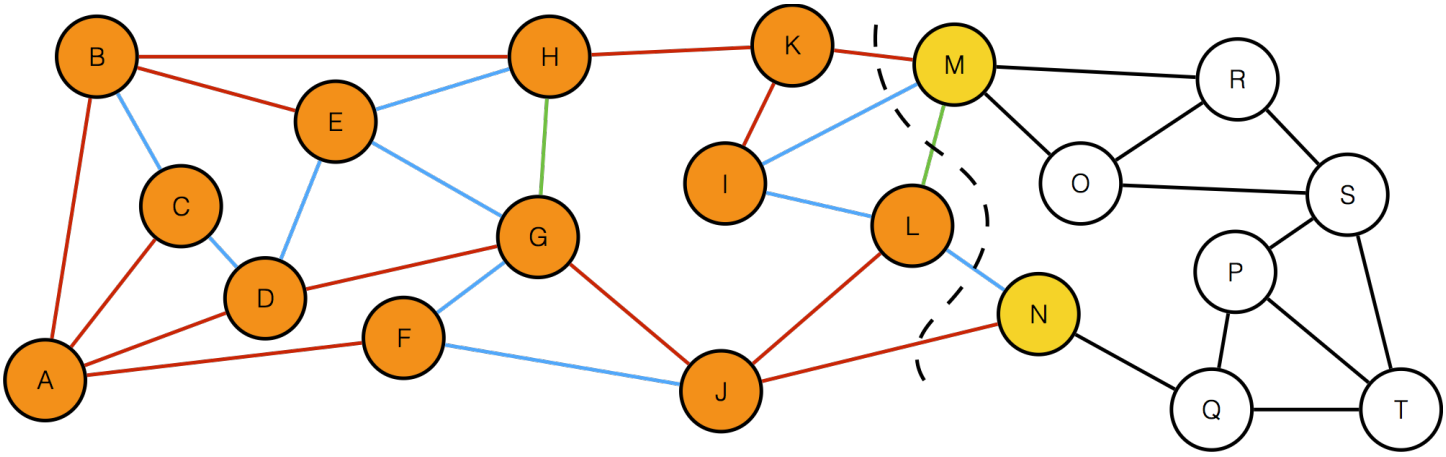

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T

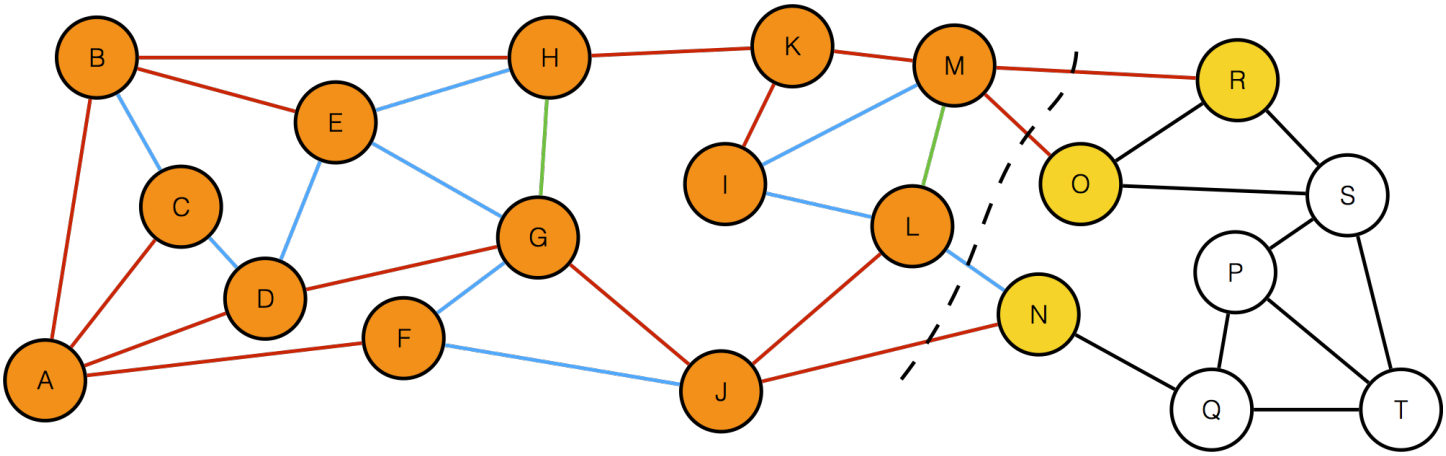

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T

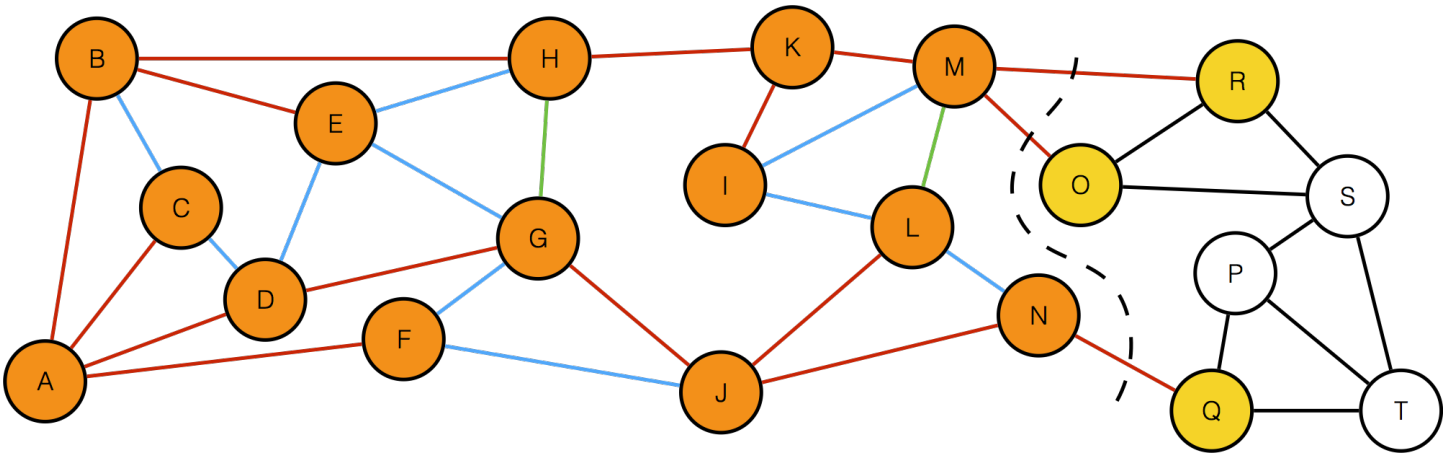

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T

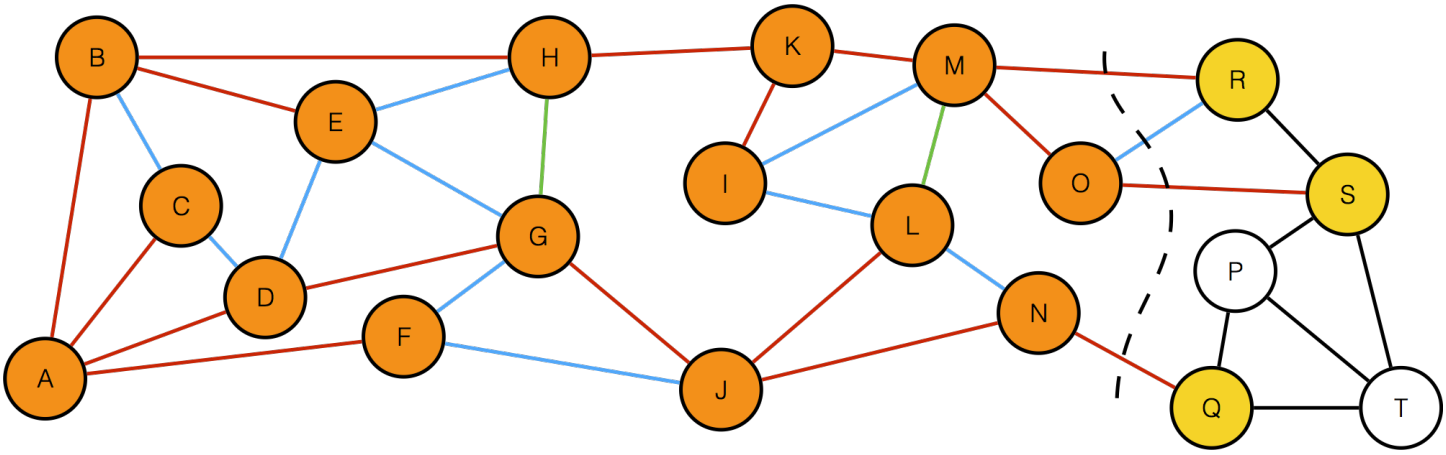

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T

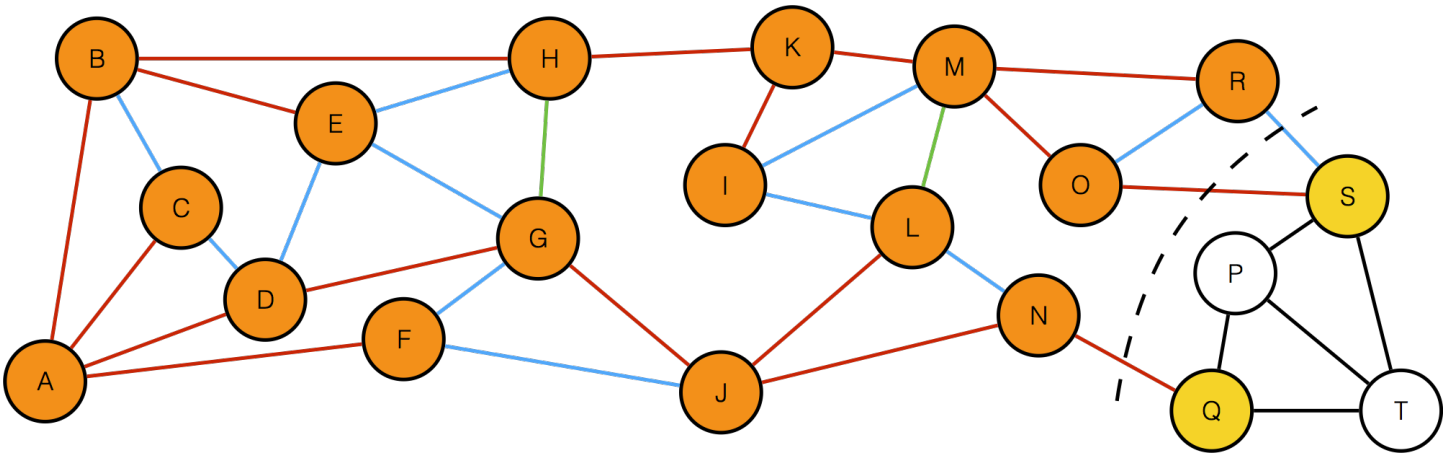

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T

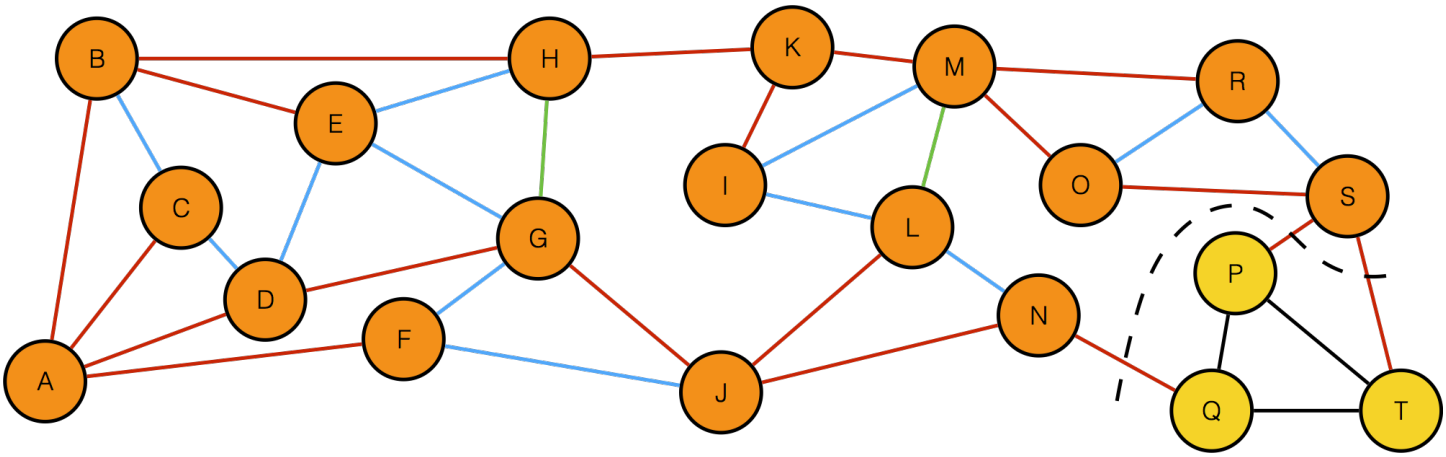

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T

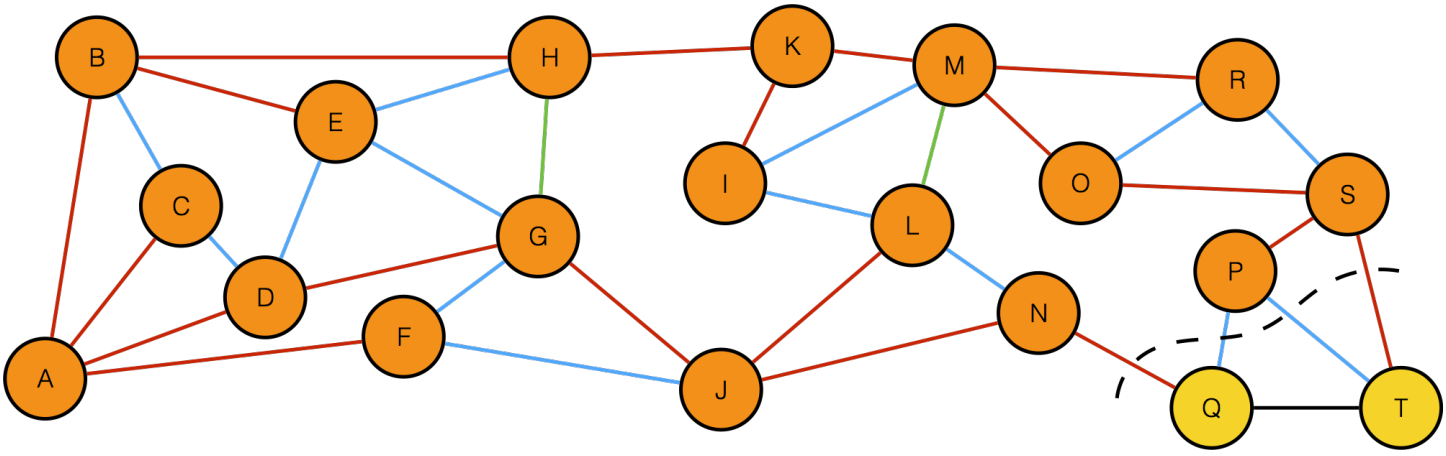

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T

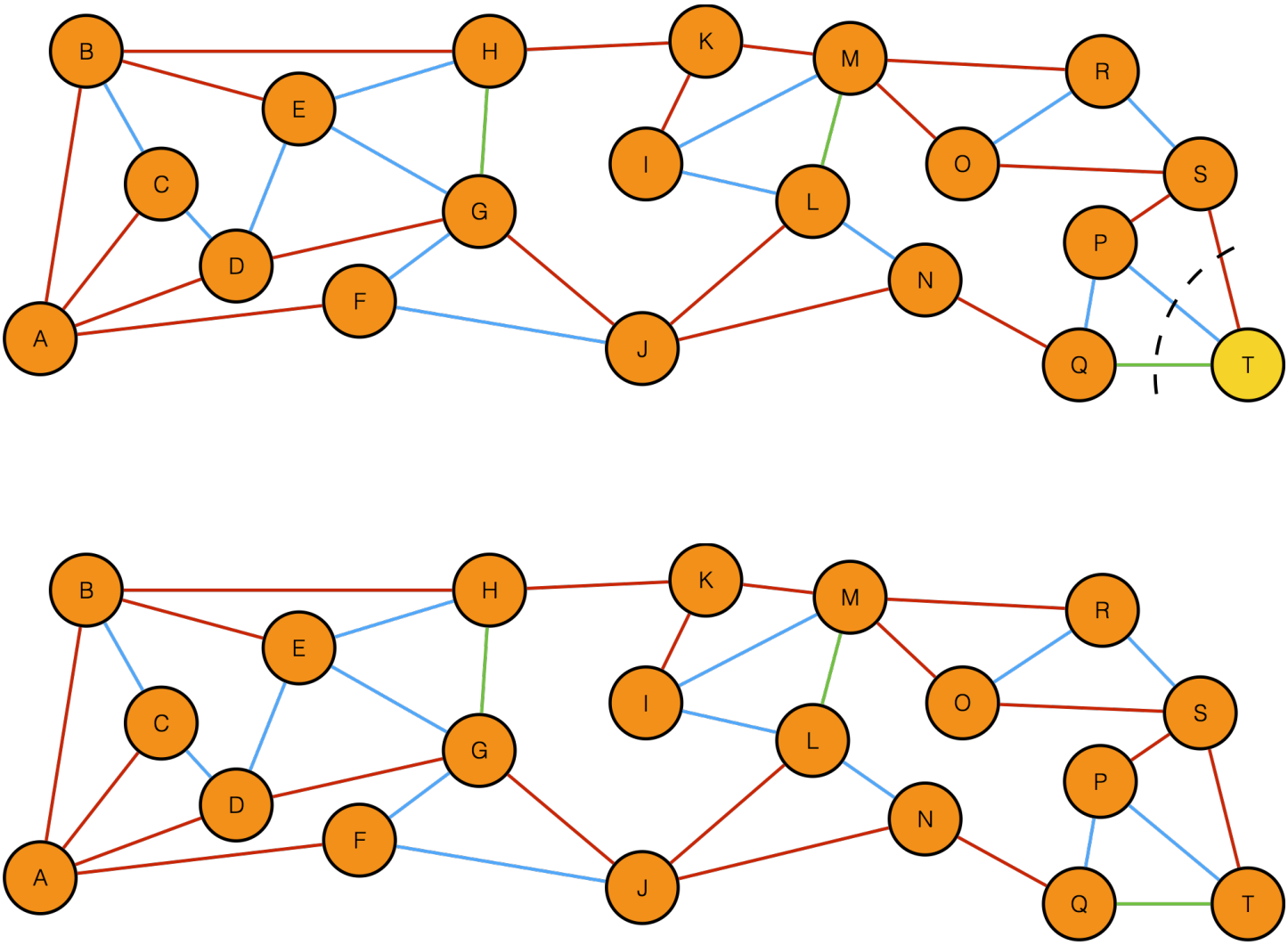

B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T


B
H
K
M
R
E
I
O
S
C
G
L
P
D
N
F
A
J
Q
T


B
H
K
M
R
E
I
O
S
C
L
G
P
D
N
F
A
J
Q
T


B
H
K
M
R
E
I
O
S
C
L
G
P
D
N
F
A
J
Q
T

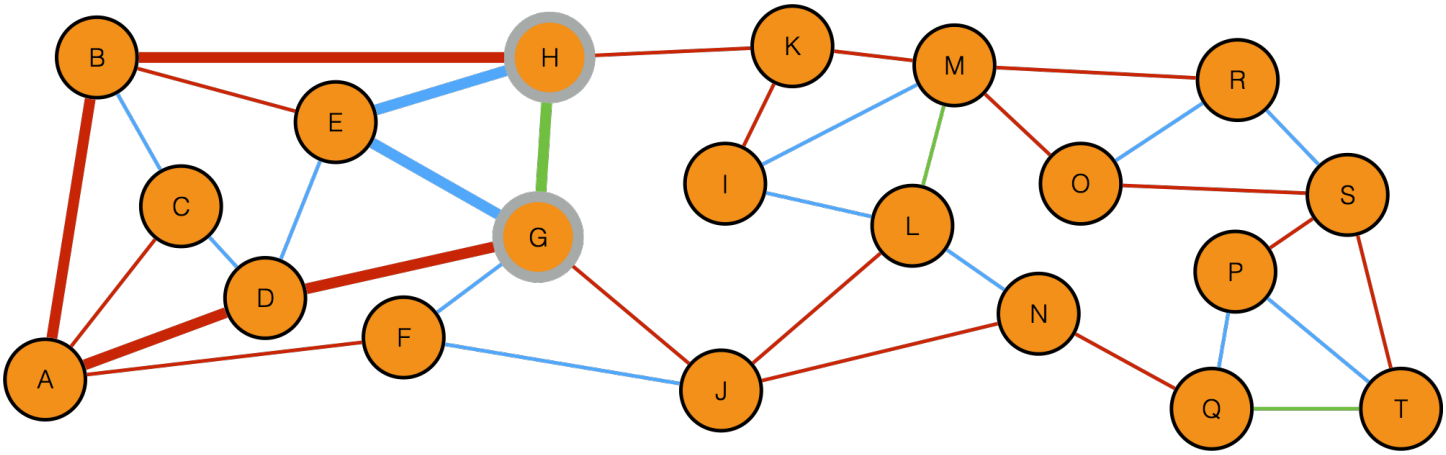

B
H
K
M
R
E
I
O
S
C
L
G
P
D
N
F
A
J
Q
T



The final node in the MAS order, T, is reached and is painted as visited.

The diagram above, which is the last step in the maximum adjacency search process, represents all the sets of pairwise k-edge connected vertices using the “k-1 cycle”. The number of edge-disjoint paths between any two vertices in the graph can be found by taking the minimum degree of the two vertices and then subtracting one from it. These paths are represented by the colored edges throughout the graph.

Using the colored edges, it is possible to determine the number of edges between two specific vertices. The example below shows the potential edge-disjoint paths between nodes G and H, which are represented as thicker lines.

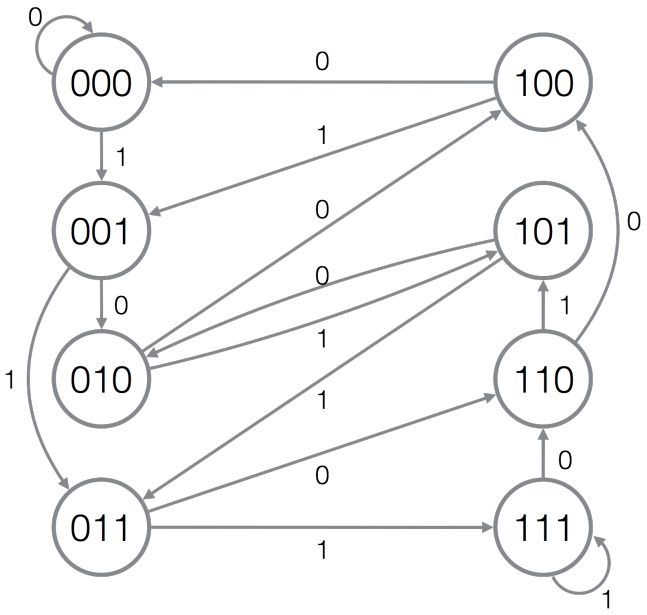


3. [***Matching Substrings***]
A matching substring pair of length k in a binary bit string $b_1, b_2...b_n$ is a pair $b_ib_{i+1}... b_{i+k-1} = b_jb_{j+1}...b_{j+k-1}$, with i not equal to j, of identical k-bit substrings. Determine a maximum length binary string with no matching substrings of length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

Reference: See notes on web page.

To determine the maximum length binary string with no matching substrings of length k, a graph structure resembling a state machine can be used. While traversing the string, this structure can be used to determine which binary bit combinations have been found and, eventually, which have been repeated.

In a binary bit string with substrings of length 4, there is a total of 16 possible substring combinations (2^4). Of these combinations, there is a total of 8 possible *k-1* length substring stems (2^3), which are the first *k*-1 bits of the *k*-length substring. In the state machine structure, each of these stems is a vertex which is connected with two additional vertices. The edges that connect these vertices have a specific direction and represent the next bit in the string. When the new bit is added to the *k*-1 stem, the *k*-bit substring is truncated from the 2nd bit until the kth bit, which becomes the new stem. The structure for determining a maximum length binary string with no matching substrings of length 4 is shown below.



While examining the binary string and traversing the structure, each edge taken is marked as having been 'taken'. During the traversal, when a specific bit, which will be referred to as *z*, is encountered and and the edge corresponding to *z*'s value has already been marked as 'taken', the *k*-bit substring ending in *z* is the first matching substring in the binary string. At this point, the length of the maximum length binary string with no matching substrings of length k is equal to the sum of the number of edges taken and *k*-1, which accounts for the number of bits in the initial stem.

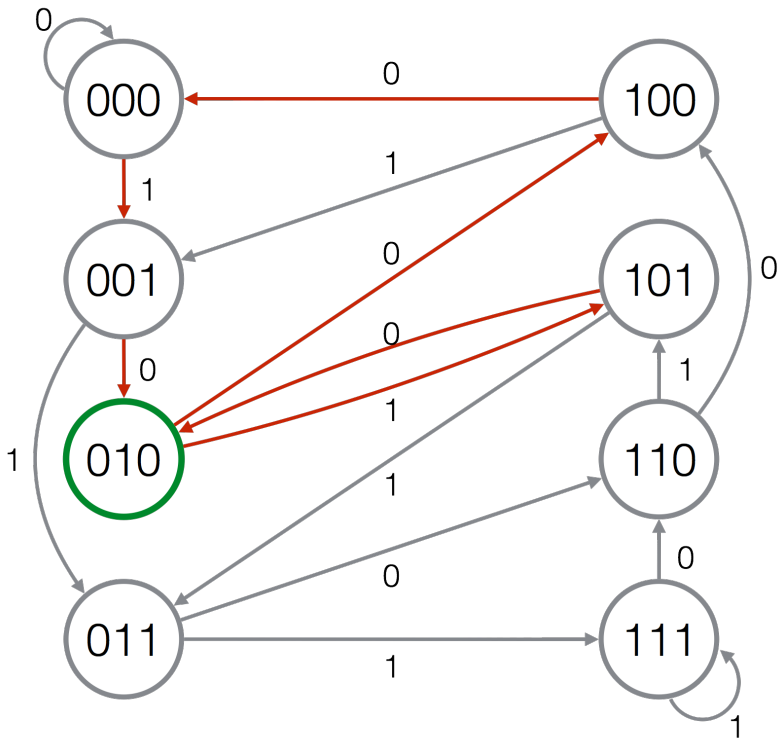
Overall, the maximum length binary string with no matching substrings of length 4 can be found by taking every edge in the above graph a single time starting with a initial stem of 3 bits. This results in a binary string of length 19 (16 edges + 3 bits in the stem).

The diagram below demonstrates how the structure is used with 4-bit substrings and a randomly generated binary string example.

Bit String: 01010001001010

Initial Stem: 010

First Match: 0100 (01 **0100 0100** 1010)



Number of Edges Taken: 6

k-1: 3

Maximum length with no matching substrings: 6 + 3 = 9

In the structure above, taken edges are represented in red. Once the *z*-bit is encountered, which attempts to take an edge that is marked as "taken", a matching substring has been found. To calculate the number maximum length of the string with no matches, sum the total number of edges taken (which will have a linear time complexity programmatically through the use of a counter) and *k*-1, resulting in a string length of 9 for this example.

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

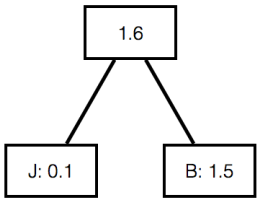
Note that the letters are arranged in decreasing frequency. What is the time complexity of building such a Huffman coding tree for an n letter alphabet arranged in decreasing frequency order?

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

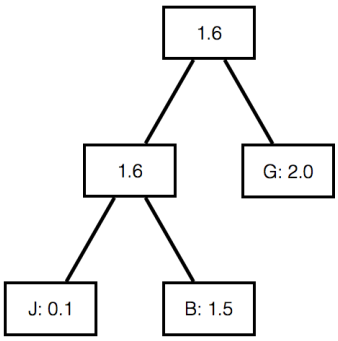
Because we are building the Huffman coding tree from the bottom up, each element is only accessed once. Furthermore, because the letters are already arranged in decreasing order, we do not incur additional time complexity due to sorting. Under the assumption that each subgraph created by the combination of two letters is *not* inserted into the original data structure (which could potentially incur a cost of O(*n*) for insertion), the overall complexity to create the Huffman coding tree is O(*n*).

The diagram on the right shows the initial letter frequencies in table form, and the diagrams below show the visualization of the subgraphs in table form and in tree form throughout the compression process.

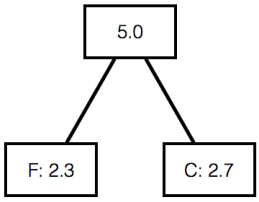
Letter	Frequency
E	12.0
A	8.1
I	7.3
H	5.9
D	4.3
C	2.7
F	2.3
G	2.0
BJ	1.6

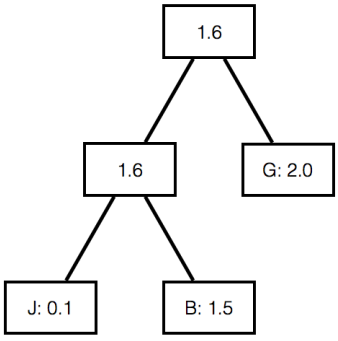


Letter	Frequency
E	12.0
A	8.1
I	7.3
H	5.9
D	4.3
BGJ	3.6
C	2.7
F	2.3

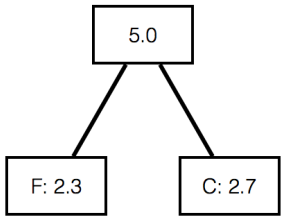


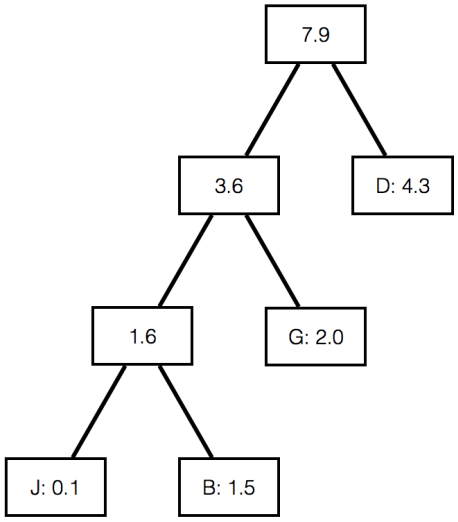
Letter	Frequency
E	12.0
A	8.1
I	7.3
H	5.9
CF	5.0
D	4.3
BGJ	3.6



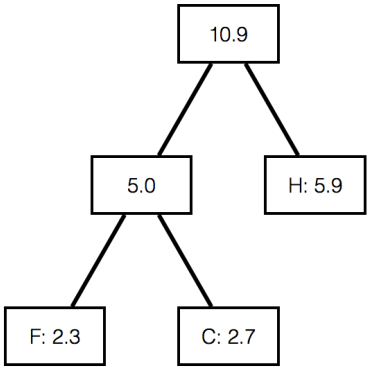


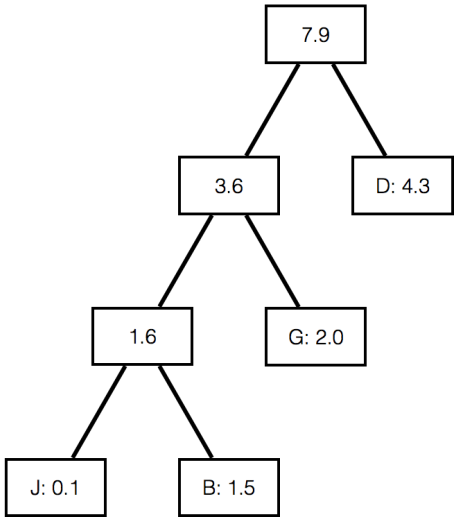
Letter	Frequency
E	12.0
A	8.1
BDGJ	7.9
I	7.3
H	5.9
CF	5.0



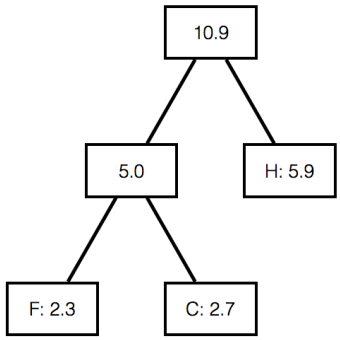


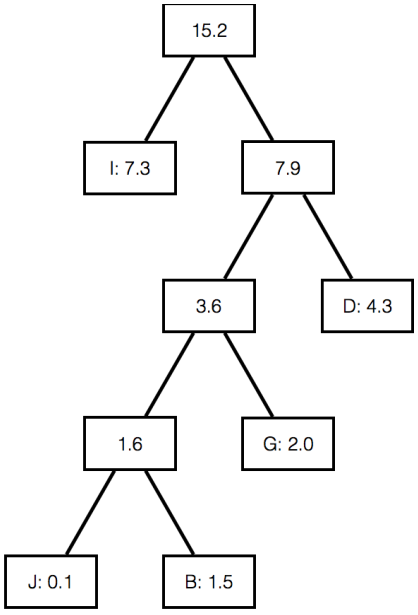
Letter	Frequency
E	12.0
CFH	10.9
A	8.1
BDGJ	7.9
I	7.3



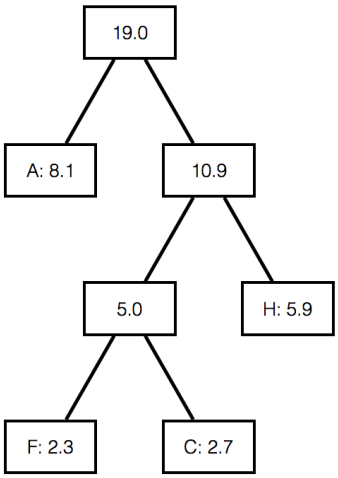


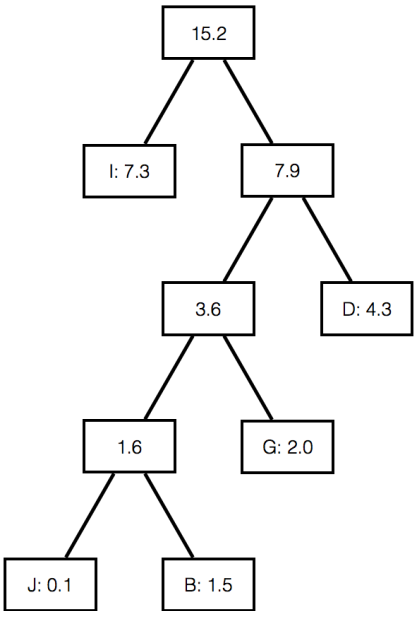
Letter	Frequency
BDGIJ	15.2
E	12.0
CFH	10.9
A	8.1



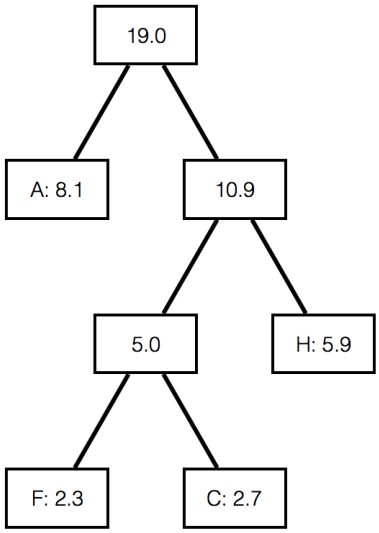


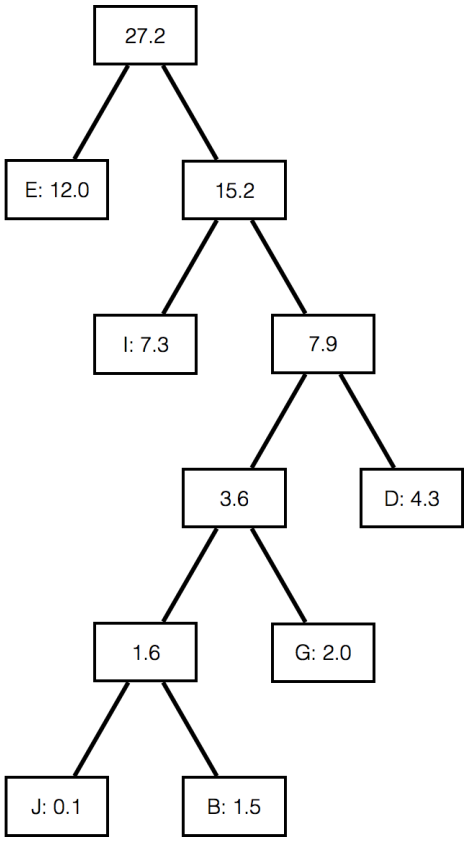
Letter	Frequency
ACFH	19.0
BDGIJ	15.2
E	12.0



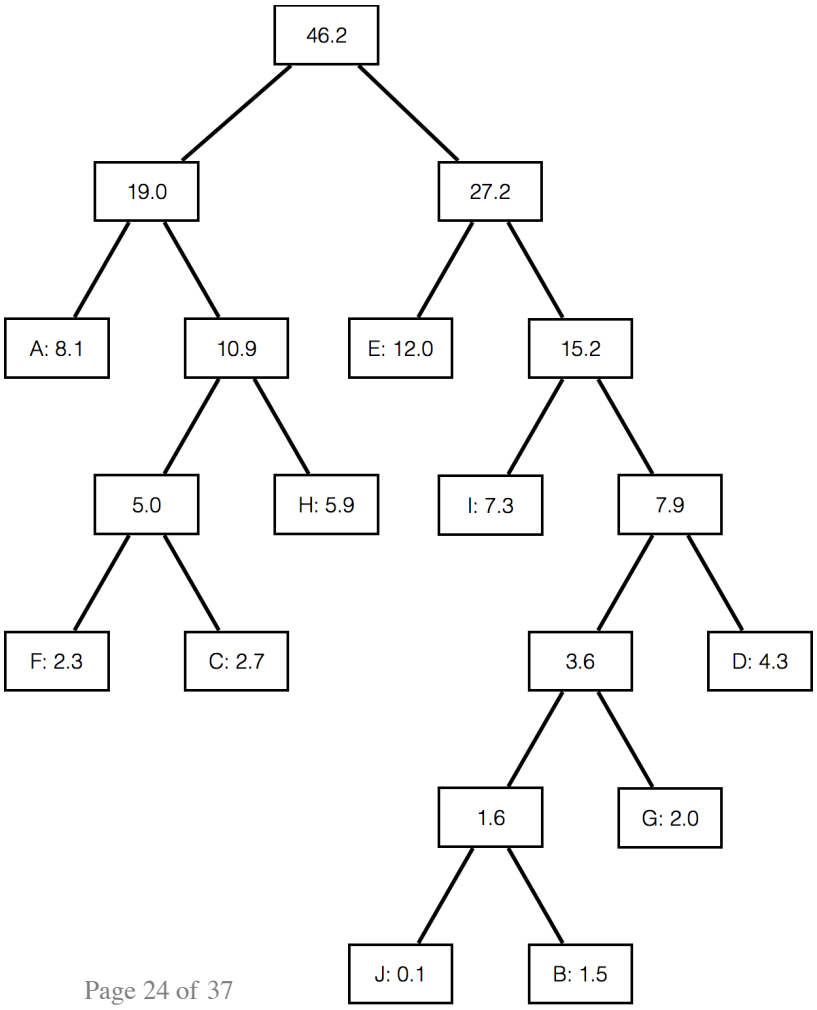


Letter	Frequency
BDEGIJ	27.2
ACFH	19.0





Letter	Frequency
ABCDE FGHIJ	46.2



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.

(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~~.

(a) Do problem 4.7.1. Include a brief explanation of your product array formation.

The purpose of partial product consolidation is to significantly increase the efficiency with which two binary numbers can be multiplied. This process effectively reduces the number of terms needed by the multiplication from n^2 terms down to (n+1)n/2 terms. The skewed triangular bit product for 8-bit squaring is shown below:

									q_7	q_6	q_5	q_4	q_3	q_2	q_1	q_0	
								q_7q_0	q_6q_0	q_5q_0	q_4q_0	q_3q_0	q_2q_0	q_1q_0	0	q_0	q_0
							q_7q_1	q_6q_1	q_5q_1	q_4q_1	q_3q_1	q_2q_1	0	q_1			q_1
						q_7q_2	q_6q_2	q_5q_2	q_4q_2	q_3q_2	0	q_2					q_2
					q_7q_3	q_6q_3	q_5q_3	q_4q_3	0	q_3							q_3
				q_7q_4	q_6q_4	q_5q_4	0	q_4									q_4
			q_7q_5	q_6q_5	0	q_5											q_5
		q_7q_6	0	q_6													q_6
	0	q_7															q_7
	s_{15}	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0	

In forming the product array, a simple way of describing the process is "vertical compression". Moving across the array, rows are compressed and like terms are combined. When terms are combined, however, the carry bit is handled by indicating that the smallest, next order bit is inverted (notated by a bar under the bit due to formatting restrictions) and the entire term *without* the inversion is carried to the next largest bit. For example, the compression of q_1q_0 and q_1 results in the term $q_1\underline{q_0}$ with a carry of q_1q_0 applied to the next high order bit (q_2q_0). This process continues throughout the array until the final reorganized triangular array is formed.

The reorganized triangular array for the 8-bit squaring is shown below, which results in an array of *n*/2 rows:

									q_7	q_6	q_5	q_4	q_3	q_2	q_1	q_0	
q_4	q_7q_6	$q_7\underline{q_6}$	q_7q_5	q_7q_4	q_7q_3	q_7q_2	q_7q_1	q_7q_0	q_6q_0	q_5q_0	q_4q_0	q_3q_0	q_2q_0	$q_1\underline{q_0}$	0	q_0	q_0
q_5			q_6q_5	$q_6\underline{q_5}$	q_6q_4	q_6q_3	q_6q_2	q_6q_1	q_5q_1	q_4q_1	q_3q_1	$q_2\underline{q_1}$	q_1q_0				q_1
q_6					q_5q_4	$q_5\underline{q_4}$	q_5q_3	q_5q_2	q_4q_2	$q_3\underline{q_2}$	q_2q_1						q_2
q_7							q_4q_3	$q_4\underline{q_3}$	q_3q_2								q_3
	s_{15}	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0	

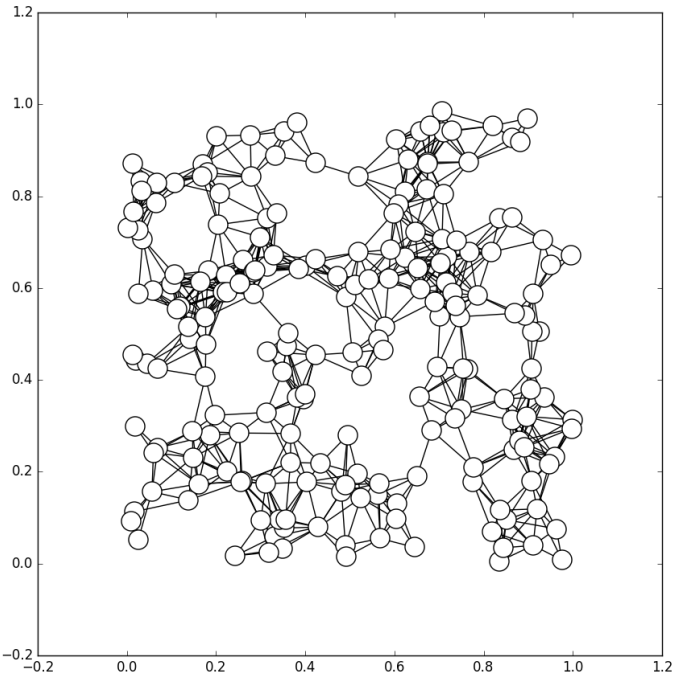
6. [***Maximal Independent Set in an RGG***]
A maximal independent set in a graph is a subset of vertices that are pairwise non adjacent and where no other vertex can be added preserving this property.

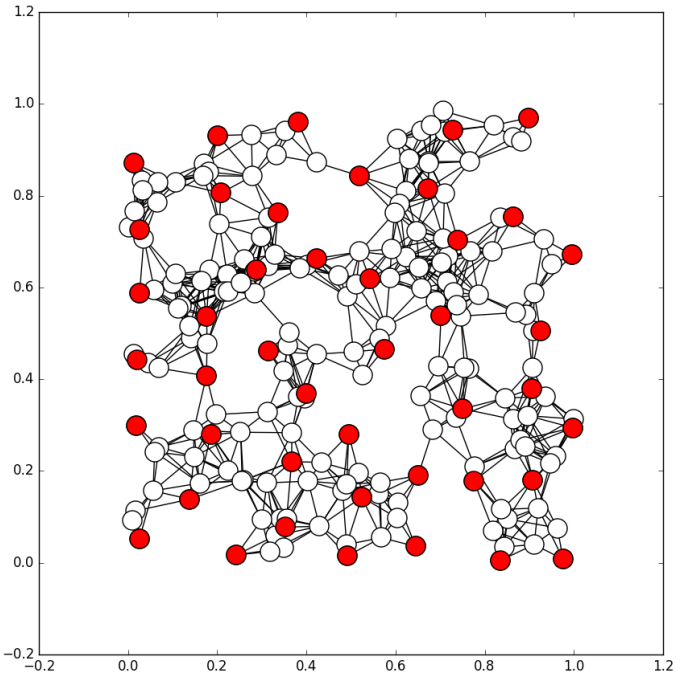
(a) Implement a greedy maximal independent set selection algorithm for a random geometric graph (RGG) where you recursively select a vertex of minimum degree for the set and delete it and its adjacent vertices, and then repeat on the residual graph.

Python was used to implement this algorithm. In addition, the library *Networkx* was used to create the random geometric graph and allow for graph manipulation and *Matplotlib* was used to visually draw the graph. The algorithm first creates a random geometric graph (RGG) using a predetermined radius and number of nodes, which were set to 0.125 and 200 respectively, then calls a recursive function to handle the creation of the maximal independent set.

The recursive function accepts the *Networkx* graph and an empty list which becomes the maximal independent set of nodes. As the recursive function executes, the nodes are sorted by their degree and the algorithm greedily picks the node with the smallest degree. All list is created containing all adjacent nodes, neighbors, and each are removed from the original graph. Finally, the original node is removed from the graph and added to the maximal independent set. At this point, the recursive function calls itself and passes it the residual graph and the maximal independent set. Finally, when all nodes have been removed from the graph, the maximal independent set is returned to the original calling function.

From a visual perspective, the program then uses data for the original graph and the list of nodes in the maximal independent set and graphs the two. The following diagrams illustrate the original graph with white nodes and the maximal independent set, which is over laid on top of the original graph, is indicated with red nodes.





6a – Algorithm Code

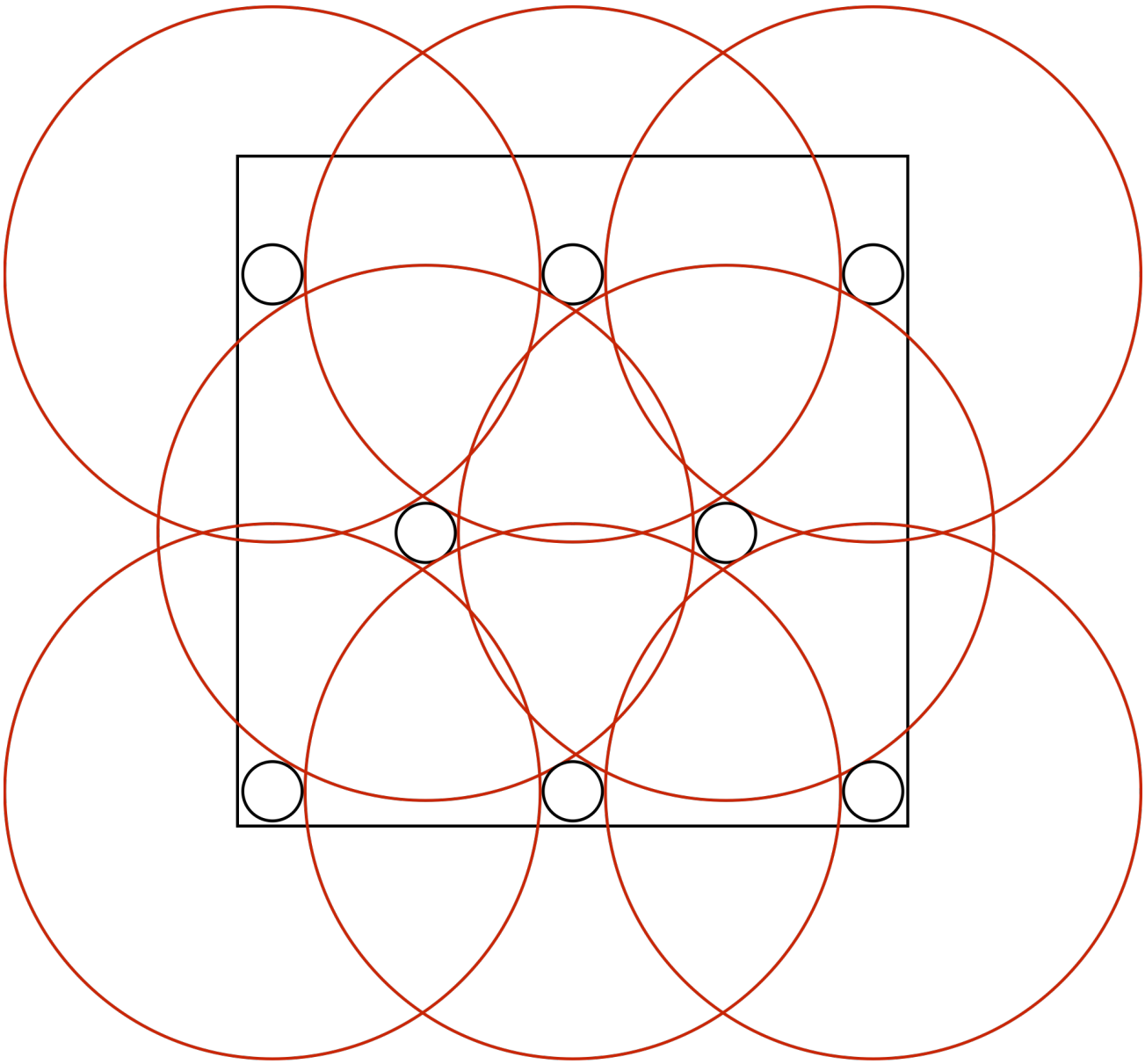
```
import networkx as nx
import matplotlib.pyplot as plt
import operator
import copy


def main():
	# Create graph
	num_nodes = 200
	G = nx.random_geometric_graph(num_nodes, 0.125)
	original_graph = copy.deepcopy(G)
	print "Graph Created”

	# Call recursive function with node data
	print "Starting Recursion"
	mis = create_mis(G, [] )

	# Creating figure
	plt.figure(figsize=(10,10))
	pos = nx.get_node_attributes(original_graph,'pos')
	# Draw original graph
	nx.draw_networkx_edges(original_graph, pos)
	nx.draw_networkx_nodes(original_graph, pos, node_color='w')
	plt.savefig('original_graph.png')

	# After drawing original graph, remove all nodes but those included in the mis
	for node in nx.nodes(original_graph):
		if node not in mis:
			original_graph.remove_node(node)

	# Graph the maximal independent set on top of the original graph
	pos=nx.get_node_attributes(original_graph,'pos')
	nx.draw_networkx_edges(original_graph, pos)
	nx.draw_networkx_nodes(original_graph, pos, node_color='r')
	pos = nx.get_node_attributes(original_graph,'pos')
	plt.savefig('mis.png')

	# Show Graph
	plt.show()


def create_mis(graph, independent_set):
	# Create data dictionary to hold node labels and their degree
	nodes = {}
	for n in nx.nodes(graph):
		nodes[n] = nx.degree(graph,n)
		print n

	if len(nodes) == 0:
		print "Base case reached; returning to main"
		return independent_set
	else:
		# Sort data structure by degree
		nodes = sorted(nodes.items(), key=operator.itemgetter(1))
```

```
            # Select vertex of minimum degree (get the label of the pair at index 0)
            min_vertex = nodes[0][0]

            # Add the vertex to the independent set
            independent_set.append(min_vertex)

            # Select minimum degree vertex's neighbors
            neighbors = nx.neighbors(graph, min_vertex)

            # Delete its neighbors from the graph
            for n in neighbors:
                  graph.remove_node(n)

            # Delete the vertex from the graph
            graph.remove_node(min_vertex)
            print "Removed min vertex"

            return create_mis(graph, independent_set)


main()
```

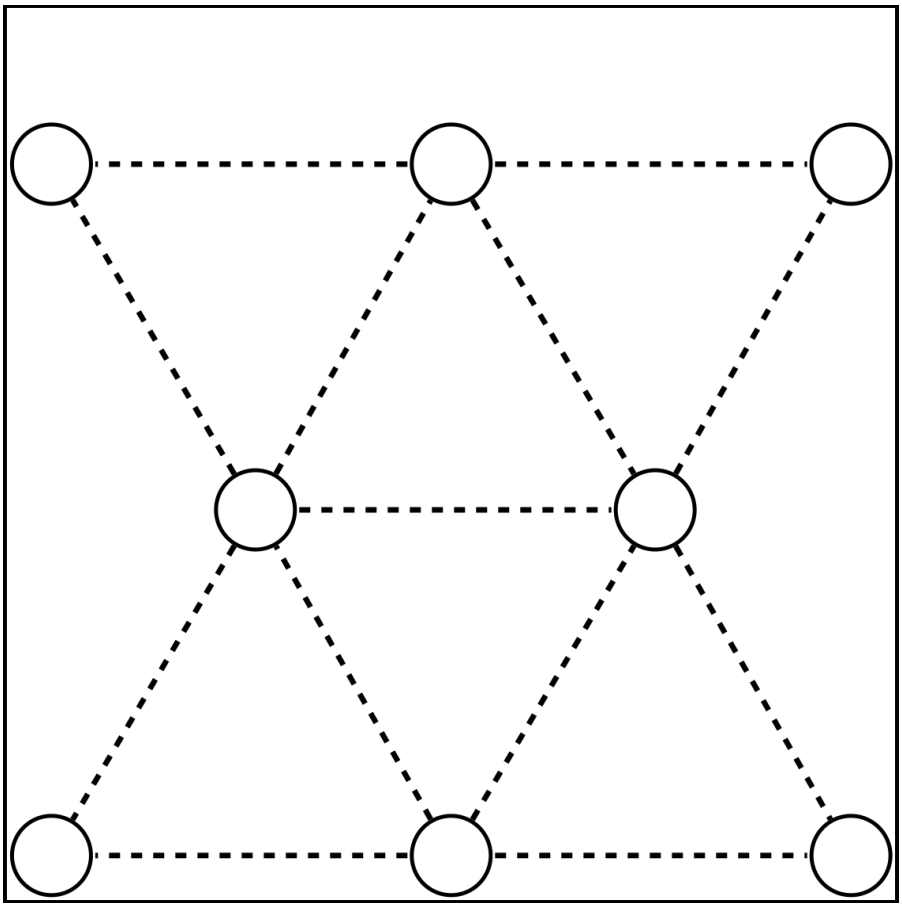
(b) Determine an approximate upper bound on the maximal independent set size by assuming the largest such set forms a triangular lattice. Run your program 10 times and find the average set size you obtain for RGG's on 2,000 vertices with average degree about 40, and also for 8,000 vertices with average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

In a maximal independent set, no two vertices share an edge. When this is represented in graphical form with each vertex's radius shown as a ring around it, the rings of each node in a maximal independent set cannot encompass other nodes from the same set. As a result, when these nodes are packed as tightly as possible, each vertex's radius is essentially as close as possible to every neighboring node in the set without actually including it within its radius.

In order to approximate the upper bound of the maximal independent set (MIS) for a graph, it must be assumed that nodes exist in all corners of the graph. As a result, the coverage of the graph by the nodes in the MIS must be complete. The following diagram indicates the structure of an example tightly packed MIS with each node's radius shown as a red circle, which begins by using the node at the origin (0,0):



By making visual connections between these nodes, it is apparent that the underlying structure resembles that of a triangular lattice. This is shown in the diagram below:



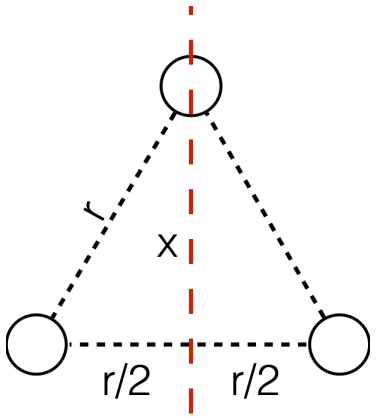
To begin approximating the upper bound on the MIS, the width of the graph will be addressed. Assuming the algorithm begins with the vertex with the smallest degree and that this vertex exists at the origin (0,0), the number of vertices needed to cover the width of the graph can be calculated by the following:

$$\text{Number of Horizontal Vertices} = \text{CEILING}[\ (width \div r)\ ]$$

After dividing the width by the radius of the nodes, the result will likely have a remainder. Because this maximal independent set must account for nodes in the bottom right corner of the graph (1,0), we must use the ceiling function to ensure the width of the graph is entirely covered.

After determining the number of vertices needed to cover the width of a graph, the number of vertices needed to cover the height of the graph must be approximated. In order to do this, the "levels" of the triangular lattice must be examined. In order to find the height between rows, the height of any given triangle in the lattice must be calculated. This can be done using the pythagorean theorem by vertically splitting a given triangle from the lattice.

In order to find the height of the triangle, represented as *x* in the diagram to the right, a triangle from the lattice must be split vertically, splitting the bottom of the triangle into two equal pieces of length $r \div 2$. Given sides *r* and $r \div 2$, the pythagorean theorem can be used to find the value of x:



$(r \div 2)^2 + x^2 = r^2$

Which can be rearranged as:

$x^2 = r^2 - (r \div 2)^2$
$x = \text{sqrt}(r^2 - (r \div 2)^2)$

After approximating the height of each triangle in the lattice, the calculation of the number of rows of nodes needed to vertically cover the graph can be represented as the following:

$$\text{Total Number of Rows of Vertices} = \text{CEILING}[\ \textit{height} \div \text{sqrt}(r^2 - (r \div 2)^2)\ ]$$

Again, ceiling is used to ensure the graph is entirely covered. Without using the ceiling function to round the number of rows up, portions of the graph, which may contain nodes, will not be entirely covered.

The final component of the approximation must take the alternating row structure of the lattice into consideration. Because every other row in the graph is offset by $r \div 2$, this must be taken into consideration when calculating the number of vertices that exist on an alternate row. This can be represented by the following:

$$\text{Number of Horizontal Vertices in Alternate Row} = \text{CEILING}[\ (\textit{width} - r \div 2)\ \div r\ ]$$

Furthermore, assuming the algorithm begins at the origin and that every row *after* the first will be an "alternate row", alternate rows will only occur on even numbered rows. The number of these rows can therefore be calculated by taking dividing the total number of rows by 2 and rounding that number down using a floor function. Alternatively, the number of normal rows can be calculated by taking the ceiling of the total number of rows divided by 2.

Number of Normal Rows: **CEILING**[$\textit{height} \div \text{sqrt}(r^2 - (r \div 2)^2) \div 2$]
Number of Alternate Rows: **FLOOR**[$\textit{height} \div \text{sqrt}(r^2 - (r \div 2)^2) \div 2$]

Finally, using the formulas above, the upper bound on the maximal independent set size can be represented by the following:

Upper Bound on Maximal Independent Set Size = # Normal Rows • Number of Horizontal Vertices + # Alternate Rows • Number of Horizontal Vertices in Alternate Row

Upper Bound on Maximal Independent Set Size = $\text{CEILING}[\ \textit{height} \div \text{sqrt}(r^2 - (r \div 2)^2) \div 2\ ] \bullet \text{CEILING}[\ (\textit{width} \div r)\ ]\ +\ \text{FLOOR}[\ \textit{height} \div \text{sqrt}(r^2 - (r \div 2)^2) \div 2\ ] \bullet \text{CEILING}[\ (\textit{width} - r \div 2)\ \div r]$

Next, in order to approximate the appropriate radius size needed in order to get a specific average degree for a specific number of nodes, the following formula must be used:

$$\text{Average Degree} = [(\pi \bullet r^2 \bullet n) \div A]\text{-}1$$

Which can be rearranged to create the following:

$$r = \text{sqrt}[\ A \bullet (\text{Average Degree} + 1) \div (\pi \bullet \text{n})\ ]$$

This formula uses r to represent the radius, n to represent the number of nodes in the graph, and A to represent the total area of the graph. For the given example, which uses a graph of size (1,1), the area is 1. As a result, the following radiuses can be approximated:`

Number of Nodes	Average Degree	Radius
2000	40.0	**0.0808**
8000	100.0	**0.0634**

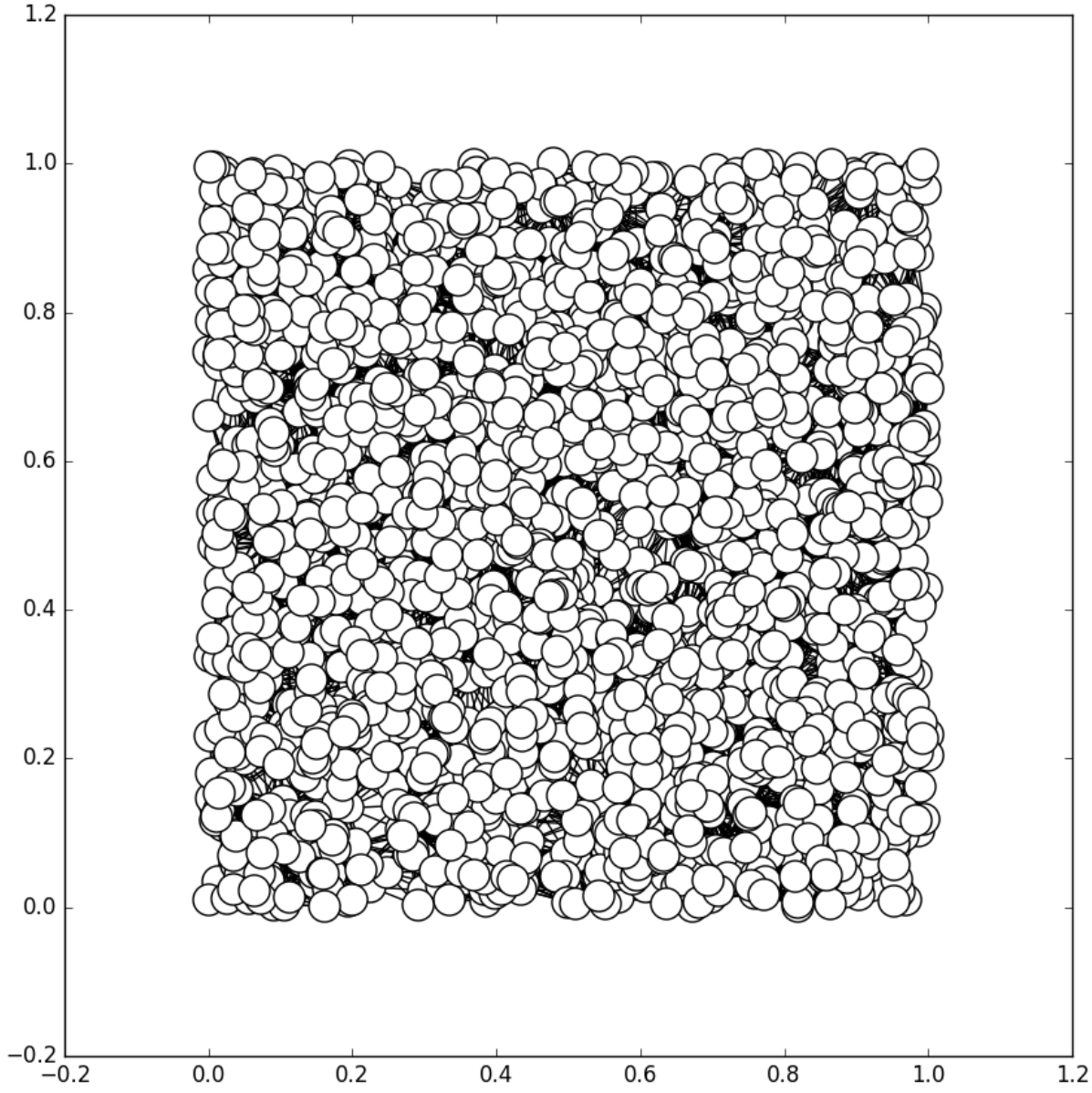
Using these values, the number of nodes per graph can be approximated:

Radius	# Normal Rows	# Alternate Rows	# Nodes Per Normal Row	# Nodes Per Alt. Row	Total Nodes (Approx.)
0.0808	8.0	7.0	12.0	12.0	180.0
0.0634	10.0	9.0	15.0	16.0	294.0

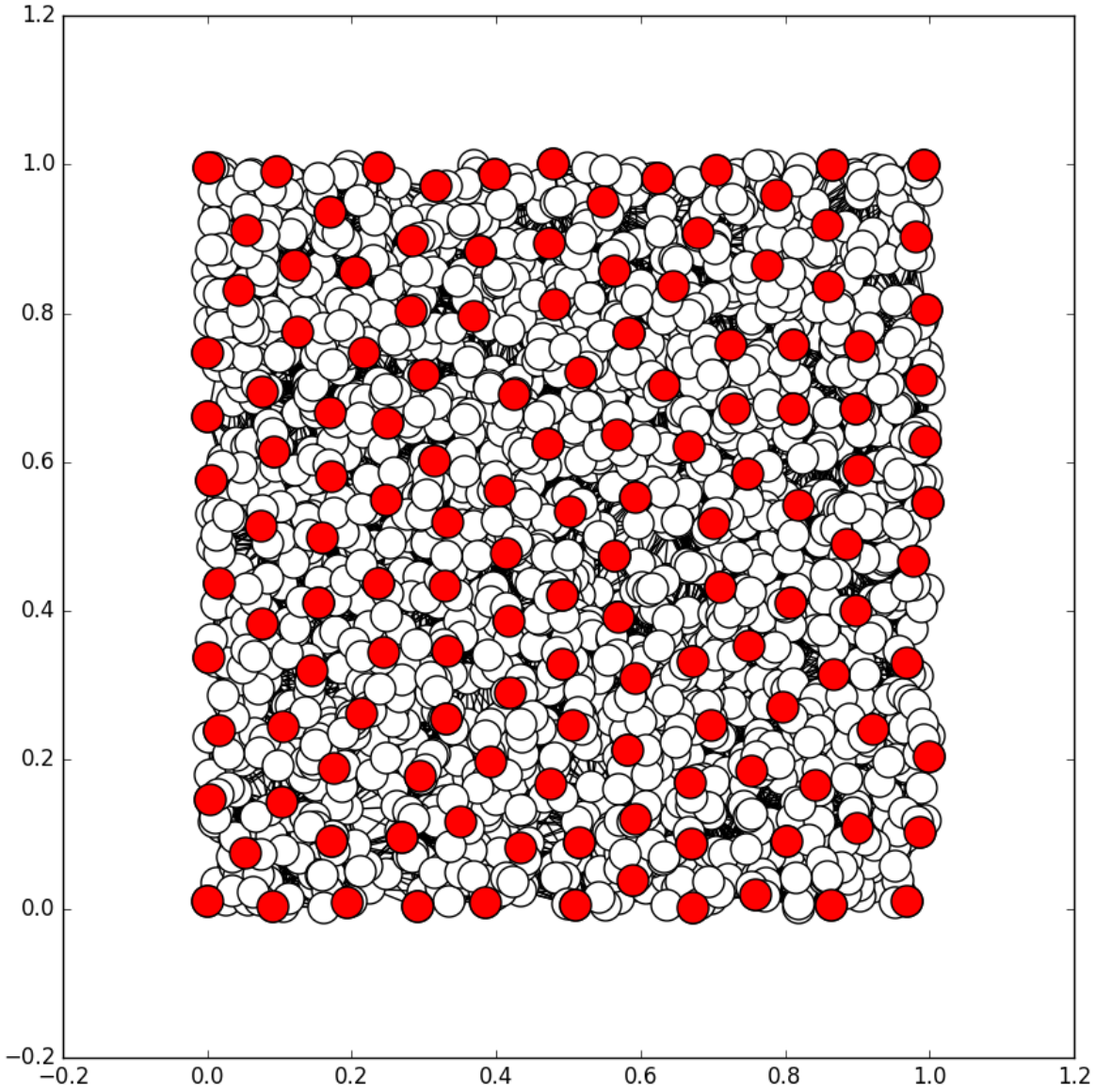
After altering the program discussed in part (a) above, the following data was collected for a 2000 node graph with radius size 0.0808 and an 8000 node graph with radius size 0.0634:

Iteration	2000 Node: Number of Nodes	8000 Node: Number of Nodes
1	134.0	235.0
2	134.0	237.0
3	133.0	236.0
4	132.0	236.0
5	134.0	236.0
6	137.0	236.0
7	137.0	239.0
8	133.0	237.0
9	137.0	232.0
10	138.0	240.0
Average:	134.9	236.4
Expected:	180.0	294.0
% Variance:	25.1%	19.6%

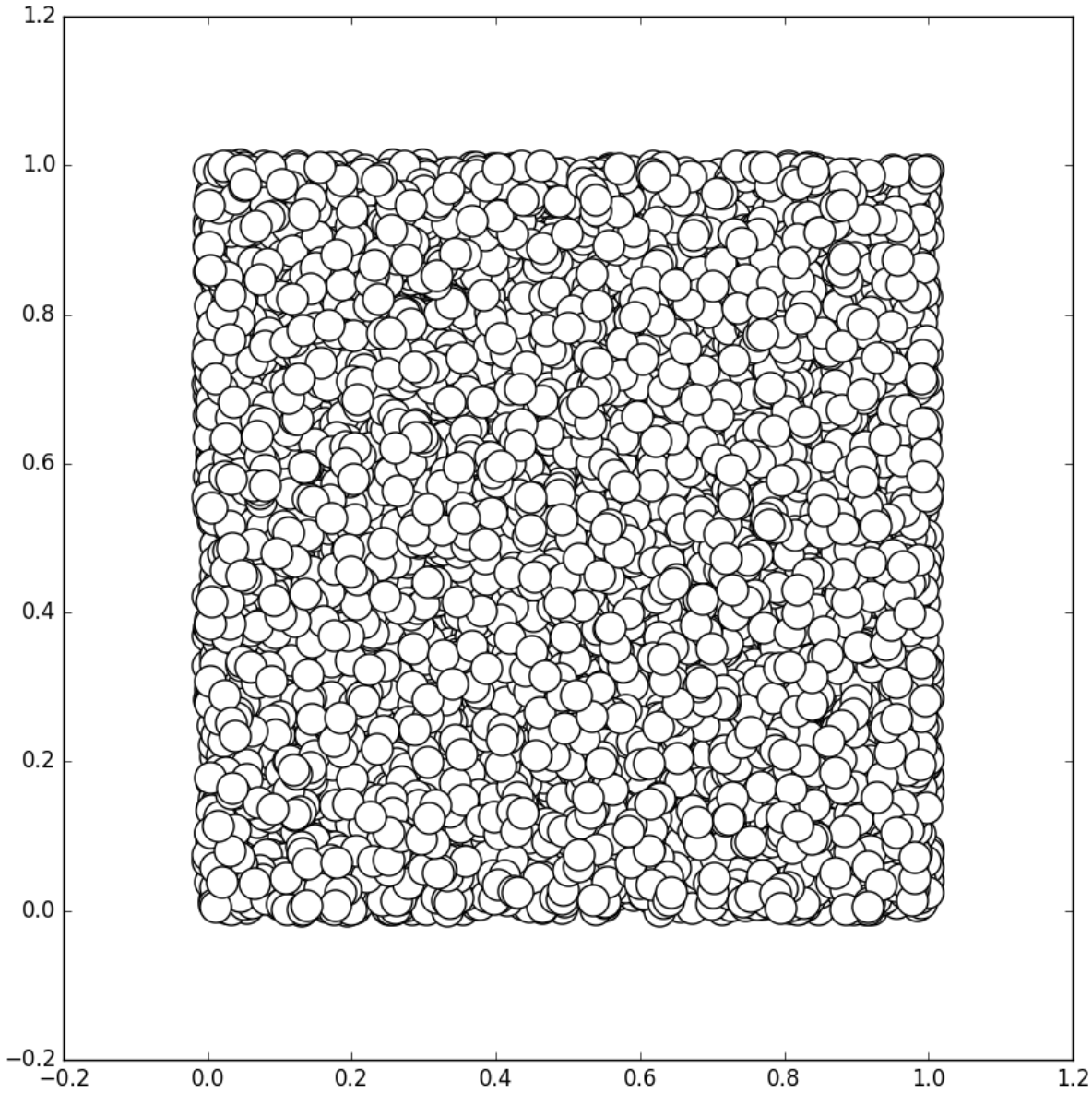
As indicated by the layout of the nodes in the maximal independent set within the graphs on the following pages, the maximal independent set is not tightly packed. As a result, there is a lesser degree of overlap between nodes. because there is less overlap and because nodes are only needed to cover nodes that actually exist, fewer needed nodes are needed in the maximal independent set. Therefore, a deviation of 19.6% - 25.1% is very reasonable for an upper bound within a maximal independent set.



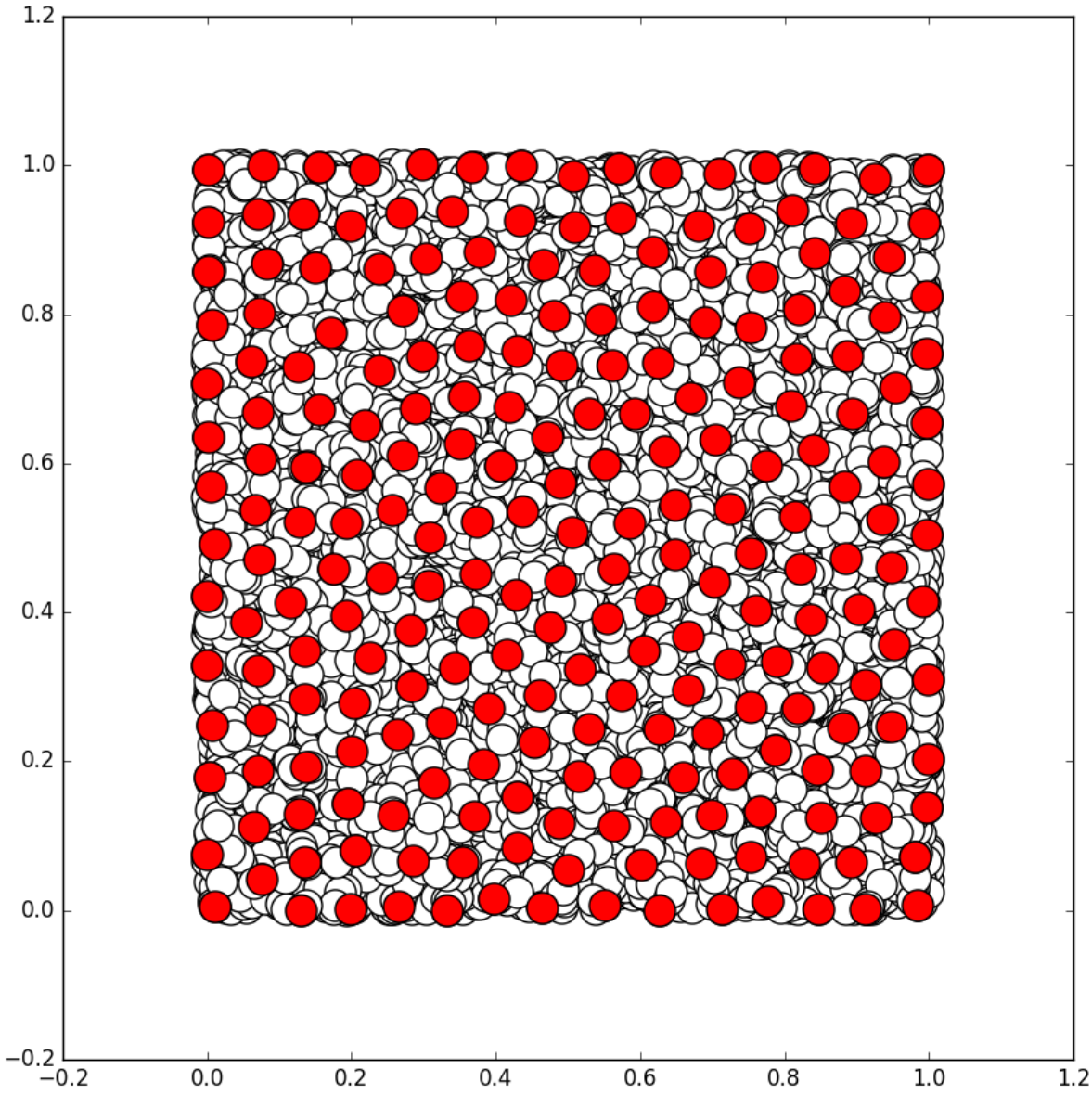
n = 2000
All Nodes



n = 2000, r = 0.0808
Nodes in the Maximal Independent Set shown in red



n = 8000
All Nodes



n = 8000, r = 0.0634
Nodes in the Maximal Independent Set shown in red