Students: Maria Arsky
Niv bar

Data structure:

Our data structure is an array of linked lists, you can access to any vertex by its index (O(1)), and you can check all it's neighbors by scanning the linked list (O(m), m = number of neighbors). This efficiency will be very useful in the BFS and is_isolated functions.

Functions:

build_random_graph

Building a random graph by probability.

input – p (probability), data type: double
output - graph, data type: array<vector<int>, V>

Using array of vectors (linked lists) data structure.
array size is a const number (V = 1000) – the indexes are representing all vertices in the graph. Vectors' sizes are unknown, they depend on the probability – representing the number of neighbors for each vertex.

Explanation:
Assume there is a matrix V*V (1000*1000) which represent any possible edge between two vertices. Hence, we scan only the upper triangular (for efficiency, checking only the entries above the main diagonal, for the reason we don't want to check each edge twice).
We are calculating the relative probability by multiply the probability we have by 10,000 (to deal with small probabilities). In each scan we randomize a number between 1 to 10,000, and if this random number is smaller than the relative probability, so the current edge is existing in the graph.

Complexity - We denote V = number of vertices. In order to scan the array, V operations are required. In order to select edges for each vertex i, V - i - 1 operations are required. Total we get O(V^2).

print_graph

print the given graph.

input – graph, data type: <array<vector<int>, V>

Explanation:

Print the given graph, when printing the neighbors of each vertex. This function is for our comfort.

Complexity – We scan the array of vertices (O(V)), and scan all neighbors (O(m), m = number of neighbors of each vertex). In total we get O(V^2).

BFS

Calculating the distance array from the given vertex to all other vertices.

input – graph, data type: <array<vector<int>, V>, vertex, data type: int
output – dist (distance array), data type: int (array)

Explanation:

Colors' array initializes to 0. The numbers 0,1,2 representing colors and there meaning in BFS scanning: white = 0 – unvisited vertex, grey = 1 – in queue, black = 2 – finish.
Distances' array initializes to -1 (= infinity). At the end, if it is an infinity (-1) at the array → the graph is not connected, else if all valued are >= 0 → the graph is connected.

Complexity - O(V + E).

Connectivity

Checking if graph is connected or not.

input – graph, data type: <array<vector<int>, V>; diam (by reference), data type: int
output – 1 if connected, else 0.

Explanation:

Calculate the distance array by BFS function.
Base on BFS statement: If num of infinity (-1) > 0 → the graph is not connected.
Scanning the distance array and if there is value of -1 → return 0. If there is not case like that → return 1.
*** At the same time the max distance from vertex 0 is updated into diam (for the diameter function).

Complexity – One running of BFS (O(V + E)) and scanning the dist array (O(V)), than we get O(V + E).

diameter

Checking what is the diameter of the given graph.

input – graph, data type: <array<vector<int>, V>; diam (of the first vertex [vertex_0]), data type: int
output – diam, data type: int

Explanation:

In order to find diameter of a graph, need to apply BFS on all vertices which belong to the graph. We get the max distance of the first vertex from connectivity function (if it returns -1, then the graph's diameter is infinity), and then we run BFS V – 1 times in order to find the max distance of all distances:
diam(G) = max{dist(e, v): for each u and v from V }.

Complexity – We run BFS V – 1 times, V * O(V + E) operations are required. Then we search for max distance (diameter) by checking the upper triangular (for the reason we don't want to check each distance twice), V operations are required.
Total we get O(V * (V + E)).

is_isolated

Checking if there is at least one vertex without neighbors.

input – graph, data type: <array<vector<int>, V>
output – 1 if there is vertex without neighbors, else 0.

Explanation:

At every iteration we check if each lined list (of neighbors) which belongs to the current vertex in the array is empty or not. If empty → return 1 (means there is a vertex without neighbors), else return 0.

Complexity – We scan the array of vertices, V operations are required, in total we get O(V).

create_csv_file

Creating csv file.

input – file name, data type: string; probabilities, data type: double array; ratio, data type: double array.

Explanation:

Creating a csv file in which first line represents probabilities, and the second line represents the ratio.

Complexity – We write 10 probabilities and 10 ratios into the csv file, than in total we get O(1).

Attributes_counters

Counts how many graphs (out of 500) are: connected or have at least one isolated vertex or have diameter = 2.

input – p, data type: double; attribute, data type: int
output – attr_counter, data type: int
Explanation:

There are 500 iterations, in each of them new random graph created (with the probability the function got). Then, respectively to the attribute the function got, the counter increases if the graph supports the attribute.
We count:

- Connectivity – if the graph is connected.
- Diameter – if the graph's diameter is 2.
- Isolated – if the graph has at least one isolated vertex.

Complexity – 500 iterations happens (generally O(1)), graph build (O(V^2)), and then we check if the graph have the attribute (each attribute has itself complexity). In total we get O(V^3 * (V + E)) operations (in worst case with the diameter).

find_ratio

Updating the ratio array for each attribute.

input – probability, data type: double array; ratio, data type: double array; attribute, data type: int

Explanation:

For each probability (10 iteration), we update the ratio array (respectively to the probability index), using the attributes_counters function.
The formula to calculate the ratio for each probability is: attr_counter / 500 (attr_counter = number of times the graph has a specific attribute out of 500 graphs).

Complexity – We have 10 probabilities (O(1)). In every iteration we call the attributes_counters function (O(V^3 * (V + E))), and then we calculate the ratio (O(1)). In total we get O(V^3 * (V + E)) operations.

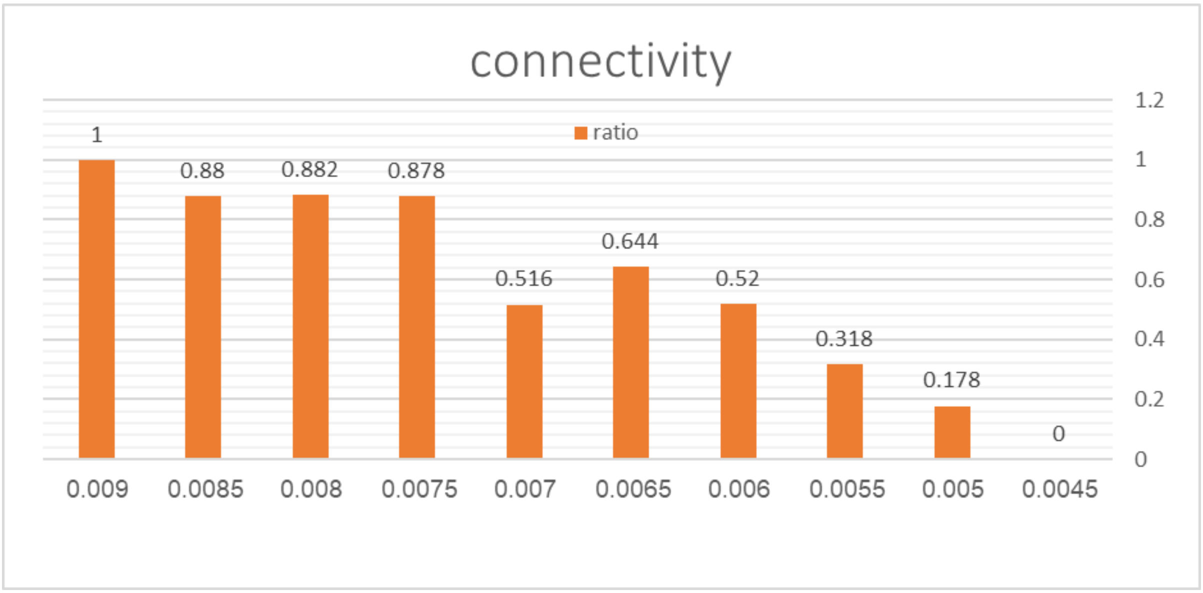
Details:

Number of vertices = 1000
Threshhold – 1,3: **ln(v)/v → ln(1000)/1000** = 0.0069
Threshhold – 2: **squrt(2 * ln(v)/v) →** squrt(2 * ln(1000)/1000) = 0.1175

From csv files:

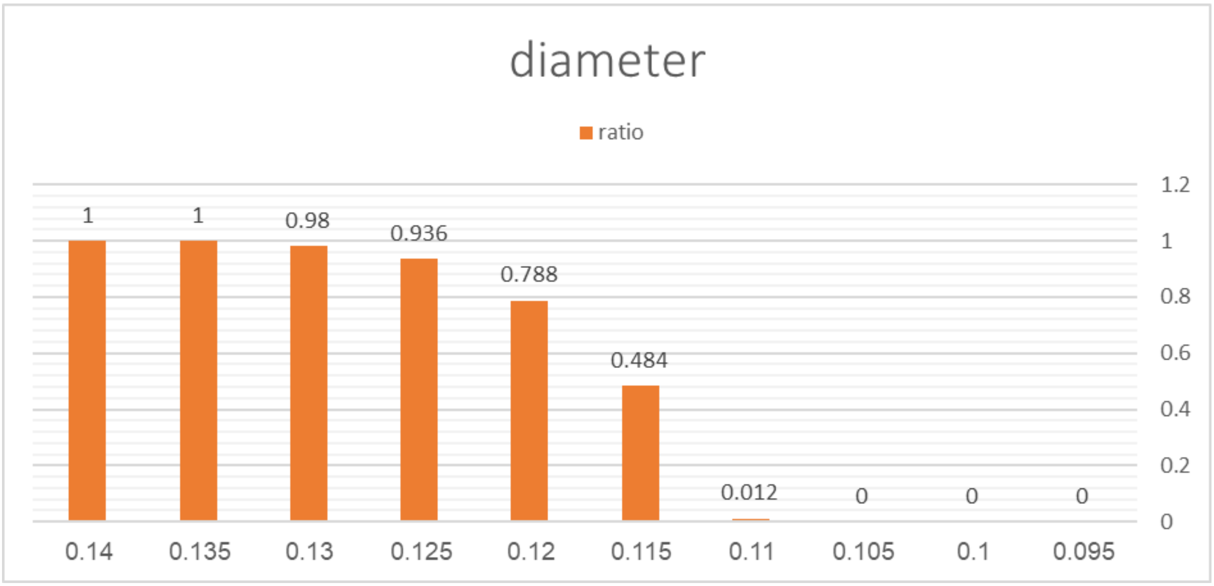
Attribute 1 – connectivity:

probability	0.0045	0.005	0.0055	0.006	0.0065	0.007	0.0075	0.008	0.0085	0.009
ratio	0	0.178	0.318	0.52	0.644	0.516	0.878	0.882	0.88	1



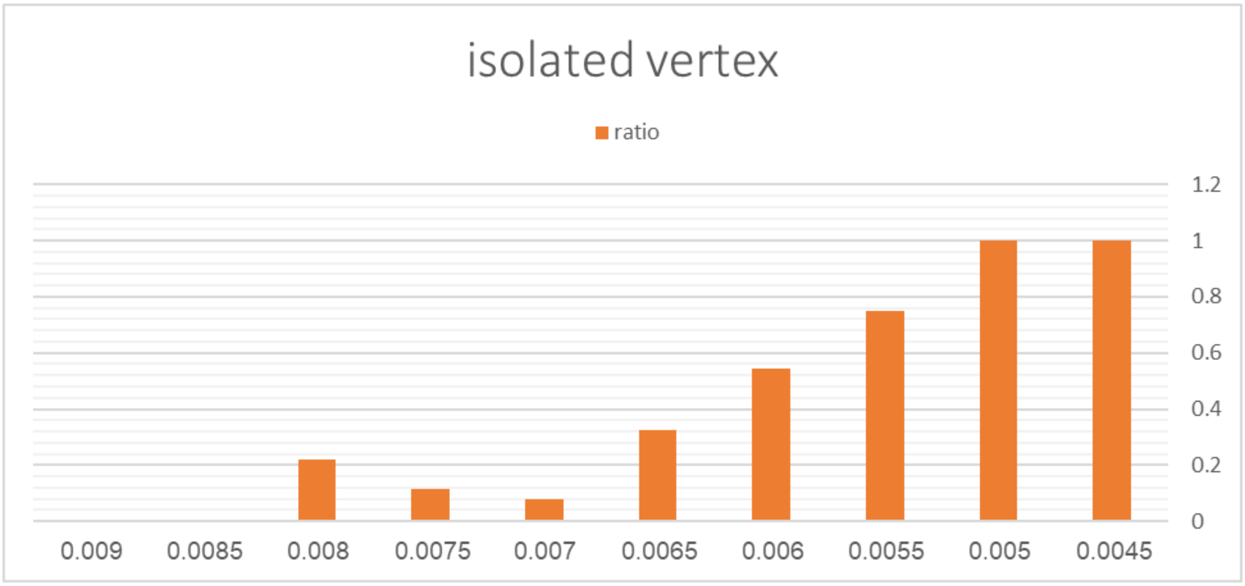
Attribute 2 – diameter:

probability	0.095	0.1	0.105	0.11	0.115	0.12	0.125	0.13	0.135	0.14
ratio	0	0	0	0.012	0.484	0.788	0.936	0.98	1	1



Attribute 3 – isolated vertex:

probability	0.0045	0.005	0.0055	0.006	0.0065	0.007	0.0075	0.008	0.0085	0.009
ratio	1	1	0.748	0.542	0.326	0.08	0.114	0.222	0	0


isolated vertex
■ ratio