INFORMATICS INSTITUTE OF TECHNOLOGY

UNIVERSITY OF WESTMINSTER

Informatics Institute of Technology

Department of Computing

BEng(Hons) Software Engineering

5SENG003C.2- Algorithm: Theory, Design and Implementation

MODULE LEADER: Mr. Sivaraman Ragu

Name : Akshayan Mohandas

UoW No. : w1867142

IIT ID : 20210420

a.

Data Structure

My choice of data structure is an adjacency list. An adjacency list is a group of lists, each of which represents a vertex's neighbors in a graph. While the inner ArrayLists represent the edges, the outer ArrayList represents the vertices. The method connectNodes adds a connection between two vertices by adding the destination vertex to the list of neighbors for the source vertex. The graph's connections between its vertices are printed out by the method called displayGraph. The method sinkElimination eliminates any sink vertices (vertices without outgoing edges) from the graph by performing sink elimination on it, which includes eliminating any such vertices from the graph and repeatedly calling the method until there are no more sinks.

Algorithm

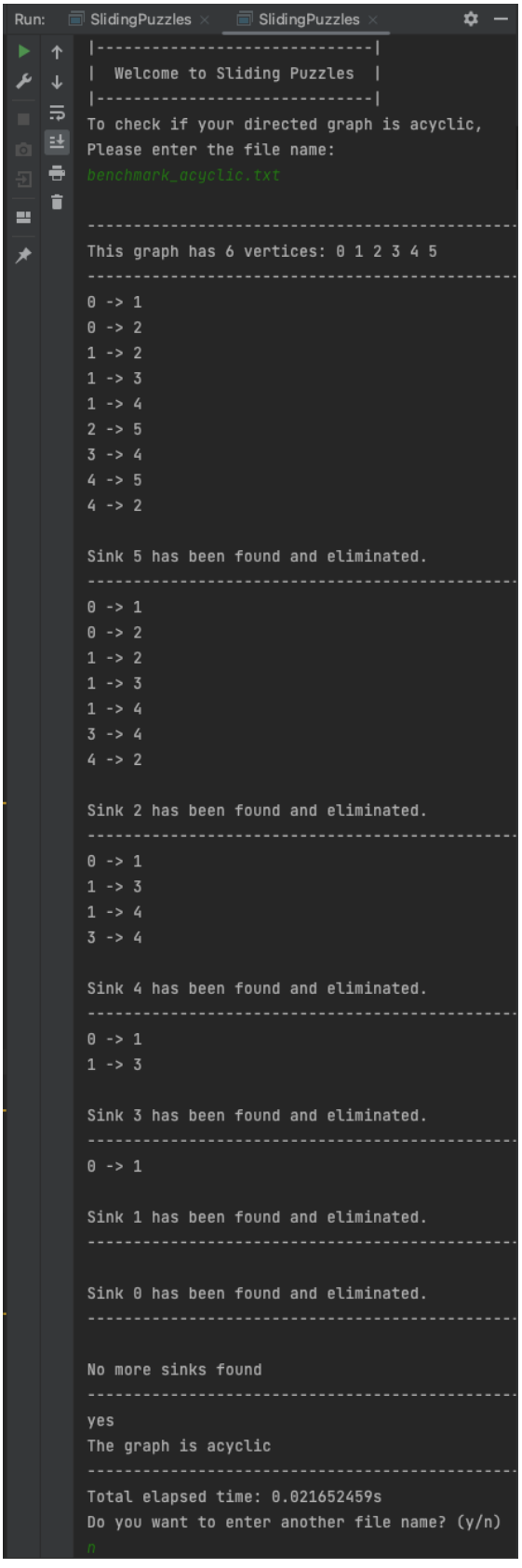
Sink Elimination Algorithm

The sink elimination algorithm locates and eliminates sink vertices in a directed graph where a sink vertex has no outgoing edges. Once a sink vertex is detected, it and its incoming edges are removed. This process is repeated until no more sink vertices are discovered. A sink vertex's incoming edges are likewise removed when it is removed, which separates neighboring vertices. By analyzing if any sink vertices remain after the process has run, the technique can determine whether a directed graph is acyclic or not. The graph is acyclic if all of the vertices have been eliminated, and cyclic otherwise.

Depth-First Search (DFS)

The cycle is found by the algorithm by conducting a depth-first search from each vertex that is currently present. A graph traversal algorithm called Depth-First Search (DFS) begins at a node and travels as far as it can before backtracking. By iteratively analyzing neighboring vertices and including them in the path, DFS is carried out starting from each vertex. By determining if the current vertex has previously been visited in the path, the program finds cycles. The algorithm returns the path as the cycle if a cycle is discovered. If not, it chooses a random unexplored neighboring vertex and keeps searching until a cycle is discovered. In general, the technique traverses the graph using DFS and recursion to find cycles.

b. Acyclic Graph

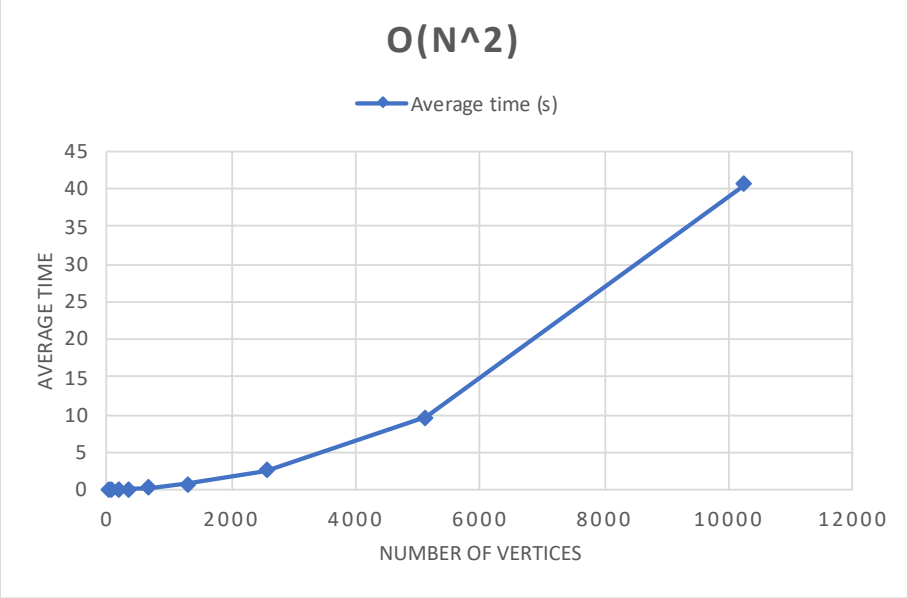


Cyclic Graph

c.

The doubling hypothesis can be used to empirically evaluate the algorithm's effectiveness. On input arrays that get bigger, we can execute the algorithm and time how long it takes to finish. We would anticipate a quadratic rise in the running time when the input size doubled if the algorithm had an O(n^2) time complexity.

Number of vertices	1st time (s)	2nd time (s)	3rd time (s)	4th time (s)	5th time (s)	Average time (s)
40	0.03398671	0.03631408	0.02857617	0.0319275	0.03208063	0.032577017
80	0.04610704	0.03521596	0.04157442	0.04366413	0.03771333	0.040854975
160	0.0966285	0.10381417	0.09571038	0.09792613	0.10012075	0.098839983
320	0.10375383	0.09729542	0.10107571	0.10346013	0.10520675	0.102158367
640	0.21977825	0.22626571	0.270804	0.26766446	0.25920233	0.24874295
1280	0.84189642	0.81575225	0.78269242	0.80982271	0.81397217	0.812827192
2560	2.46349683	2.54310946	2.59379871	2.57451546	2.47401842	2.529787775
5120	9.34310975	9.56844308	9.917116	9.54751558	9.66271158	9.6077792
10240	40.1588236	39.279012	39.2400912	41.347039	42.7927907	40.56355127



The algorithm is quadratic. The input size doubles between inputs, and the runtimes grow by a factor of approximately 4. Since $4 = 2^2$ the complexity class is $O(n^2)$.

We can also examine the effectiveness of the provided algorithm theoretically. The algorithm's outer loop executes n times, where n is the size of the input array. For each iteration of the outer loop, the inner loop executes n-i times.

This is an arithmetic series, and its total can be calculated as (n-1) * n / 2. Therefore, the total number of iterations of the inner loop is n(n-1)/2.
The time complexity of the algorithm is then O(n^2). This is so because the constant factor is not taken into account in the Big-O notation, and n^2 is the dominant factor in the equation for the total number of inner loop iterations.

The algorithm's overall time complexity is O(n^2). This implies that the algorithm's execution time will grow quadratically as the size of the input rises. As a result, the technique might not work well with large input sizes.