Algorithms

Homework 2: 2023-12753 EunSu Yeo

1. Environment & Setting

Environment	Setting
OS	Window 11
Code Editor	Visual Studio Code (Linux VM)
Language	Python
Version	3.11.5

2. Program description

   To implement the SCC Algorithm given in the Lecture Note, I made the 3 functions to make adjacency matrix, list and array. In the below, I'll explain more about these.

   a. build_adjacency_matrix

      To implement the adjacency matrix function in Lecture Note, I made 2 matrices. Adjacency matrix is matrix to build the matrix that represents the relation of vertex and edge. Transpose matrix is matrix to build the matrix that is used for checking the SCC. With these two matrices, we check SCC by checking some vertex I goes to vertex J and vertex J goes to vertex I.

```python
def build_adjacency_matrix(vertex_count, edge_list):
    # Make the adjacency matrix
    # Initialize matix with False values
    adjacency_matrix = []
    for _ in range(vertex_count):
        adjacency_matrix.append([False] * vertex_count)
    # Fill the adjacency matrix with edges
    for source, target in edge_list:
        adjacency_matrix[source][target] = True

    # Make transposed adjacency matrix
    # Initialize matix with False values
    transpose_matrix = []
    for _ in range(vertex_count):
        transpose_matrix.append([False] * vertex_count)
    # Fill the transposed adjacency matrix with edges
    for source, target in edge_list:
        transpose_matrix[target][source] = True

    return adjacency_matrix, transpose_matrix
```

b. build_adjacency_list

To implement the adjacency list function in Lecture Note, I made 2 list structures. As the python does not support pointer, I made the structure that works as linked list, which is conceptually same as the pointer in C. In this ListNode structure, when new node is created, the reference is contained in the adjacency_heads[source] so it works equal to pointer in C.

```
class ListNode:
    def __init__(self, val, nxt=None):
        self.val = val
        self.next = nxt

def build_adjacency_list(vertex_count, edge_list):
    # Initialize head pointers for each vertex
    adjacency_heads = [None] * vertex_count
    transpose_heads = [None] * vertex_count

    # Insert each edge at the head for O(1) insertion
    for source, target in edge_list:
        # forward edge
        node = ListNode(target, adjacency_heads[source])
        adjacency_heads[source] = node
        # transpose edge
        tnode = ListNode(source, transpose_heads[target])
        transpose_heads[target] = tnode

    return adjacency_heads, transpose_heads
```

As the code above, I implemented the adjacency list by arrays that has linked list, which is conceptually same as the adjacency list shown is ppt 11.

c. build_adjacency_array

To implement the adjacency array function in Lecture Note, I made 2 tuple structures. Since Python doesn’t support pointers, I instead built the same structure by using two lists and bundling them into a tuple.

In ppt 11, the adjacency array contains information about the end position of vertices adjacent to each vertex in an array, which works as index. So, I made the array prefix_degree that contains information about the end position of vertices.

Like the array shown in ppt 11, prefix_degree works to find the start and end index of the vertex i in the adjacency array. As you can see in the code below, the prefix_degree is made by adding the number of adjacent edges for each vertex. But to make the DFS simple, I added the 0 (index for starting) in front of the prefix_degree which makes the array length to |V|+1.

```python
def build_adjacency_array(vertex_count, edge_list):
    # Get the number of edges
    total_edges = len(edge_list)

    # Compute degree for each vertex
    degree_count = [0] * vertex_count
    for source, _ in edge_list:
        degree_count[source] += 1

    # Compute prefix sums to determine end positions of vertices
    prefix_degree = [0] * (vertex_count + 1)
    for index in range(1, vertex_count + 1):
        prefix_degree[index] = prefix_degree[index - 1] + degree_count[index - 1]

    # Get the start position in array of each vertices
    index = prefix_degree[:-1].copy()

    # Fill adjacency arrays
    adjacency_array = [0] * total_edges
    for source, target in edge_list:
        adjacency_array[index[source]] = target
        index[source] += 1

    # Compute in-degree for transpose
    in_degree_count = [0] * vertex_count
    for _, target in edge_list:
        in_degree_count[target] += 1

    # Compute prefix sums for transpose
    transpose_prefix = [0] * (vertex_count + 1)
    for index in range(1, vertex_count + 1):
        transpose_prefix[index] = transpose_prefix[index - 1] + in_degree_count[index - 1]

    # Get the start position in array of each vertices for transpose
    index_transpose = transpose_prefix[:-1].copy()

    # Fill transpose arrays
    transpose_adjacency_array = [0] * total_edges
    for source, target in edge_list:
        transpose_adjacency_array[index_transpose[target]] = source
        index_transpose[target] += 1

    return (prefix_degree, adjacency_array), (transpose_prefix, transpose_adjacency_array)
```

As shown above, the adjacency-array function returns a tuple of two lists: the first list contains the end indices for each vertex's adjacency block, and the second list contains all the adjacent vertices for each vertex.

d. kosaraju_strongly_connected_components

While researching algorithms for computing strongly connected components, I discovered that the one in our PPT is Kosaraju's algorithm.

So, I constructed the Kosaraju's algorithm using 2 DFS.

Part 1. Perform DFS on the original directed graph

Part 2. Process DFS to get the SCC vertices in stack order on the reversed graph

Part 3. Return the SCCs

```python
def kosaraju_strongly_connected_components(vertex_count, graph_structure,
                                  transpose_structure, representation_mode):

    sys.setrecursionlimit(max(1000000, vertex_count + 10))
    visited_nodes = [False] * vertex_count
    finish_stack = []

    # 1) First DFS pass to compute finish stack
    # finish_stack: list of vertices in the order
    for node in range(vertex_count):
        if not visited_nodes[node]:
            depth_first_search_order(node, visited_nodes, finish_stack,
                               graph_structure, representation_mode, vertex_count)

    # 2) Reset visited markers for second pass
    visited_nodes = [False] * vertex_count
    strongly_connected_components = []

    # 3) Second DFS pass on transposed graph to collect components
    while finish_stack:
        # Pop a vertex from the finish stack
        node = finish_stack.pop()
        if not visited_nodes[node]:
            component_vertices = []
            # Collect strongly connected component vertices
            # component_vertices: list of vertices in the current SCC
            # visited_nodes are updated through the recursive process of DFS collect
            # So we can get the group of strongly connected components
            depth_first_search_collect(node, visited_nodes, component_vertices,
                               transpose_structure, representation_mode, vertex_count)
            strongly_connected_components.append(component_vertices)
    return strongly_connected_components
```

In the First DFS function it puts a list of vertices in the order in stack.

In the Second DFS function it checks SCCs with the vertices on the stack with a transpose graph.

Through these 2 DFS, the SCCs are searched and finally returns the SCCs.

In main function we will sort the SCCs in lexicographical order to give the result that HW2_specifications requires.

e. depth_first_search_order

This is the function of first DFS in Kosaraju’s algorithm. It works like below.

① Once a DFS finishes, if there are still unvisited vertices, pick one of them and start a new DFS.

② Continue until every vertex has been visited and pushed onto the stack.

```
def depth_first_search_order(source_vertex, visited_nodes, finish_stack, graph_structure,
representation_mode, vertex_count):
    """
    First DFS pass: marks visited nodes and pushes onto finish_stack in order of completion.
    """
    # Mark the current node as visited
    visited_nodes[source_vertex] = True
    # Check the representation mode
    if representation_mode == 'matrix':
        adjacency_matrix = graph_structure
        # Check all vertices to find neighbors
        for neighbor in range(vertex_count):
            # Check if there is an edge and if neighbor is not visited
            if adjacency_matrix[source_vertex][neighbor] and not visited_nodes[neighbor]:
                # Recursively visit the neighbor until all neighbors are visited
                depth_first_search_order(neighbor, visited_nodes, finish_stack,
                    graph_structure, representation_mode, vertex_count)
    if representation_mode == 'list':
        adjacency_heads = graph_structure
        node = adjacency_heads[source_vertex]
        # Check all vertices to find neighbors
        while node:
            if not visited_nodes[node.val]:
                # Check if there is an edge and if neighbor is not visited
                depth_first_search_order(node.val, visited_nodes, finish_stack,
                           graph_structure, representation_mode, vertex_count)
            node = node.next
    if representation_mode == 'array':
        # as we give the output of adjacency_array as a tuple, we need to unpack it
        # use pointer_array to check the index of start and end of the vertex
        pointer_array, adjacency_array = graph_structure
        # Check all neighbors in the adjacency array
        for index in range(pointer_array[source_vertex], pointer_array[source_vertex + 1]):
            neighbor = adjacency_array[index]
            # check if neighbor is not visited
            if not visited_nodes[neighbor]:
                # Recursively visit the neighbor until all neighbors are visited
                depth_first_search_order(neighbor, visited_nodes, finish_stack,
                           graph_structure, representation_mode, vertex_count)
    # Push the current node onto the stack
    finish_stack.append(source_vertex)
```

To implement the functionality, I implemented the stack in finish_stack. Though, finish_stack is actually the array in python, but it works like stack in this DFS.

So, by checking the vertices in the finish_stack in second DFS, we can define the

SCCs as I explained above.

From the code above, we can see that the code differs with the selected modes. With the functions represented (which is adjacency matrix, list and array) gives different data structure, depth_first_search_order gives the equal output with the stack that has vertices searched in order.

f. depth_first_search_collect

① All vertices reached in this reversed-graph DFS form one SCC.

② Repeat until the stack is empty.

③ If you pop a vertex that's already been visited in the reversed-graph phase, simply discard it and continue.

```python
def depth_first_search_collect(source_vertex, visited_nodes, component_vertices,
transpose_structure, representation_mode, vertex_count):
    # Mark the current node as visited
    visited_nodes[source_vertex] = True
    # Add the current node to the component vertices
    component_vertices.append(source_vertex)
    # Check the representation mode
    if representation_mode == 'matrix':
        transpose_matrix = transpose_structure
        # Check all vertices to find neighbors
        for neighbor in range(vertex_count):
            # Check if there is an edge and if neighbor is not visited
            if transpose_matrix[source_vertex][neighbor] and not visited_nodes[neighbor]:
                # Recursively visit the neighbor until all neighbors are visited
                depth_first_search_collect(neighbor, visited_nodes, component_vertices,
transpose_structure, representation_mode, vertex_count)
    if representation_mode == 'list':
        transpose_heads = transpose_structure
        node = transpose_heads[source_vertex]
        # Check all vertices to find neighbors
        while node:
            if not visited_nodes[node.val]:
                # Recursively visit the neighbor until all neighbors are visited
                depth_first_search_collect(node.val, visited_nodes, component_vertices,
transpose_structure, representation_mode, vertex_count)
            node = node.next
    if representation_mode == 'array':
        # as we give the output of adjacency_array as a tuple, we need to unpack it
        transpose_pointer, transpose_array = transpose_structure
        for index in range(transpose_pointer[source_vertex],
                     transpose_pointer[source_vertex + 1]):
            neighbor = transpose_array[index]
            if not visited_nodes[neighbor]:
                # Recursively visit the neighbor until all neighbors are visited
                depth_first_search_collect(neighbor, visited_nodes, component_vertices,
                              transpose_structure, representation_mode, vertex_count)
```

In Kosaraju’s algorithm I made, second DFS is called for vertex in finish stack that is not visited. While running second DFS functions, visited vertices are updated throughout the recursion. So, the appended component_vertices in Kosaraju’s algorithm, is SCC each. So, we can find all SCCs through these 2 DFS.

g. main()

The main function consists of 4 main steps.

1. Parse input and build both the forward and reverse graph representations outside the timed section.
2. Call kosaraju_strongly_connected_components
3. After timing, sort each SCC's vertices in ascending order, then sort the list of SCCs lexicographically.
4. Print each SCC on its own line, followed by the elapsed time `(end - start) * 1000` in milliseconds at the end.

You can see the more explicit and detailed information in the code itself.

3. Result Analysis

As there may be differences in running time of finding SCC due to the method of adjacency matrix, list and array works, let’s use given 4 examples and 1 additional example I made for comparing running time.

Below is the graph that shows the number of vertices, number of edges, and edge density of given 4 given examples and example I made for comparing.

Dataset	|V|	|E|	Density
Example	300	8970	0.1000
congress-Twitter	475	13,289	0.0591
email-Eu-core	986	24,929	0.0257
soc-bitcoinalpha	3,783	24,186	0.0017
p2p-Gnutella08	6,301	20,777	0.0005

Below is the graph of running time(ms) for finding SCCs in each method.

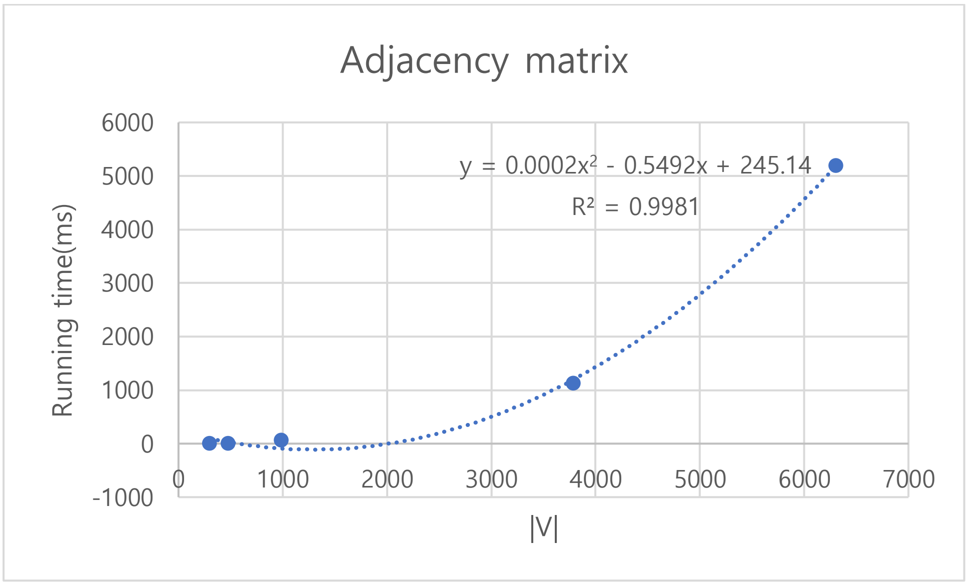
Method / Dataset	adjacency matrix	adjacency list	adjacency array
Example	3.296494	1.210107	0.649967

congress-Twitter	14.0957	2.163989	2.122071
email-Eu-core	67.50796	4.732438	6.888731
soc-bitcoinalpha	1129.359	5.191162	6.805011
p2p-Gnutella08	5197.684	5.641732	7.368615

a. Comparison based on time complexity

(1) Adjacency matrix

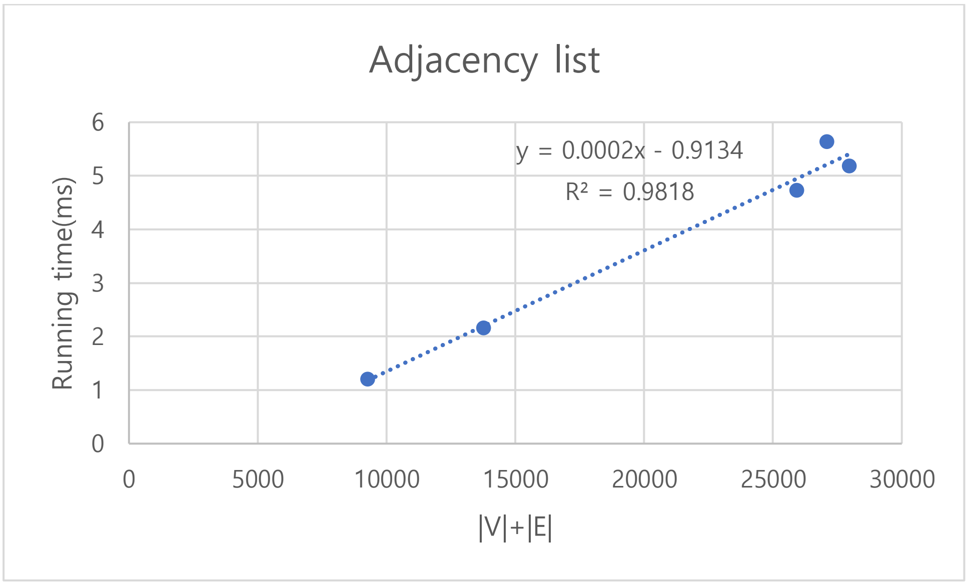
While using adjacency matrix as the method, the size of graph structure has $|V^2| + |E|$ amount of data. But the dominant part is $|V^2|$. As the Kosaraju Algorithm loops through all $|V^2| + |E|$ elements of the matrix, the time complexity of Algorithm is $O(|V^2|)$. So, let's check if the result of algorithm matches the theoretical running time.



As we can see from the graph above, the graph shows high accuracy 0.99 for 2nd order polynomial trendline. So, we can know that the result shows as we assumed.

(2) Adjacency list

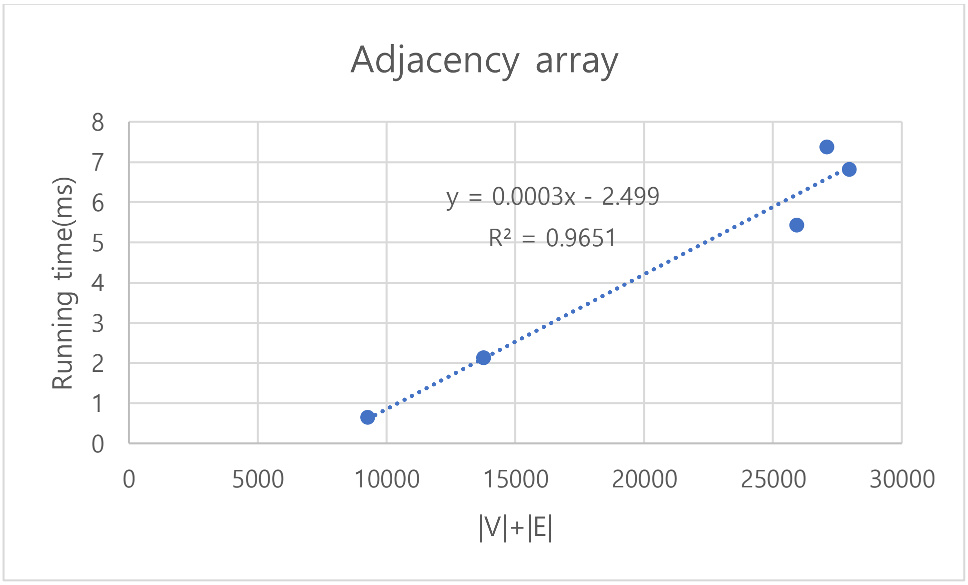
While using adjacency list as the method, the size of graph structure has $|V| + |E|$ amount of data. As the Kosaraju Algorithm loops through all $|V| + |E|$ elements of the matrix, the time complexity of Algorithm is $O(|V| + |E|)$. So, let's check if the result of algorithm matches the theoretical running time.



As we can see in the graph above, the graph shows high accuracy 0.98 in linear Trendline. So, we can know that the algorithm runs well as we assumed.

(3) Adjacency array

While using adjacency array as the method, the size of graph structure has $|V| + |E|$ amount of data which is similar to the adjacency list. As the Kosaraju Algorithm loops through all $|V| + |E|$ elements of the matrix, the time complexity of Algorithm is $O(|V| + |E|)$. So, let's check if the result of algorithm matches the theoretical running time.

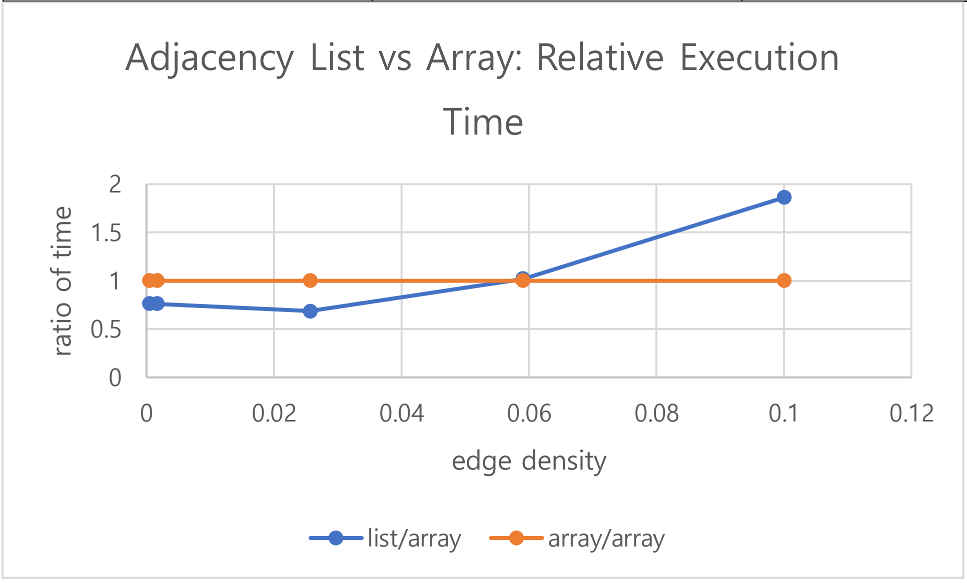


As we can see in the graph above, the graph shows high accuracy 0.96 in linear Trendline. So, we can know that the algorithm runs well as we assumed.

b. Comparison based on edge density

As the adjacency matrix takes too long time let's compare the adjacency list and array. Let's set the time of adjacency array as the standard. Then the execution time is shown below.

Ratio / Edge density	list/array	array/array
0.1	1.861798	1
0.0591	1.019753	1
0.0257	0.686983	1
0.0017	0.762844	1
0.0005	0.765643	1



As the density increases the adjacency list gives the faster running time in the graph above.

This might be occurred form the memory access of adjacency list and array. Typically, each edge is represented by a dynamically allocated node structure. Because these nodes are scattered around the heap, traversing them requires following pointers to different memory locations, which leads to frequent cache misses.

However, the adjacency array has all edge information contiguously in the single array, so iterating over a vertex's neighbors involves sequential memory accesses that are cache friendly. This is likely the cause of being faster when edge density increases.

As the graph becomes sparser, the adjacency list outperforms the adjacency array because DFS only visits actual neighbors. In contrast, with the adjacency array, the DFS loop scans every possible slot via index ranges. Even those vertices without outgoing edges, resulting in extra, unnecessary checks.

For the outputs of my code, you can visit my Github. Below is the link.
https://github.com/Ice-Moca/SNUCSE/tree/main/AlgorithmRetake/HW2

4. Reference

1. Linked list structure: https://velog.io/@yeseolee/python-%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-%EC%97%B0%EA%B2%B0%EB%A6%AC%EC%8A%A4%ED%8A%B8Linked-List-feat.LeetCode
2. Kosaraju's algorithm: https://wondy1128.tistory.com/130