**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 |

1. 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.

* 1. 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.

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

* 1. 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.

* 1. 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.

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.

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)

* 1. 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

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.

* 1. 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.

* 1. 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.

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.

* 1. 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.

1. 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 |

1. Comparison based on time complexity
2. Adjacency matrix

While using adjacency matrix as the method, the size of graph structure has amount of data. But the dominant part is . As the Kosaraju Algorithm loops through all elements of the matrix, the time complexity of Algorithm is . 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.

1. Adjacency list

While using adjacency list as the method, the size of graph structure has amount of data. As the Kosaraju Algorithm loops through all elements of the matrix, the time complexity of Algorithm is . 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.

1. Adjacency array

While using adjacency array as the method, the size of graph structure has amount of data which is similar to the adjacency list. As the Kosaraju Algorithm loops through all elements of the matrix, the time complexity of Algorithm is . 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.

1. 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.

1. Reference
2. 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
3. Kosaraju’s algorithm: https://wondy1128.tistory.com/130