

# Analysis of Algorithms: Assignment 1

## Results and Observations

By,

Kamal Sai Raj Kuncha  
(UFID: 48548114)

Ayswarya Nandan  
(UFID:14540491)

### Simulated Data:

#### Time –analysis:

(order: connected\_components, one\_cycle,shortest\_path(0))

Number of Nodes/Type of graph	N-cycle			Completely-connected			Empty			Heap			Trunc Heap M=2,200,2000,20000,200000		
10	0	2	0	1	1	1	0	0	0	0	0	0	0	0	0
1000	12	109	102	662	1	101	37	0	51	37	4	57	11	0	48
10000	1182	3251	2623	2784	4	838	786	1	1057	1190	28	308	198	4	922
100000	462	6210	504032	SO	SO	SO	8911	34	9463	25130	173	16038	3458	58	14025
1000000	4671	34326	SO	SO	SO	SO	186584	1928	25309	304320	3765	235299	82613	2043	228152

*Note: All values are in milliseconds.*

From the table it is clearly evident that, for all methods and simulations, the time increases with the number of nodes.

The time taken to get the cycle path is relatively quite small in all cases. We know that only N-cycle and completely connected graphs will have cycles. N-cycle takes the maximum time to print cycle, as it has to travel the whole graph to detect the cycle i.e., the length of the cycle is equivalent to graph size.

We can see that the time taken is less for nodes with lesser number of edges i.e. truncated heap takes least time. But in that case empty graph should have taken the least time. However, this is not the case, as empty graph means adjacency list will be tried to access for each element unnecessarily. As DFS implementation takes  $O(V+E)$  time, we know connected components and cycle detection time should increment linearly with the increase in number of nodes as well as edges.

As number of edges approaches  $N*N$  in case of completely connected, the time taken was a lot, and after a while memory usage increases so much thereby leading to stack overflow.

The stack size in Visual code was incremented to 512kB to support 100000 nodes for case 1. As we tried to increase it further, to try to accommodate 1 million, the program memory limit was over.

All the cases for which SO is marked refers to Stack overflow.

In case of 10 nodes, almost every case and method take only 0ms. In case of 1000 nodes, we see the completely connected node taking maximum time due to deep dfs traversal. In case of the other three cases completely connected is likely to take maximum time, provided sufficient stack size would have been allocated to it.

**Note:** For connected\_component and shortest\_paths the output is written to file outside the function. Hence the file writing time is not included in the calculation. But, in case of cycle, we are printing the path as and when we backtrack. So, one\_cycle includes the cycle path print time as well.

### Memory –analysis - peak memory values:

(order: connected\_components, one\_cycle, shortest\_path(0))

Numb er of Nodes /Type of graph	N-cycle			Completely- connected			Empty			Heap			Trunc Heap M=2,200,2000,2000 0,200000		
10	83.78	84.66	83.78	84.62	84.62	84.62	83.78	83.78	83.78	83.78	83.78	83.78	83.78	83.78	83.78
1000		243.2	1504.1	4331.1	4203.6	4245.7	202.2	105.7	157.1	209.1	131	225.2	187.65	126.2	176
10000	2062.8	1674	148514	SO	SO	SO	1271.7	321.7	836.2	1355.4	563	1606.5	1134.5	515	1028.5
100000	SO	SO	SO	SO	SO	SO	11975.1	2481.7	7633.1	12825.3	4833.4	17576.4	10617.1	4377	9516
1000000	SO	SO	SO	SO	SO	SO	119547.1	24084.2	76022.6	128160.4	47854.1	237466.1	105256.3	43409.2	95132.8

*Note: All values are in kilobytes (kB)*

For connected component, we are using a list of lists, where each member of the list is a list representing connected component. This list representing connected component, gives the entire list of nodes in that component.

We have bool vector of node size to mark whether the nodes have been visited or not. This vector is there in both connected component as well as cycle detection. When it comes to stack both connected component and cycle detection use recursive Depth First Search, which can lead to stack overflow in case of too deep graphs. In case of shortest paths, we have a vector of vectors for storage. For maintaining the current shortest distance of each element, we have a vector.

Memory seems to be increasing linearly with the number of nodes. For the case of 10 nodes, all cases have approximately the same peak memory usage. In case of 1000 nodes, we can see a sudden rise in the peak memory of completely connected and n-cycle. n-cycle memory usage is probably due to the multiple recursive function overheads, whereas in case of completely connected is due to large number of edges.

## ➤ Data structures used:

### Graph operations

We have represented the Graph as a class having a data member stating the number of nodes and a vector of vectors for storing the adjacency list against each element.

The parameterized constructor resizes the adjList with number of nodes.

It has addEdge operation which adds one node to adjacency list of other nodes and vice versa.

For connected components, we are using a list of lists to store the connected components and a bool vector to keep track of the visited nodes.

In case of cycle detection, we are using only a bool vector to keep track of visited nodes. The path is being printed in backtracking as and when the element is encountered.

Shortest\_paths use a priority queue to ensure minimum distance vertex is encountered, a vector containing the current minimum distance of the nodes and a vector of vectors to store the final output, i.e. path from every node to every other node.

Note: For all simulated cases recursive Depth First Search was used.

Note: We have placed detailed outputs in specific folder for simulated cases.

### Real Data:

### Final Criteria:

The following adjacency criteria were finally considered to generate graphs:

- *Adjacency criteria 1:*

Adding edges between 2 users who have given rating 5 to a movie and watched it in May,2003.

- *Adjacency criteria 2:*

Adding edges between 2 users who have given rating 3 to a movie and watched it in December,2004.

- *Adjacency criteria 3:*

Adding edges between 2 users who fall in the same range of number of 1\* ratings they have given in Jul – December ,2005 span.

If they have given between 10-20 1\* ratings, they fall into one connected component.

If they have given between 20-40 1\* ratings, they fall into another connected component.

If they have given more than 40 such ratings, they fall into another.

### Criteria Attempted:

- 1) Edges between 2 users who have given same rating to a movie and watched it in the same year as well.
- 2) Edges between 2 users who have given rating 5 to a movie and watched it in the same year as well.
- 3) Edges between 2 users who have given rating 5 to a movie and watched it in the year 2003.
- 4) Edges between 2 users who have seen 5 movies or more in common.
- 5) Edges between 2 users who have 5 movies to which they have given same ratings in common.

### Issues Faced:

Initially we tried several criteria for 1 file alone.

But due to hardware limitations the code led to issues such as: run out of memory- both our laptops had 8GB Ram, heap memory size wasn't enough and these lead to segmentation fault, and if the cases ran the execution times for them were more than 4-5 hours for a single file and wouldn't complete.

After understanding this, we decided to make the adjacency criteria, stricter, so we added specific rating and review date (only year initially) to the present criteria. There was improvement as the graph generation time was significantly reduced. But we were not able to calculate connected components sizes due to an issue of stack overflow which led to segmentation fault.

So, we finally decided to add specific month to the review date, making the criteria stricter. These led to present working adjacency criteria by processing all the files.

We tried varying the ranges between the months we could take and different ratings, in all possible combinations as to get the exact amount of data which might lead to segmentation fault, and after several experiments concluded with the above-mentioned cases.

### Time Analysis:

#### **Case 1** (Rating:5, Review Date: May,2003)

Graph Generation	794853 ms
------------------	-----------

Connected components	15172 ms
----------------------	----------

#### **Case 2** (Rating:1, Review Date: December ,2004)

Graph Generation	815574 ms
------------------	-----------

Connected components	1636311 ms
----------------------	------------

#### **Case 3** ((Rating :1, Review Date: Jul –Dec ,2005) - Based on different ranges, considering number of reviews

Graph Generation	23689 ms
------------------	----------

Connected components	7370 ms
----------------------	---------

ms-milliseconds

There is a variation in times because of the logic and hardware used.

## Memory Analysis:

Case 1 (Rating:5, Review Date: May,2003)

Graph Generation 4437954560B = 4.43GB

Connected components 5635162112B = 5.635GB

Case 2 (Rating:1, Review Date: December ,2004)

Graph Generation 5227528196B = 5.22GB

Connected components 364658176B = 364.65 MB

Case 3 (Rating :1, Review Date: Jul –Dec ,2005) - Based on different ranges, considering number of reviews.

Graph Generation 2277785600B = 2.27GB

Connected components 2286747648B = 2.286 GB

B – bytes

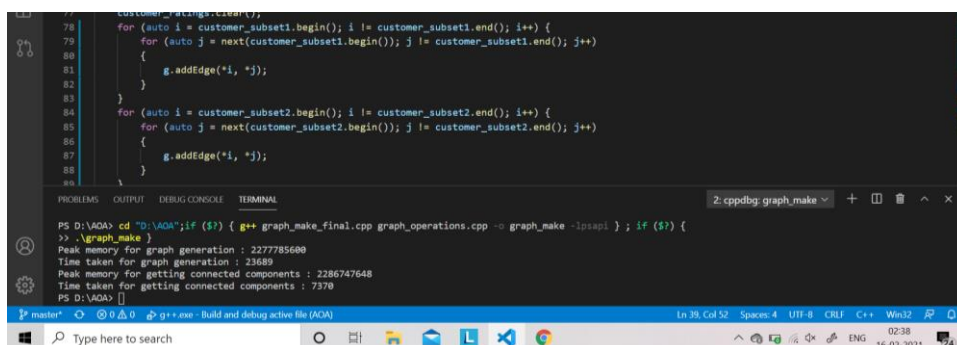
There is a variation in memory because of the logic and hardware used.

## Output Analysis:

We have placed detailed outputs in specific folder for real cases.

For case 1, after filtering out the 25540 customer nodes were created with one component size being 25316 and the rest of the component sizes being 1 and 2. The one component with 25316 is strongly connected implies that many customers have seen multiple movies and given ratings to each of the movie that they watched. Possible use of data: Since May is the beginning of summer, by analyzing this to get a perspective of what kind of movies do people like to watch in summer.

For case 2, after filtering out the 45011 customer nodes were created with one component size being 44150 and the rest of the component sizes being 1 and 2. The one component with 44150 is strongly connected. Possible use of data: Although December being holiday season, number of customer using Netflix and watching movies has increased, why the customer have given poor reviews.



```
78 for (auto i = customer_subset1.begin(); i != customer_subset1.end(); i++) {
79     for (auto j = next(customer_subset1.begin()); j != customer_subset1.end(); j++)
80     {
81         g.addEdge(*i, *j);
82     }
83 }
84 for (auto i = customer_subset2.begin(); i != customer_subset2.end(); i++) {
85     for (auto j = next(customer_subset2.begin()); j != customer_subset2.end(); j++)
86     {
87         g.addEdge(*i, *j);
88     }
89 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

2: cppdbg: graph\_make

PS D:\AOA> cd "D:\AOA";if (\$?) { g++ graph\_make\_final.cpp graph\_operations.cpp -o graph\_make -std=c++11 }; if (\$?) {  
>> .\graph\_make }  
Peak memory for graph generation : 2277785600  
Time taken for graph generation : 23689  
Peak memory for getting connected components : 2286747648  
Time taken for getting connected components : 7370  
PS D:\AOA> }

For case 3, after filtering out, graph was created 193659 customer nodes. 3 connected components of size 6229, 2643 and 14615 were obtained and remaining were 1 element connected components. Each of the three connected components are strongly connected, one being users who have given between 10-20 1 star ratings in the last 6 months of 2005, another being users who have given 20-40 1 star ratings and another for users who have given more than 40 1 star ratings. Possible use of data: To enquire with the customers if they were dissatisfied with the Netflix streaming service or choice of movies, due to such a huge amount of 1 star ratings, changing their movie recommendations.

## **Real Data Generation**

Case 1 and 2:

A vector of list was used to store the movies with their respective customers, ratings of that movie and date reviewed is stored as customer object.

A hashmap was used to store the mapping from customer ids to their unique counters.

A vector of vector was used the relevant positions where there is a change in either date or rating.

Another vector of vector was used to store the only those subsets of customers according to the adjacency criteria.

A list is used to store the output of connected components as well.

Case 3:

A hashmap was used to store the number of movie ratings per customer, with customer id being the key and number of movie ratings being the value.

Another hashmap was used to keep track of unique customer ids and give them their graph node number.

3 lists were used to split the customers based on their number of movie reviews into 3 ranges.

A list is used to store the output of connected components as well.

Note: However, case 2 in real case set was failing due to stack overflow, which is probably due to several recursive calls. So, for this case alone, we have used iterative approach. Other cases recursive proved to be consuming less time than iterative.