Resource Optimization

Kaushal Joshi, Bhagirath Talaviya, Satyam Chhatrala, Vishrut Mehta, Manan Patel, and Nemin Shah
DA-IICT, 201901199, 201901207, 201901209, 201901213, 201901257, 201901280, as@daiict.ac.in

Problem Statement

Monaco officially known as principality of Monaco is a sovereign-city state and the second smallest country (by area) in the world, located in western Europe. 38,682 residents inhabit it. Interesting part is that it is a country with 'No-Airport'. Let's consider a hypothetical situation where the Prince of Monaco decides to build the first airport in the country and launches new Airlines named "Monte-Carlo Airways" so that people of Monaco can travel to any part of the world. Our group is appointed as the management team of "Monte-Carlo Airways" in order to primarily minimize installation of new flights in existing flight-network of the world i.e. optimization of resources in general. In simple words, our task is to decide how many minimum new flights should be purchased by "Monte-Carlo Airways" as well as the minimum new connection which needs to be introduced in existing flight-network of the world such that any traveler boarding from Monaco can reach any other airport in the world.

***Abstract* - The paper talks about optimization of resources, it considers a hypothetical scenario of Monaco: located in western Europe. Monaco currently does not have an airport located in its premises. In the hypothetical scenario we wish to build a new airport in Monaco with the aim of introducing the minimum number of new flights in the existing flight-network of the world, thus optimizing resources in general. The primary objective of this paper is to come up with a solution which finds the minimum number of connections that must be made from 'Monaco Airport' such that any passenger boarding from 'Monaco' shall be able to reach all destinations in the already present mesh of airports. The paper considers three methods to come up with the solution: Brutal force solution, Kosaraju Algorithm and Tarjan's Algorithm. It explains why the Brutal force solution might not be the best choice to come up with a solution due to its exponential time complexity. It then compares the Kosaraju Algorithm and Tarjan's Algorithm which have linear time complexity and also uses the concept of strongly connected components to come up with the best possible solution that optimizes the resources. At the later stage, the paper shades light on the proposed problem solution with pseudocode as well as a computerized C++ code which implements the proposed solution and is accompanied by its results.**

Introduction

The primary objective of this paper is to devise a novel solution to the above-mentioned problem statement. More precisely, we shall present a highly optimized solution which finds the minimum number of connections that must be made from 'Monaco Airport' such that any passenger boarding from 'Monaco' shall be able to reach all destinations in the already present mesh of airports.

Evidently, this is a minimization problem (since the minimum number of connections must be made) and can also be thought of as resource minimization from a general viewpoint. This is because establishing an airplane connection requires a lot of resources such as buying new airplanes, it's associated fuel consumption, hiring new employees, setting up logistical support, etc. Therefore, minimizing the number of new connections to be introduced is equivalent to minimizing resources, or optimizing the resource utilization. Hence, it is an optimization problem at its core.

It is worth mentioning that the objective is not to find the minimum distance, rather, it is to find the minimum new connections which must be introduced.

Before diving into the proposed optimal solution for this problem, it is necessary to discuss the brutal force solution which shall motivate the necessity for optimizing the brute approach. Assume there to be V existing airports and E existing flight routes (directional) within these V airports. The naïve approach shall be as follows:

For each subset S of V airports

- Establish connections from 'Monaco' to all the airports in S.
- Check if every airport is reachable (directly or indirectly) from 'Monaco'.

The cardinality of the smallest subset which satisfies step 2 above, shall be equal to the minimum number of new connections to be made.

However, the naïve solution is practically infeasible due to its exponential time complexity. This is because step 1 takes $O(2^V)$ to generate all the possible subsets, and step 2 takes polynomial time. Hence the overall complexity of the brutal force solution takes exponential time which is not practically realizable for large V values. Thus, there is a need for a faster and better solution to find the minimal number of new connections to be added from 'Monaco'.

Our proposed solution is to convert the problem statement into an equivalent graph theory framework, and by applying certain non-trivial algorithms within this graph framework, our solution can find the minimum number of new connections in linear time complexity. The existing V airports form the V vertices of the graph, and the E airplane connections between these airports are equivalent to E directed edges between these V vertices. The airports and their connections can be succinctly represented by an adjacency list or matrix.

The core part of our proposed solution involves finding strongly connected components within this graph, which is then followed by path compression and further additional processing. It is precisely the finding of strongly connected components which reduce the time complexity from exponential time down to linear time.

In order to improve the readability of the paper, we provide the definition of strong connectivity as follows:

In graph theory, a directed graph is said to be a strongly connected graph if, from every vertex of the graph, we can reach every other vertex. In other words, for a strongly connected graph, there should be a path existing between all its pairs of vertices. A strongly connected component of a directed graph is a strongly connected subgraph. Strongly connected components of a directed graph form partitions into the directed graph resulting in the formation of subgraphs that are strongly connected. Every single vertex of a directed graph can also be considered as a strongly connected component.

Since finding strongly connected components in linear time is a non-trivial problem, a considerable part of this paper is devoted to the literature of strongly connected components, and explaining its algorithm. Our proposed detailed solution and its fully functioning computerized program are described in later sections of the paper.

The remaining paper is organized as follows: The literature review section reviews the literature on strongly connected components. The next part of the paper sheds light upon Kosaraju and Tarjan's algorithm in order to find strongly connected components in a directed graph. Further, we have described our proposed solution to the problem statement and thereby strengthening the problem solution with a pseudocode and a computerized C++ program with an analysis of space and time complexity. The result/output of the C++ code has been mentioned in the next step and doing so we concluded the paper.

LITERATURE REVIEW

Suppose that we are interested in finding the strongly connected components of a directed graph as shown below, wherein the cycle formed by 0, 1, and 2, we can see that there exists a path from each 0 to 1 and 2, 1 to 0 and 2, 2 to 0 and 1. Hence, cycle 0,1 and 2 is a strongly connected component. As there is no path from 4, 3, and 4 are 2 individuals strongly connected components. So, in the graph below, there are 3 strongly connected components. **boxes for captions.**

This problem can be connected to the problem of determining the ergodic sub chains and transient states of a Markov chain. To solve the problem of determining the ergodic sub chains and transient states, an algorithm was developed by B.L. Fox and D.M. Landy. At a later stage P.W.Purdom and I. Munro came up with virtually identical methods for solving the problem of efficient determination of the strongly connected components and transitive closure of a directed graph.

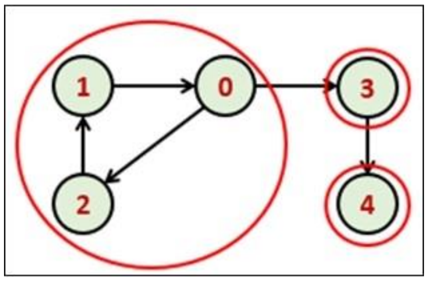


FIGURE I
SIMPLE EXAMPLE

Depth-first search was used in these algorithms. Purdom claims that the time complexity for his algorithm is kl/rE and Muhro claims the time complexity to be k max (E, V*log V), where the graph has V vertices and E edges. Basically, what their algorithm does is, it first selects a starting point and applies a depth-first search on the graph in order to construct a cycle. On finding a cycle, the vertices that are involved in formation of that cycle are marked and they are said to be in the same strongly connected component. This process goes on and on until we find each and every strongly connected components in the given graph. The only disadvantage that this algorithm has, is that when there are two small components with strong connectivity then they may be disintegrated into a bigger component. As a result of this, there might be some extra work needed in relabeling. If there is a possibility of using a more advanced approach, then the work could be completed in V*log V steps. If we think of using an efficient list merging algorithm then we can reduce the time complexity further. A more careful study of what a depth-first search does to a directed graph reveals that an O (V, E) algorithm which requires no merging of components may be devised. Two such linear time algorithms were proposed by Kosaraju and Tarjan which are described in the following sections.

I. Kosaraju Algorithm

The algorithm is used for finding strongly connected components in any given graph. In the algorithm, we can use either of the search traversal techniques i.e., DFS – Depth-first search or BFS – Breadth-first search. We would be using DFS to find strongly connected components.

The following are the steps to find all the strongly connected components via the Kosaraju algorithm:

1. Perform DFS traversal on the given graph. As we keep on visiting different nodes, we will push the nodes into a stack before returning.
2. Find a transpose of the actual graph by reversing the direction of each edge.
3. Now, pop the nodes from the stack one by one and perform DFS on the modified graph i.e., the

transpose of the graph. Keep popping the nodes from the stack.

4. Each successfully performed DFS will give us one strongly connected component of that graph.
5. Step-3 and 4 are repeated until and unless we have performed DFS on all the nodes and they are popped out of the stack.
6. At the end, we will have all our strongly connected components.

II. Tarjan Algorithm

Robert Tarjan also gave a linear time algorithm for finding the strongly connected components of a directed graph. The most important part of this algorithm is to find the head or root of each strongly connected component. The algorithm is as follows:

```
BEGIN
  INTEGER i;
  PROCEDURE STRONGCONNECT (v);
    BEGIN
      LOWLINK (v) := NUMBER (v) :=i := i +1;
      put v on stack of points;
      FOR w in the adjacency list of v DO
        BEGIN
          IF w is not yet numbered THEN
            BEGIN comment (v, w) is a tree arc;
              STRONGCONNECT (w);
              LOWLINK (v):= min (LOWLINK (v),
                                 LOWLINK (w));
            END
          ELSE IF NUMBER (w) < NUMBER (v) DO
            BEGIN comment (v,w) is a frond or cross-link;
              if w is on stack of points THEN
                LOWLINK (v) := min (LOWLINK (v),
                                    NUMBER (w));
            END;
        END;
      IF (LOWLINK (v) = NUMBER (v)) THEN
        BEGIN comment v is the root of a component;
          start new strongly connected component;
          WHILE w on top of point stack satisfies
            NUMBER (w) >= NUMBER (v) DO
              delete w from point stack and put w in
                current component;
        END;
    END;
  i := 0;
  empty stack of points;
  FOR w a vertex IF w is not yet numbered THEN
     STRONGCONNECT(w);
END;
```

- In this algorithm, two arrays, NUMBER, and LOWLINK have been used. NUMBER[v] is used to number the vertices in order according to the time. LOWLINK[v] represents the smallest index of any index that is reachable from v. A stack is used to store the vertices during the DFS calls and print the vertices when the head of SCC has been found.
- Starting from the vertex 0, each of the vertices is traversed using DFS calls, and the number for that vertex and LOWLINK value is updated every time we visit a vertex not visited earlier and push the respective vertices into the stack. When there are no further adjacent of the vertex being traversed, when we return, the LOWLINK value of the parent node is updated if a value less than the current LOWLINK value is obtained.
- If the adjacent vertex has already been visited, we will have two cases:
    1. Either it is a back edge or
    2. A cross edge.
- If the adjacent vertex is present in the stack, it is a back edge and the LOWLINK value of the vertex having back edge is updated. Otherwise, it is a cross edge and we need to neglect it.
- After traversing through the adjacent vertices of a vertex, before returning, we would check if the vertex is the head of a strongly connected component. If the LOWLINK value and NUMBER value for that vertex are the same, it is the head of SCC. So, all the elements till the head vertex are popped out from the stack and are printed as a part of 1 strongly connected component.

III. Complexity of Tarjan's Algorithm

The time and space complexity of Tarjan's Algorithm is in terms of O (V, E) for a graph having V vertices and E edges.

As Tarjan's algorithm is called once for every node which is a DFS traversal and all the edges are traversed at most once, the time complexity of this algorithm is O (| V | + | E |) where |V| = number of vertices and |E| = number of edges. The space complexity of this algorithm is linear and it is O(V).

This algorithm uses a number array of size V, a LOWLINK array of size V and a stack array to keep track of elements present in the stack of size V. In addition, there is a stack that is also of size V.

Both Kosaraju and Tarjan's algorithm use DFS traversal but Kosaraju's algorithm requires 2 DFS traversals whereas Tarjan's algorithm requires only 1 DFS traversal. So, we will be using Tarjan's algorithm for our project. paper.

SYSTEM DESCRIPTION

I. Problem Solution

Let's breakdown problem statement and try to understand it in a simpler way using Graph theory.

Let's consider Node as Airport and Edge as Flight route. Now, it is clearly stated in the problem statement that there exists a "flight-network" which connects different airports of the world. So, let's depict this existing "flight-network" with a directed graph of 'n' nodes and 'm' directed edges. Each node in a graph represents an airport and each directed edge represents that there exists a flight that flies from the source node (i.e., source Airport) to the destination node (i.e., destination Airport). In other words, each directed edge represents that it is possible for people from the source node of that edge to travel to the destination node of that edge.

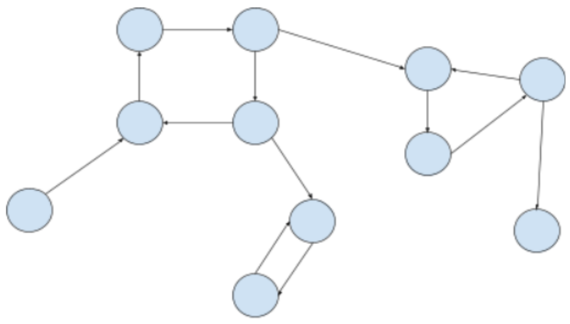


FIGURE III
SIMPLER EXISTING "FLIGHT-NETWORK" GRAPH

Now as per our problem statement, it is stated that Monaco earlier didn't have any Airport so the obvious inference is that none of the nodes of existing "flight-network" graph represent newly built Monaco Airport.

So, let's add an isolated node corresponding to 'Monaco Airport' into our existing "flight-network" graph. It is isolated because currently there are no incoming or outgoing flights from that airport.

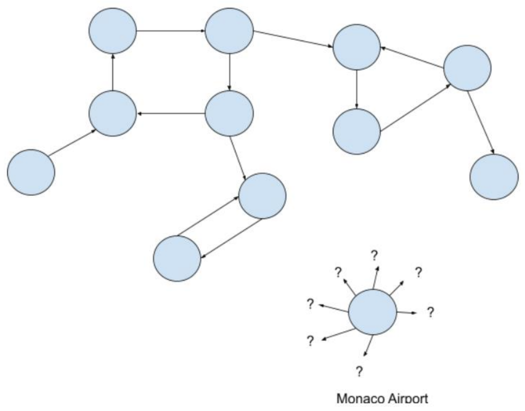


FIGURE IV
INTRODUCTION OF MONACO AIRPORT

Now let us consider that "Monte-Carlo Airways" decides to launch 'x' number of new flights boarding from 'Monaco Airport' so that passengers/people of Monaco can travel to any Airport present in the existing "flight-network" graph.

Our group appointed as a management team of "Monte-Carlo Airways" have to find the minimum value of 'x' as well as destination nodes of each of these new 'x' flights which need to be installed.

II. Solution

Now let's dive into the proposed optimal solution for this problem.

- **Process Input:**

  Existing "Flight-network" graph will be fed as an input to our code which will be processed and stored in the form of an adjacency list for easy access and our own convenience.

- **Find Strongly connected components**:

  Let's first understand the term "strongly connected components" in a simpler way before jumping into how to find them.

  "Strongly connected components" are subsets of nodes in a graph that are connected in such a way that there always exists a directed path between any two nodes of that subset. Reflecting this definition in reference to our problem statement means that there always exists a flight route between any two different airports which are present in the same "strongly connected component" of the existing "flight-network" graph.

  This, in turn, infers that if a person reaches any Airport (node) say 'X' then he/she will be able to reach all other Airport (node) say 'Y' such that both node-X and node-Y are present in the same "strongly connected component".

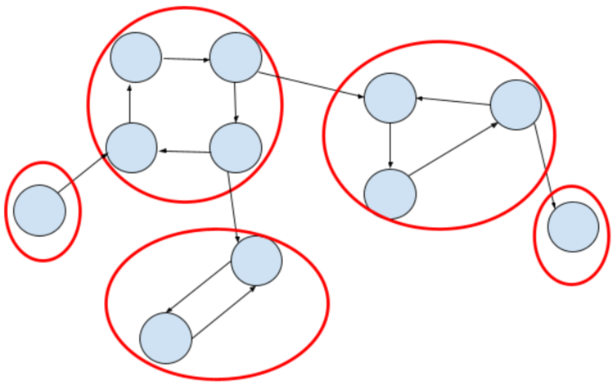


FIGURE V
STRONGLY CONNECTED COMPONENT DEPICTED BY RED CIRCLE

Now let's address the question of How to find strongly connected components in a graph? This in itself is a non-trivial problem that can be solved using Kosaraju's Algorithm and Tarjan's Algorithm (Refer to section-II of Literature Review for detailed explanation).

- **Compress the graph based on components**:
  This step is implemented purely for the sake of simplicity and to avoid confusion in code. We know for the fact that there always exists a flight route between any two nodes of the same strongly connected component, So we're compressing the graph in such a way that we consider the whole strongly connected component as a single node irrespective of number of nodes which are present in that particular strongly connected component.
  We'll call this resultant graph a "compressed-graph" where each node will represent a strongly connected component of the "flight-network" graph.

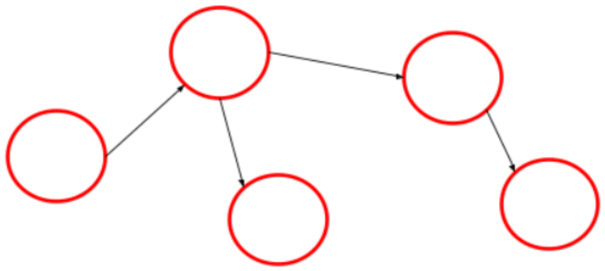


FIGURE VI
COMPRESSED GRAPH WHERE EACH NODE IS REPRESENTS STRONGLY CONNECTED COMPONENT

Also, each of these nodes in a compressed graph will have a corresponding representative node which is an arbitrary node from that strongly connected component.

- **Find Number of nodes with in-degree=0 in the compressed graph**:
  This one is the last step of our solution where all those nodes whose in-degree = 0 in the compressed graph are our answer. New flight routes which need to be introduced will be from Monaco Airport to the corresponding representative node of all those nodes whose in-degree=0 in the compressed graph.

III. Pseudocode

- **Process Input**
  1. n – Total number of Airports
  2. airports(n) - A vector of strings of size n which stores the list of airports
  3. m – Total number of existing flight routes
  4. l, r – strings indicating a flight route from l to r

- **Find strongly connected components (SCC) using Tarjan's Algorithm**
  For each unvisited node, we would call dfs which performs the tarjan algorithm explained before. In this algorithm, we find the strongly connected components and store it in rep vector with a node u as the representative node of the strongly connected component. So, vector rep contains all the strongly connected components.

- **Find Indegree of all the representative nodes of each strongly connected components.**
  Now, we create a compressed graph from the vector rep containing strongly connected components and find indegree of node. Indegree of a node u is the number of edges directed towards node u.

- **Process Output**
  On implementing the entire code, we get the following output:
  1. New flight routes that are to be added
  2. cnt – Number of new flight routes added

IV. Code

```
#include<bits/stdc++.h>

using namespace std;

const int maxN=1e5;

// stores list of Airports
vector<string> airports;

//stores Airport's index in vector airports
unordered_map<string,int> mp;

// adjacency list of existing flight network
vector<vector<int>> g;

//To check if Airport is visited or not
vector<bool> vis(maxN);

// stores discovery time
vector<int> disc(maxN);

//low[v] represents node with earliest discovery time
//reachable from v
vector<int> low(maxN);

// represents if node-v is present in stack-st or not.
vector<bool> stackMem(maxN);

stack<int> st;

int timer=0; // current time
```

```
//rep[u] is representative node of connected component in
//which node-u is present
vector<int> rep;

// implementation of Tarjan's Algorithm
void dfs(int u)
{
    vis[u] = true;
    disc[u] = low[u] = timer++;
    stackMem[u] = true;
    st.push(u);
    for(auto v:g[u])
    {
        if(!vis[v])
        {
            dfs(v);
            low[u]=min(low[u],low[v]);
        }
        else if(stackMem[v])
            low[u]=min(low[u],disc[v]);
    }

    if(low[u]==disc[u])
    {
        while(st.top()!=u)
        {
            stackMem[st.top()]=false;
            rep[st.top()]=u;
            st.pop();
        }

        stackMem[u]=false;
        rep[u]=u;
        st.pop();
    }
}

int main()
{
    cout<<"Total Number of Airports: "<<endl;
    long n;    cin>>n;

    cout<<"List of Airport: "<<endl;
    airports.resize(n);
    g.resize(n);

    for(int i=0;i<n;i++)
    {
        cin>>airports[i];
        mp[airports[i]]=i;
    }

    cout<<"Starting Airport is Monaco Airport: MNC "<<endl;
    cout<<"Total number of existing Flight routes: "<<endl;

    long m;
    cin>>m;
    cout<<"List Flight routes: "<<endl;

    //forming adjacency list of existing flight-network graph
    while(m--)
    {
        string l,r;
        cin>>l>>r;
        g[mp[l]].push_back(mp[r]);
    }

    rep.resize(n);

    for(int i=0;i<n;i++)
   {
        if(!vis[i])
        {
            dfs(i);
        }
    }
//stores in-deg of representative node of each connected
//component in compressed graph
    vector<int> indeg(n,0);

// adjacency list of compressed graph
    vector<vector<int>> g2(n);

    for(int u=0;u<n;u++)
    {
        for(auto v:g[u])
        {
            if(rep[u]!=rep[v])
            {
                g2[rep[u]].push_back(rep[v]);
                indeg[rep[v]]++;
            }
        }
    }

    //output
    cout<<"New flight routes to be added: "<<endl;
    long cnt=0;

    for(int i=0;i<n;i++)
    {
        if(rep[i]==i && indeg[i]==0)
        {
            cout<<"MNC "<<airports[i]<<endl;
            cnt++;
        }
    }

    cout<<"Count of new flight routes added: "<<cnt<<endl;

    return 0;
}
```

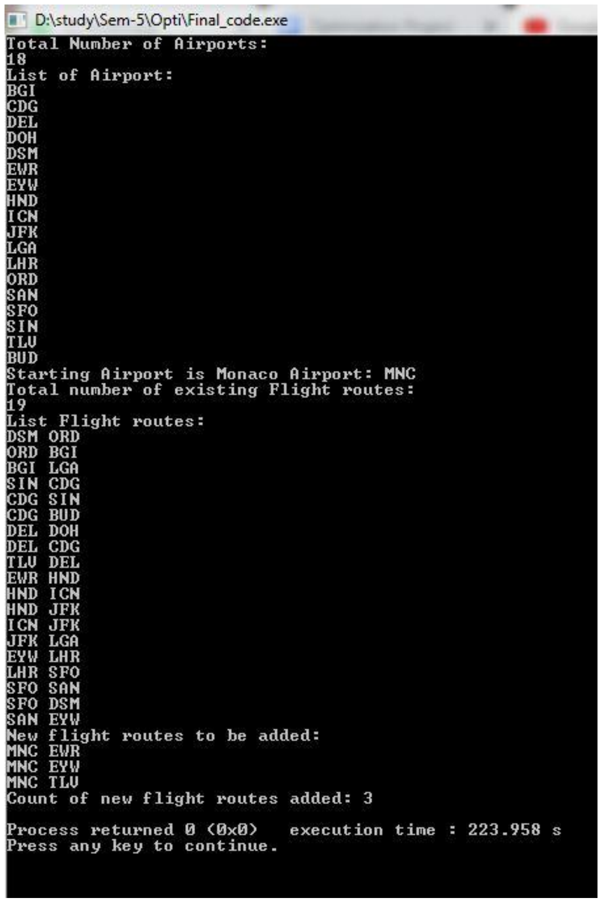
V. Complexity Analysis

Let |V| is Total number of Airports/nodes in existing flight-network graph and |E| is Total number of edges in existing flight-network graph.

- Space Complexity: O(|V| +|E|))
  As we are using Adjacency list to store existing flight-network graph

- Time Complexity: O(|V| +|E|))
  As Tarjan's algorithm is called once for every node which is a DFS traversal and all the edges are traversed at most once.

VI. Output of Code



```
D:\study\Sem-5\Opti\Final_code.exe
Total Number of Airports:
18
List of Airport:
BGI
CDG
DEL
DOH
DSM
EWR
EYW
HND
ICN
JFK
LGA
LHR
ORD
SAN
SFO
SIN
TLV
BUD
Starting Airport is Monaco Airport: MNC
Total number of existing Flight routes:
19
List Flight routes:
DSM ORD
ORD BGI
BGI LGA
SIN CDG
CDG SIN
CDG BUD
DEL DOH
DEL CDG
TLV DEL
EWR HND
HND ICN
HND JFK
ICN JFK
JFK LGA
EYW LHR
LHR SFO
SFO SAN
SFO DSM
SAN EYW
New flight routes to be added:
MNC EWR
MNC EYW
MNC TLV
Count of new flight routes added: 3

Process returned 0 (0x0)   execution time : 223.958 s
Press any key to continue.
```

FIGURE VII
OUTPUT OF IMPLEMENTED CODE

TECHNOLOGY REVIEW

We used the C++ programming language (the 17th edition) for writing the computerized program of our proposed algorithm wherein the GNU GCC GPLv3+ compiler was used to compile the source code. At the time of simulations, our local machine was equipped with an Intel i-5 chip operating at around 3 GHz and two DDR4-4400 slots each of them hosting 8 Gigabytes of Random-Access Memory, this was in addition to the NVIDIA GeForce GTX 1040 graphics processor unit. The drastic difference between the time complexities of the brutal force solution and our proposed algorithm is indeed verified by our exhaustive simulations.

CONCLUSION

In this paper we present a novel algorithmic approach to solve a resource optimization problem which is introduced in the beginning of this paper. More specifically, by viewing the problem statement into a graph theoretical framework, we propose a modus operandi to determine the minimum number of new connections that must be introduced (from a specific node) into an already existing mesh of connections such that the graph becomes a single component. This, we believe, is best tackled by understanding the idea of strongly connected components in a graph which is thoroughly reviewed in this paper. By comparing and contrasting various existing algorithms currently in the literature and their time complexities, we chose Tarjan's algorithm as a base for our proposed solution. Our proposed solution including the pseudocode revolve around the following steps of operations: (i) Finding the strongly connected components in the graph, (ii) Using path compression, compress each component into it's representative element, and (iii) Finding such zero indegree elements, and introducing new connections with them. This procedure is then conscientiously programmed into a computerized C++ code which implements the proposed solution and is accompanied by its results.

REFERENCES

[1] Tarjan, Robert (1972). Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing, 1(2), 146–160. doi:10.1137/0201010

[2]. B.L. Fox AND D. M. LANDY, an algorithm for identifying the ergodic sub chains and transient states of a stochastic matrix, Comm. ACM, 11 (1968), pp. 619-621.

[3]. P.W. PURDOM, a transitive closure algorithm, Tech. Rep. 33, Computer Sciences Department, University of Wisconsin, Madison, 1968.

[4]. I. MUNRO, Efficient determination of the strongly connected components and transitive closure of a directed graph, Department of Computer Science, University of Toronto, 1971.

[5] https://www.tutorialspoint.com/assets/questions/images/182943-1531219955.jpg

[6] https://m.youtube.com/watch?v=qz9tKlF431k&t=2403s