Intimate Algorithm

Student's Name: Mohammad Anagreh

Univeristy of Tartu- Estonia

Abstract Graph is a non-linear data structure which consists of vertices and edges. It's given by *G = (V, E)* where *V* is a group of vertices and *E* is a group of weighted edges among that vertices. There are many applications of the graph which widely used in social media, communication, maps and etc. In this project, we present a new algorithm to find the matching attributes among a group of people (Vertices). Each person has two lists of attributes (like and Dislike), the algorithm will check and find an intimate value which is the difference between convergence and spacing attributes among all people in the network. The proposed algorithm properly works in a complete unweighted graph to find the intimate value between every two people in the graph consequently. Each vertex has a unique edge with another edge in the graph, the algorithm checks all that vertices one-by-one to find the closest person. The data structure in this work is a big player to organize the routing and searching operations. An array of Linked List-based data structure is used to link and to save the data of the vertices. As well as, we introduce another algorithm to find the closest person to each person in the graph based on the intimate algorithm.

1.1 Introduction

A graph *G = (V, E)* in this context is made up by vertices or nodes V connected with each other by edges or lines E, while each node has a specific value. To get from one vertex to another vertex in the graph *G*, one may have a choice of edges *E* to traverse which may pass through several other vertices. There are two main types of graph, undirected graphs which means there is no distinction direction between its two vertices. Directed graph which means that the edges have a specific direction from the vertices to another one [1]. A graph is a mathematical structure that can be sparse dense or a mixture of both used to form pairwise relations $E \subset V \times V$ between graph objects.

A complete graph is a connected graph has a unique edge between every pair of vertices in the graph. For a given number of vertices, there's a unique complete graph, which is often written as K_n, where n is the number of vertices. The complete graph can be a directed or undirected graph (both cases). So, a complete digraph is a directed graph in which every pair of distinct vertices in the given graph G is connected by a pair of unique edges (one in each direction) [2,3].

Graphs are today used in many real-world systems, such as in social media networks, communications, biological, physical, community detection and sparse linear solvers. One of the most important branches of the graph theory is the matching algorithms that are widely used in many applications like social media and dating site. Matching algorithms are a set of algorithms used to solve the main issues in this branches which is a graph matching problems in graph theory. The main idea is when a set of edges E must be drawn that do not share any vertices V in the graph G, this is what it's called a matching problem [4]. Graph matching problems applications are very common in daily applications and activities. The most important issues start from online social media, matchmaking and dating websites, to medical residency placement programs (medical software's). Matching algorithms are used in areas spanning scheduling, comparison, identical issues, planning, the pairing of vertices, and network flows based-network of the graph [5,6]. More specifically, matching strategies based on matching graph algorithms are very useful in flow network algorithms such as the Ford-Fulkerson algorithm [7] and the Edmonds-Karp algorithm[8], which is an implementation of algorithm for computing the maximum flow in a flow network based on graph [9,10].

Graph matching problems generally consist of making connections within graphs using a group of edges E that do not share common a group of vertices V in the graph G, such as pairing cars in an according to their model or capacity of engines; or it may consist of creating a bipartite matching [11].

In this project, we modify the conception of the matching among the vertices in the graph G. The matching her is according to the given attributes of the vertices. Each vertex has a group or more of attributes, that will be led us to modify the definition of the graph to be able to satisfy the goal of our project which is finding the matching's attributes among a group of vertices in a complete graph G. In our job we suppose that each vertex has two lists of attributes like and dislike. The algorithm should find that's identical matchings in attributes for the two given lists.

This work organized as follows: we start by giving an abstract about the project, the first section is an introduction to the project. The second section is a brief preliminary of the main concept we are going to use. Proposed the algorithm and details about the algorithms in the section number 3. Discussion, in the section four is testing and getting the result . Section number 5 is a conclusion and some future work.

1.2 Preliminaries

1.2.1 Complete Graph

Mathematically, a **complete graph** is a graph has a group of vertices connected with each other by a unique edge. Each two vertices in the graph G as a unique edge. Complete graph can be a directed or undirected graph. **Directed graph** is graph G in which every pair of distinct vertices V in the given graph G is connected by a pair of unique edges E. While **undirected graph** is a graph in which every pair of distinct vertices V in the graph G is connected by a unique edge E, (See figure1).

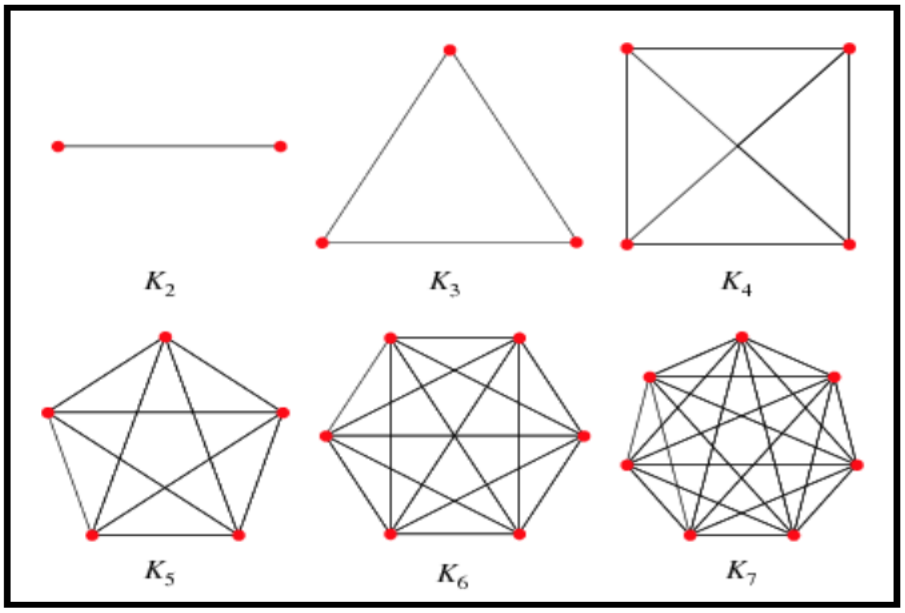


Figure 1: Complete graphs for several number of vertices

The complete graph on *n* vertices *V* is denoted by $K_{n.}$ *k* means the complete graph while n the number of vertices *n* the given graph *G*. Mathematically and based on the definition of the complete graph, the number of edges in the complete graph is given by *n(n-1)/2* while in the regular graph of degree *n-1*. The diameter and radius of the complete graph *G* for *n* vertices is given by $\begin{Bmatrix} 0 & n \leq 1 \\ 1 & otherwise \end{Bmatrix}$.

1.3 Algorithm and Proposed work

1.3.1 Over view a bout proposed method

In social media and dating site, there are many people consists of a huge network connected with each other. The network extends within a big geographical area within different countries or continents. Each person in the network has a group of characteristics and attributes. We looking for creating an algorithm or a protocol that's can find the identical attributes among that people in the one network. Such a work can be helpful in some applications based-social media or dating site. This is proposed method for finding the matching attributes in a group of people connected to each other in a huge network. We suppose that the connections of the people in a huge network are as a graph. Each person is a vertex and other connections ad reaching to each other is an edge. So, each person has ability to have a connection with a person in the network (as a direct connections). In case the connection can be directly within each other, we suppose that the graph we have is complete graph. Furthermore, each person can connect with anther person in the network and both has ability for that. There is no restriction in connections among all the people in the graph we have, see the Figure 2.

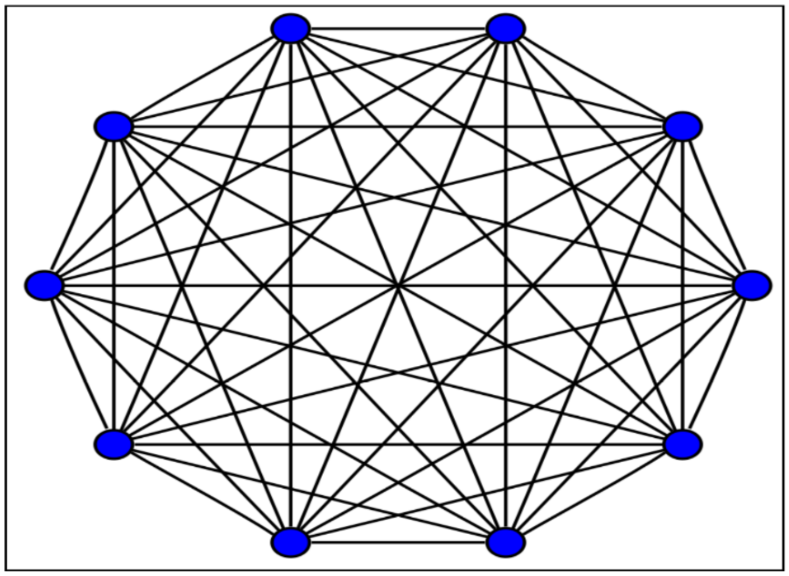


Figure 2: The edges in the complete graph

1.3.2 Data Structure and Framework

The general definition of the Graph is given *G = (V,E)* while *V = (v1, v2,..., vn)* is the group of vertices and E is a group of edges *E = (e1, e2, ..., en)*. If *(vi, vj)* ∈ E, then the *vi* and *vj* are neighbor's vertices. Representation of the data's graph is based on some data structure technique. We are going to keep the attributes of the vertices in the vertices itself. The link list based- array is used to represent and keep that data for the whole vertices in the graph. Our proposed method consider that each person has two lists of attributes based on negative or positive attributes (like and dislike). The data of the like and dislike lists will be saved in an array inside the linked list, the number of the vertex and the name of the person,...,etc. The data for each vertex will be presented inside the struct as follows:

```
struct NodeData
{
       int ver_num ;  // number of the vertex
       string ver_name;  // name of the person
       string Like[Attributes_Size];  // the list of the Like-attributes
       string Dislike[Attributes_Size]; // the list of the Dislike-attributes
};
```

The data structure considers as a big player in our proposed method for finding the identical attributes among a group of the people in the graph. Each a vertex has four main elements. Element number one is a *ver_num* which is to keep the number of vertices, data type is an integer number. Element two is a ver_name which is to keep the name of person, the data type is string. Note, *ver_num* is a unique number while the *ver_num* can be repeated. The element number three in the struct is an array of the *Like* which is used to keep the positive attributes, the data type is a string or an integer number. The last element in the main struct is *Dislike* which is for keeping the negative attributes and the data type is a string or integer. The size of both lists (*like* and *Dislike*) is based on the number of attributes we have.

It's important to note that in our implementation, we gave same number of the attributes in both list *like* and *Dislike* which is 5. It can be different, but in this case, we have to change the size for each list. After defining the struct, we can define array from the struct we created, there is no specific size of the array. The size of that array is based on the number of the people in the network.

In the proposed method, there are another two structs which are the *cluster* and *closest* structs respectively. The cluster struct is to save the processed data after finding the relationship between the people in the graph, it's consists from eight elements, as follows:

```
struct cluster
{
	int ver_num;       // number of locations
	int num1;          // the number of first vertex
	string ver_name1;  // the name of the person
	int num2;          // the number of the second vertex
	string ver_name2;  // the name of the second person
	float convergence; // the element of convergence among two persons
	float spacing;     // the element of spacing among two person
	float intimate;    // the element of intimate
};
```

First element is a *ver_num* which is to represent the number of locations. By another way, how many relationships we have after finding the calculation. Usually, it given by the number of edges that the algorithm will construct, it's *n(n-1)/2*. Element number two is *num1* which is the number of a person (it's a unique value), while the element number three is *ver_name1* which is the name of that *num1*. Element 4 is *num2*, which is the number of the person who's has a relationship with the person *num1*. The element *ver_name* is the name of the second person. The last three elements in the cluster struct are shows the relationship among any two people in the graph, in the next section gives more details about them.

The last struct in our algorithm is closest struct which is used by algorithm to give the information about the close4st person to each person in the graph.

Closest struct has five elements, the first element is the *ver_num* which has the number of the person (or vertex). It's important to note that this element has a unique value. The second element is ver_name which is to keep the name of the person. The third elements is an array has the numbers of the vertices which have a relationship with the ver_nam(and ver_name), the size of this array is usually the graph size(the number of the people or vertices in the graph). And the same size of the remains two elements in the struct. Th elements four is an array of the friend's name.

```
struct closest
{
       int ver_num;
       string ver_name;
       float friend_num[Graph_Size];
       string friend_name[Graph_Size];
       float  intimate[Graph_Size ];
};
```

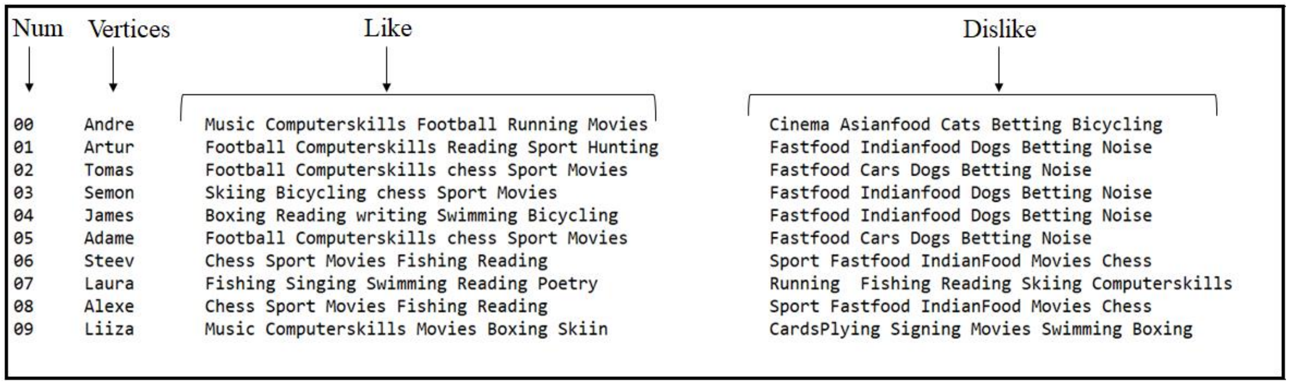
The last element in the struct is array of the intimate values, which is used to save the closest person to each person in the graph (more details about in the next section).

According to the main struct in our proposed algorithm, data should be organized in some file to be a readable and savable by the algorithm, see the figure 3.

The data of each vertex in the given graph should be prepared in a completely row in the given file, such data should be prepared as a pre-request operation before start running the algorithm. each row has a data for one person (or vertex), it has four main section. Section one is the number of the person (or vertex) in the graph, this number should be a unique value.

Section number two is the name of the person and should match the num. Each person in the graph has a number and name.

The third section is the like-list which has a group of a positive attributes for the person. As example, first row is 00 the name of the person is *Andre*.



Num	Vertices	Like	Dislike
00	Andre	Music Computerskills Football Running Movies	Cinema Asianfood Cats Betting Bicycling
01	Artur	Football Computerskills Reading Sport Hunting	Fastfood Indianfood Dogs Betting Noise
02	Tomas	Football Computerskills chess Sport Movies	Fastfood Cars Dogs Betting Noise
03	Semon	Skiing Bicycling chess Sport Movies	Fastfood Indianfood Dogs Betting Noise
04	James	Boxing Reading writing Swimming Bicycling	Fastfood Indianfood Dogs Betting Noise
05	Adame	Football Computerskills chess Sport Movies	Fastfood Cars Dogs Betting Noise
06	Steev	Chess Sport Movies Fishing Reading	Sport Fastfood IndianFood Movies Chess
07	Laura	Fishing Singing Swimming Reading Poetry	Running Fishing Reading Skiing Computerskills
08	Alexe	Chess Sport Movies Fishing Reading	Sport Fastfood IndianFood Movies Chess
09	Liiza	Music Computerskills Movies Boxing Skiin	CardsPlying Signing Movies Swimming Boxing

Figure 3: The data structure of the graph

He has a five positive attributes (something the person likes), *Music, Computer Skills, Footballs, Running* and *Movies.* This attribute can be different based on the person. The last section of the data file is Dislike attributes which is the negative attributes of the person (something the person don't likes), for Andre is *Cinema, Asianfood, Cats, Betting* and *Bicycling.* The whole data in then given file should be saved in the main struct which is *NodeData* before starting processing the calculation based on the proposed algorithm.

1.4.3 Proposed Algorithm (Intimate Algorithm)

The main goal of our proposed algorithm is to find the identical attributes among a group of people addressed in a complete graph. Our proposed algorithm prepared to find the identical attributes in a complete graph, but with a modification in routing it can help to be applied in an incomplete graph. There is no specific number of people (Vertices) in the graph. As well as, there is no specific number in the people's attributes or the number of attributes lists. Furthermore, we can apply the numbers we have, but we have to manipulate the structures.

The proposed algorithm has four loops iterations that can iterate the whole vertices in the graph and the attributes of each vertex, See Algorithm1. The main processes are to find the identical attributes for the whole vertices in graph G with each other. Loops (i) and (j) are to route all the vertices in the given complete graph. Loop (i) starts from 0 to the graph size (the number of vertices), while loop (j) starts from (i+1) to the graph size. The index of loop (j) started from (i+1) to avoid checking the vertex with itself. So, the number of iterations of the total calculation based on the actual amount of the unweighted edges which is given by *n*(n-1)/2*. Loops (k) and (L) are to check the convergence and spacing of the attributes, then giving the intimate vale for each two vertices in the graph.

```
Algorithm 1: Intimate Algorithm
Input: Node_graph[], Graph_size, Attributes_size
Output: Cluster[]
   1.  Attributes_constant = (float) Constant / Attributes_Size;
   2.  For(i=0 to Graph_size){
   3.       For(j=i+1 to Graph_size){
   4.             For(k=0 to Attribute_size)
   5.              {
   6.                 For(L=0 to Attribute_size)
   7.                  {
   8.                    if (Node_Graph[i].Like[k] == Node_Graph[j].Like[L])
   9.                       Like_convergence = Like_convergence + Attriutes_constant
   10.                   if (Node_Graph[i].DisLike[k] == NodeGraph[j].DisLike[L])
   11.                       DisLike_convergence = DisLike_convergence + Attributes_constant
   12.                   if (NodeV[i].Like[k] == NodeV[j].DisLike[L])
   13.                      Like_spacing = like_spacing + Attributes_constant
   14.                   if (NodeV[i].DisLike[k] == NodeV[j].Like[L])
   15.                      DisLike_spacing = DisLike_spacing + Attributes_constant;
   16.                }
   17.          }
   18.          convergence = like_convergence + DisLike_convergence
   19.            if (spacing1 >= spacing2)
   20.                spacing = spacing1
   21.           Else
   22.                spacing = spacing2;
   23.          intimate = convergence - spacing
   24.          clusterN[count].ver_num = counter++
   25.          clusterN[count].num1 = NodeV[i].ver_num
   26.          clusterN[count].ver_name1 = NodeV[i].ver_name
   27.          clusterN[count].num2 = NodeV[j].ver_num
   28.          clusterN[count].ver_name2 = NodeV[j].ver_name
   29.          clusterN[count].convergence = convergence
   30.          clusterN[count].spacing = spacing
   31.          clusterN[count].intimate = intimate
   32.          count = count + 1
   33.          convergence = spacing = spacing1 = spacing2 = 0
   34.          like_convergence = DisLike_convergence = intimate = 0
   35.      }
   36.   }
   37.  Return   Cluster[]
```

Operators of Intimate Algorithms

In our proposed algorithm, there are three main operators, convergence, spacing and the intimate. Finding the values of intimate algorithm's operators by processing the algorithm itself without giving any initial value. Initial values for three operators should be zero before starting and should be zero as well once the algorithm routes the vertices.

1) Convergence operator

Convergence is a flag to show the value of the identical attributes among each two persons in the given graph. There are two main lists of attributes, like and Dislike. So, there are two sub convergences, *like_convergence* and *Dislike_convergence.*

- **Like_convergence** is to add the constant of attributes based on the first list which is *like* list, *like_convergence* given by applying the following equation:

```
like_convergence = like_convergence + Attributes_constant
```

Attributes constant is a one unit of the attributes size. The total units of the attribute's constants are the same of the attribute size which is given by

```
attributes size = u1+u2+u3+….+un = 1

attributes constant = 1/attributes size
```

Example 1: In our implementation there are five attributes in the like list and in the Dislike list, Then:

```
Attributes constant = 1 / 5 = 0.2
```

Suppose two people (James and Jones) in the graph both of them have same attributes such as: {*Music, Football, ComputerSkills, Reading, Runnin*}.

The *Like_convergence* for both of them is given by

```
like_convergence = like_convergence + Attributes constant

like_convergence = 0.2(music)+ 0.2(Football)+0.2(Computerskills)+0.2(Reading)+0.2(Runnin)=1
```

From the given attributes we can notice, that both James and jones have same the five attributes so the value of *like_convergence* among them is 1

- **Dislike_convergence** is the same as previous one (Like_converegnce). The main difference here that both persons have same attributes in case, they don't like something. As an example, both James and jones don't like Asian food, so they agree with each other in case a something they don't like.
  Dislike_convergence given by applying the following equation:

$$Dislike\_convergence = Dislike\_convergence + Attributes\_constant$$

Example 2: Suppose that two-person James and Jones have same attributes in case the Dislike list (something they don't like) as follows: *CardsPlying, Signing, Movies, Swimming and Boxing*

Dislike_convergence=$0.2(CardsPluing)+ 0.2(signing)+0.2(Movies)+0.2(Swimming)+0.2(Boxing)=1$

We can recognize that both James and Jones have a same attribute in case something they don't like it

Now, after finding the convergences for both lists in negative (Dislike) and positive(like) attributes, we can find the general convergence which is given by the following equation:

$$convergence = like\_convergence + Dislike\_convergence$$

In the two examples above for both James and Jones, the $Lik\_convergence$ is 1 and the $Dislike\_convergence$ is 1, so by applying the equation:

$convergence = like\_convergence + Dislike\_convergence$

$Convergence = 1 + 1 = 2$

Which is the optimal case, both James and Jones agree with each other in all attributes they like and they don't like, so both of them are perfect intimate.

2) Spacing operator

This operator in case two person don't agree with each other in some attributes in the two lists Like and Dislike. There are two subs spacing each one for a list.

- **Like_spacing**

In case there are one attributes or more in like list are not identical the attributes in the like list with the second person. Like_spacing is given by applying following equation:

```
Like_spacing = Like_spacing + Attributes_constant;
```

Attributes_size = 5

Attributes_constant =1/attributes_size

Attributes_constant =1/5 = 0.2 (each attribute unit is 0.2)

Example 3: Suppose two-person James and Jones has some different attributes as follows:

*James= {Music, Computerskills, Football, Running, Movies}**_Like***
*James= {Fastfood, Indianfood, Dogs, Betting, Noise}**_Dislike***

*Jones = {Music, Computerskills, Football, Bicycling, Fastfood}**_Like***
*Jones = {Movies, Asianfood, Cats, Betting, Running}**_Dislike***

Now, we compare the like_list of James with Dislike-list of Jones, in case that James like *x*, while jones dislike *x*, then they don't agree with each other in case of *x*. So, the spacing is (0.2), we keep comparing to the all attributes in the lists. Each time there is a spacing, we add (0.2) to the Like_spacing.

From the lists above, we can recognize that James Like *{Running}* while Jones don't like it, so the spacing is (0.2). As well as, James like *{Movies}* while Jones don't like it, so the spacing is (0.2). the total *like_spacing* is:

Like_spacing = 0.2(Running) + 0.2 (Movies) = 0.4

The spacing between James and Jones is 0.4 in case of like-List

- **Dislike_spacing**

In Dislike spacing is vice versa, in case the first-person dislike something with the second person like it, it's a spacing as well, is given by following equation:

```
Dislike_spacing = Dislike_spacing + Attributes_constant;
```

Example 4: Suppose two-person James and Jones has some different attributes as follows:

*James= {Music, Computerskills, Football, Running, Movies}**_Like***
*James= {Fastfood, Indianfood, Dogs, Betting, Noise}**_Dislike***

```
Jones = {Music, Computerskills, Football, Bicycling, Fastfood}_Like
Jones = {Movies, Asianfood, Cats, Betting, Running}_Dislike
```

By comparing the Dislike list of James and like list of Jones, we found that James dislike *{FastFood}* while Jones like it, so this is a spacing (0.2)

Dislike_spacing = 0.2(Fastfood) = 0.2

There is no more spacing.

So, the spacing between James and Jones is 0.2 in case of Dislike-list

Now, after finding the spacing for both person James and Jones, we find the total spacing among them, is given by following equation:

$$Spacing\ is \begin{cases} like_{spacing} & Dislike_{spacing} \le like_{spacing} \\ Dislike_{spacing} & otherwise \end{cases}.$$

So, the spacing between James and Jones is 0.4

Because the Like_spacing is 0.4 while dislike_spacing is 0.2 (Maximum value is taken)

The idea that we have two lists (like and dislike), that the identical attributes is not only in case both of persons like something, maybe by agreeing with each other in case of something they don’t like, as an example both of them agree with each other that they don't like fast food. mathematically, negative multiply negative is positive, see the Figure 4.

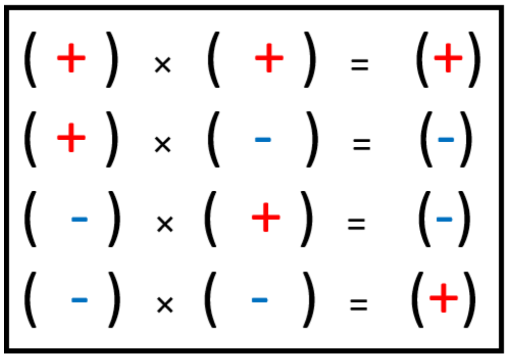


Figure 4: identical attributes (like, Dislike)

3) Intimate operator

Intimate operator is the last one and it’s the most important operator which is tell the average relationship between each two persons in the graph. Finding the intimate is after finding the Convergence and spacing, which is the difference(subtraction) between convergence and spacing by applying the following equation:

```
intimate = convergence - spacing;
```

Example 5: Suppose there are two-person Andre and Artur in the same graph, and both of them have five attributes in two list like and dislike.

The convergence between them is 2.0

The spacing is 0.6

Finding the intimate between them by applying the equation

intimate = convergence – spacing

intimate = 2.0 – 0.6 = 1.4

The intimae between Andre and Artur is 1.4

- In case that convergence is 2.0 and spacing is 0.0,

intimate = convergence – spacing
intimate = 2.0 – 0.0 = 2.0

This is the optimal case which is 2.0 (maximum point).
Both two persons agree each other in a case like-list and dislike-list and they have 100% identical attributes. So, they are a perfect intimate.

Finding the Closest Person

After finding the total relationship among whole persons in the given graph G, and saved all that's data in the cluster link list. It's the time to find the closest person to each other, by another way, the classification of the closest person to each other in the whole network. In some cases, maybe more then one person is a closest person to some vertex. Algorithm 2 is to find the closest person to each vertex in the given graph *G* based on the results of the intimate algorithm.

```
Algorithm 2: Finding the closest vertex based on Intimate Algorithm
Input: Cluster[], Graph_size
Output: closest[]
   1. closest[] = cluster[]
   2. For(x=0 to Graph_size)
   3.   {
   4.     max = closestN[x].intimate[y]
   5.     For(y=0 to Graph_size)
   6.       {
   7.         if (closestN[x].intimate[y] > max)
   8.          {
   9.           max = closestN[x].intimate[y]
   10.          name = closestN[y].ver_name
   11.          num = closestN[y].ver_num
   12.         }
   13.      }
   14.     Print(closest[x].num1, closest[x].name1, “closest to”, num, name, ”intimate=” max)
   15.     max = 0.0
   16.  }
```

1.4 Testing and Result

We implemented the Intimate Algorithm (Algorithm 1) and the Closest Person Algorithm (Algorithm 2) which is based on the intimate algorithm, all the related codes, functions, struct, reading the file and printing out of the intimate algorithm implemented and written in C++ language in the environment of Visual Studio. NET 2017.

We prepared the file for the given graph which has the numbers and names of the vertices and the two lists (Like and Dislike) for each vertex in the graph. As we said before, the graph we implemented in our job is a complete graph with 10 vertices as shown in the figure 5.

Each vertex has an unique edge with all vertices in the complete graph, started from Andre (00) then to Artur (01), Tomas (02),…etc. Once Vertex of Andre check all the vertices in the complete graph routes to another vertex and start checking all the vertices and their attributes one by one.

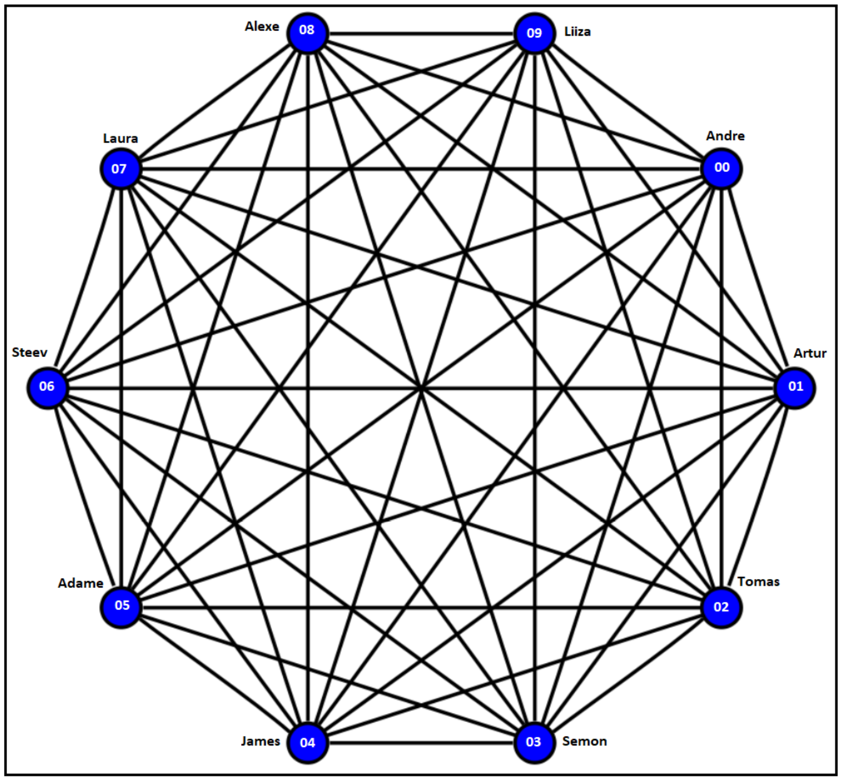
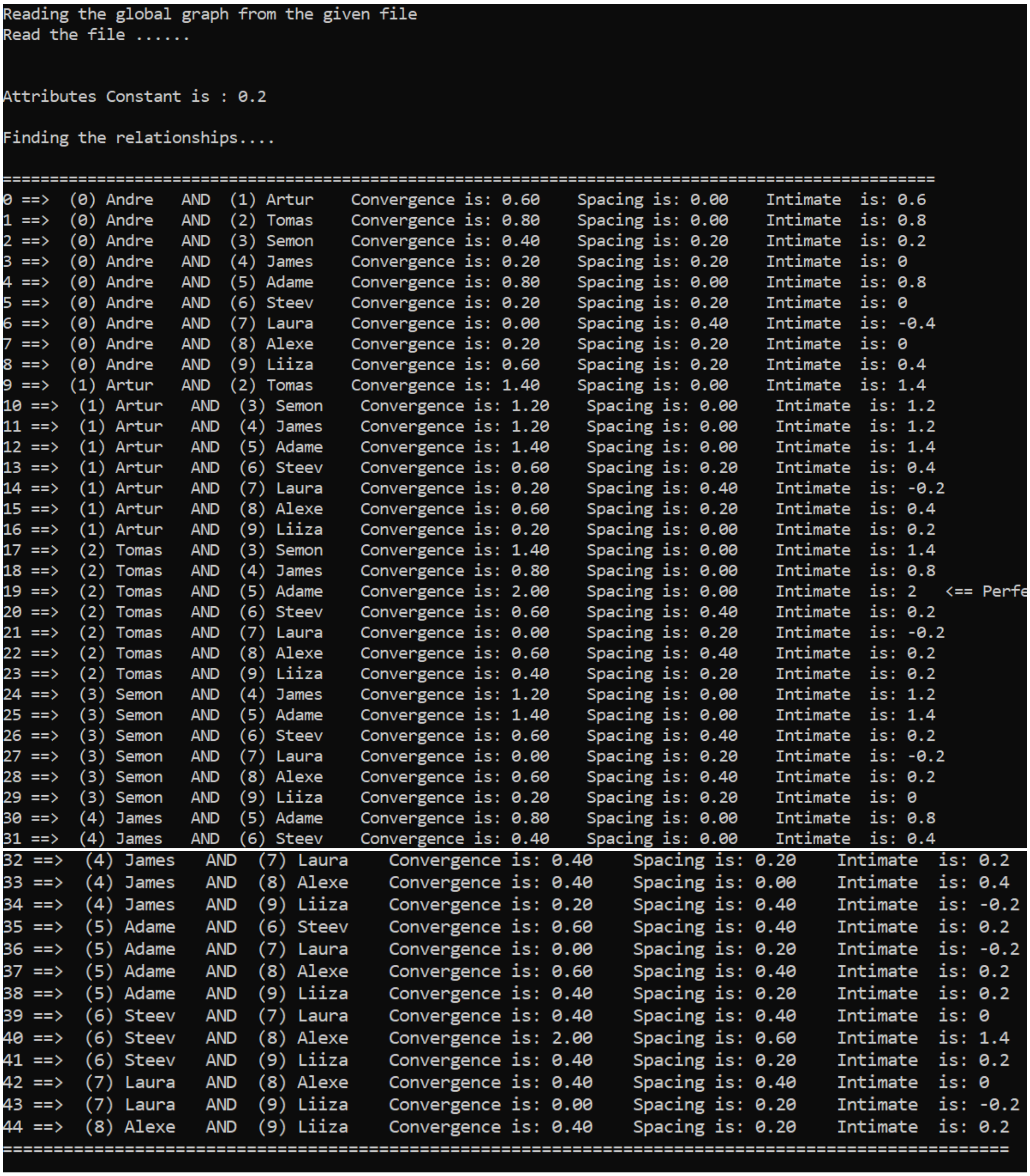


Figure 5: A framework of complete graph in the project

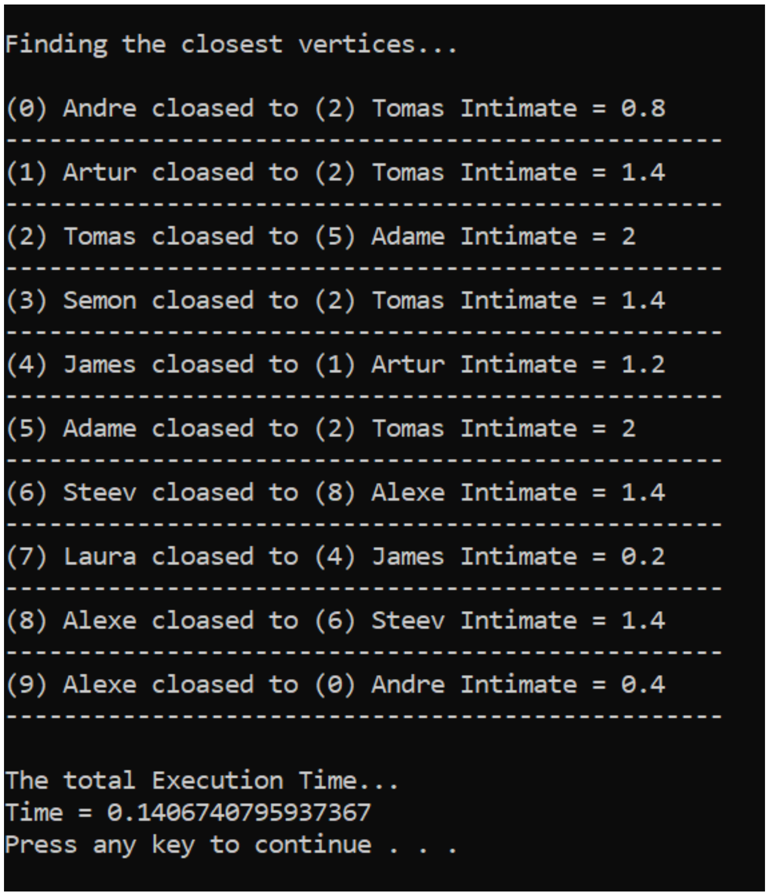
The following print screen for the running of Intimate Algorithm shows the relationship among all persons in the given graph, it shows as well the values of the convergence, spacing and the difference among them which is appear in intimate Value.



```
Reading the global graph from the given file
Read the file ......


Attributes Constant is : 0.2

Finding the relationships....

================================================================================
0 ==>  (0) Andre   AND  (1) Artur    Convergence is: 0.60    Spacing is: 0.00    Intimate  is: 0.6
1 ==>  (0) Andre   AND  (2) Tomas    Convergence is: 0.80    Spacing is: 0.00    Intimate  is: 0.8
2 ==>  (0) Andre   AND  (3) Semon    Convergence is: 0.40    Spacing is: 0.20    Intimate  is: 0.2
3 ==>  (0) Andre   AND  (4) James    Convergence is: 0.20    Spacing is: 0.20    Intimate  is: 0
4 ==>  (0) Andre   AND  (5) Adame    Convergence is: 0.80    Spacing is: 0.00    Intimate  is: 0.8
5 ==>  (0) Andre   AND  (6) Steev    Convergence is: 0.20    Spacing is: 0.20    Intimate  is: 0
6 ==>  (0) Andre   AND  (7) Laura    Convergence is: 0.00    Spacing is: 0.40    Intimate  is: -0.4
7 ==>  (0) Andre   AND  (8) Alexe    Convergence is: 0.20    Spacing is: 0.20    Intimate  is: 0
8 ==>  (0) Andre   AND  (9) Liiza    Convergence is: 0.60    Spacing is: 0.20    Intimate  is: 0.4
9 ==>  (1) Artur   AND  (2) Tomas    Convergence is: 1.40    Spacing is: 0.00    Intimate  is: 1.4
10 ==>  (1) Artur   AND  (3) Semon    Convergence is: 1.20    Spacing is: 0.00    Intimate  is: 1.2
11 ==>  (1) Artur   AND  (4) James    Convergence is: 1.20    Spacing is: 0.00    Intimate  is: 1.2
12 ==>  (1) Artur   AND  (5) Adame    Convergence is: 1.40    Spacing is: 0.00    Intimate  is: 1.4
13 ==>  (1) Artur   AND  (6) Steev    Convergence is: 0.60    Spacing is: 0.20    Intimate  is: 0.4
14 ==>  (1) Artur   AND  (7) Laura    Convergence is: 0.20    Spacing is: 0.40    Intimate  is: -0.2
15 ==>  (1) Artur   AND  (8) Alexe    Convergence is: 0.60    Spacing is: 0.20    Intimate  is: 0.4
16 ==>  (1) Artur   AND  (9) Liiza    Convergence is: 0.20    Spacing is: 0.00    Intimate  is: 0.2
17 ==>  (2) Tomas   AND  (3) Semon    Convergence is: 1.40    Spacing is: 0.00    Intimate  is: 1.4
18 ==>  (2) Tomas   AND  (4) James    Convergence is: 0.80    Spacing is: 0.00    Intimate  is: 0.8
19 ==>  (2) Tomas   AND  (5) Adame    Convergence is: 2.00    Spacing is: 0.00    Intimate  is: 2   <== Perfe
20 ==>  (2) Tomas   AND  (6) Steev    Convergence is: 0.60    Spacing is: 0.40    Intimate  is: 0.2
21 ==>  (2) Tomas   AND  (7) Laura    Convergence is: 0.00    Spacing is: 0.20    Intimate  is: -0.2
22 ==>  (2) Tomas   AND  (8) Alexe    Convergence is: 0.60    Spacing is: 0.40    Intimate  is: 0.2
23 ==>  (2) Tomas   AND  (9) Liiza    Convergence is: 0.40    Spacing is: 0.20    Intimate  is: 0.2
24 ==>  (3) Semon   AND  (4) James    Convergence is: 1.20    Spacing is: 0.00    Intimate  is: 1.2
25 ==>  (3) Semon   AND  (5) Adame    Convergence is: 1.40    Spacing is: 0.00    Intimate  is: 1.4
26 ==>  (3) Semon   AND  (6) Steev    Convergence is: 0.60    Spacing is: 0.40    Intimate  is: 0.2
27 ==>  (3) Semon   AND  (7) Laura    Convergence is: 0.00    Spacing is: 0.20    Intimate  is: -0.2
28 ==>  (3) Semon   AND  (8) Alexe    Convergence is: 0.60    Spacing is: 0.40    Intimate  is: 0.2
29 ==>  (3) Semon   AND  (9) Liiza    Convergence is: 0.20    Spacing is: 0.20    Intimate  is: 0
30 ==>  (4) James   AND  (5) Adame    Convergence is: 0.80    Spacing is: 0.00    Intimate  is: 0.8
31 ==>  (4) James   AND  (6) Steev    Convergence is: 0.40    Spacing is: 0.00    Intimate  is: 0.4
32 ==>  (4) James   AND  (7) Laura    Convergence is: 0.40    Spacing is: 0.20    Intimate  is: 0.2
33 ==>  (4) James   AND  (8) Alexe    Convergence is: 0.40    Spacing is: 0.00    Intimate  is: 0.4
34 ==>  (4) James   AND  (9) Liiza    Convergence is: 0.20    Spacing is: 0.40    Intimate  is: -0.2
35 ==>  (5) Adame   AND  (6) Steev    Convergence is: 0.60    Spacing is: 0.40    Intimate  is: 0.2
36 ==>  (5) Adame   AND  (7) Laura    Convergence is: 0.00    Spacing is: 0.20    Intimate  is: -0.2
37 ==>  (5) Adame   AND  (8) Alexe    Convergence is: 0.60    Spacing is: 0.40    Intimate  is: 0.2
38 ==>  (5) Adame   AND  (9) Liiza    Convergence is: 0.40    Spacing is: 0.20    Intimate  is: 0.2
39 ==>  (6) Steev   AND  (7) Laura    Convergence is: 0.40    Spacing is: 0.40    Intimate  is: 0
40 ==>  (6) Steev   AND  (8) Alexe    Convergence is: 2.00    Spacing is: 0.60    Intimate  is: 1.4
41 ==>  (6) Steev   AND  (9) Liiza    Convergence is: 0.40    Spacing is: 0.20    Intimate  is: 0.2
42 ==>  (7) Laura   AND  (8) Alexe    Convergence is: 0.40    Spacing is: 0.40    Intimate  is: 0
43 ==>  (7) Laura   AND  (9) Liiza    Convergence is: 0.00    Spacing is: 0.20    Intimate  is: -0.2
44 ==>  (8) Alexe   AND  (9) Liiza    Convergence is: 0.40    Spacing is: 0.20    Intimate  is: 0.2
================================================================================
```

As well as, the following print screen of the running intimate algorithm show the closest person to each person in the graph. As example, the closest person to Andre is Tomas, the intimate value between them is only 0.8 .

While the intimate value of Adam with Tomas is 2 (maximum). So, it's a perfect intimate, the closets person to Adame is Tomas because of intimate value is maximum point.



```
Finding the closest vertices...

(0) Andre cloased to (2) Tomas Intimate = 0.8
-------------------------------------------------
(1) Artur cloased to (2) Tomas Intimate = 1.4
-------------------------------------------------
(2) Tomas cloased to (5) Adame Intimate = 2
-------------------------------------------------
(3) Semon cloased to (2) Tomas Intimate = 1.4
-------------------------------------------------
(4) James cloased to (1) Artur Intimate = 1.2
-------------------------------------------------
(5) Adame cloased to (2) Tomas Intimate = 2
-------------------------------------------------
(6) Steev cloased to (8) Alexe Intimate = 1.4
-------------------------------------------------
(7) Laura cloased to (4) James Intimate = 0.2
-------------------------------------------------
(8) Alexe cloased to (6) Steev Intimate = 1.4
-------------------------------------------------
(9) Alexe cloased to (0) Andre Intimate = 0.4
-------------------------------------------------

The total Execution Time...
Time = 0.1406740795937367
Press any key to continue . . .
```

1.5 Conclusion

In this work, we introduced a new algorithm to find the relationship between a group of people in a given graph. The relationship in term the closest person to each other. The searching and routing will be based on two given lists have a different attribute to each person. We presented some a simple mathematics calculation for some proposed measurement to determine the relationship between the people and to find the closest person. Intimate Algorithm is based on a complete graph. The future work is replacing the algorithm and modify it to be able to route and search within any given graph. Its challenge because each vertex has to route and check all vertices in the graph. In case we have incomplete graph and there are weighted edges, we have to do some pre-calculation to determine where is the given graph is a reachable to all vertices by any vertices, within shortest path or not. For the future work, we suppose to use a Hamiltonian cycle to route all vertices in the graph and save the data of each vertex in a Queue before start checking.

References:

[1]- P. Hell, J. Nesetril, Graph and homomorphisms, Book Oxford University Press.
[2]- https://en.wikipedia.org/wiki/Complete_graph.
[3]- http://mathworld.wolfram.com/CompleteGraph.html
[4]- https://en.wikipedia.org/wiki/Matching_(graph_theory)
[5]- https://brilliant.org/wiki/matching-algorithms/
[6]- Zwick, Uri. "The smallest networks on which the Ford-Fulkerson maximum ow procedure may fail to terminate." Theoret. Comput. Sci 148.1 (1995): 165-170.
[7]- Ford, L.R. and Fulkerson, D.R., 1956. Maximal flow through a network. Canadian journal of Mathematics, 8, pp.399-404.
[8]- Dinic, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". Soviet Mathematics - Doklady. Doklady. 11: 1277–1280.
[9]- Dinitz, Yefim. "Dinitz'algorithm: The original version and Even's version." Theoretical computer science. Springer, Berlin, Heidelberg, 2006. 218-240.
[10]- Edmonds, Jack, and Richard M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems." Journal of the ACM (JACM) 19.2 (1972): 248-264.
[11]- https://en.wikipedia.org/wiki/Graph_matching

Appendix

```
// C++ Code, Source.cpp : Intimate Algorithm (Finding the same attributes among a group of people)
// A project for Algorithmics Course, University Of Tartu
// Mohammad Anagreh

#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <omp.h>
#include <sstream>
#include <stdio.h>
#include<algorithm>


using namespace std;

#define Graph_Size 10  //  Number of Vertices (Number of People)
#define Attributes_Size 5  //  Number of atributes in both arrays Like and Dislike
#define Constant 1.0   //  The total number of attributes is 1


float like_convergence = 0.0;   // A counter to count the number of converging attributes (Positive)
float Dislike_convergence = 0.0;// A counter to count the number of converging attributes (Negative)
float spacing1 = 0.0;     // A counter to count the number of incompatible attributes (like ==> Dislike)
float spacing2 = 0.0;     // A counter to count the number of incompatible attributes (Dislike ==> like)
float spacing = 0.0;      // A counter to calculate uncomputable attributes (for both cases)
float convergence = 0.0;  // A counter to calculate convergence attributes (for both cases)
float intimate = 0.0;     // A counter to calculate the difference between convergence and spacing

// The main structure of the vertex in the graph
struct NodeData
{
        int ver_num ;
        string ver_name;
        string Like[Attributes_Size];
        string Dislike[Attributes_Size];
};


// The structure of the relationship of the vertices
struct cluster
{
        int ver_num;
        int num1;
        string ver_name1;
        int num2;
        string ver_name2;
        float convergence;
        float spacing;
        float intimate;
};


// The structure of the closest vertex
struct closest
{
        int ver_num;
        string ver_name;
        float friend_num[Graph_Size];
        string friend_name[Graph_Size];
        float  intimate[Graph_Size ];

};

// Prototypes of the functions
NodeData *ReadFile();
```

```
void WriteFile();



int main()
{

        double start = omp_get_wtime();
        // The graph size of the complete graph which is n(n-1)/2
        cluster clusterN[Graph_Size*(Graph_Size - 1) / 2];
        closest closestN[Graph_Size];
        NodeData *NodeV = ReadFile(); // To read the data of the graph

        /*
        int j = 0;
        for (int i = 0; i < Graph_Size; i++)
        {
                cout << NodeV[i].ver_num << " ==> " << NodeV[i].ver_name << endl;
                for (int j = 0; j < Attributes_Size; j++)
                {
                        cout << "Like " << NodeV[i].Like[j] << endl;
                }
                for (j = 0; j < Attributes_Size; j++)
                {
                        cout << "Dislike " << NodeV[i].Dislike[j] << endl;
                }

                cout << "--------------------------------" << endl;
        }
        */

        //A counter to provide the number of the vertices
        int counter = 0;

        //A counter to index the size of the clauster graph
        int count = 0;


        //To provide the number of attributes for each vertex in the grapg G
        float Attributes_constant;
        Attributes_constant = (float) Constant / Attributes_Size;
        cout << endl;
        cout <<"Attributes Constant is : "<< Attributes_constant << endl;


        // The main processes to find the attributes matching for the whole vertices in graph G
        // Loops i and j are to route all the vertices in the given complete graph
        // Loops k and l are to check the convergence and spacing of the attributes
        // like_convergence is a counter to count the number of identical attributes among vertices (Likes)
        // Dislike_convergence is a counter to count the number of identical attributes among vertices(
DisLikes)
        // Convergence is a counter to find the convergence among vertices in both cases (Like + Dislike)
        // Spacing1 is a counter to find the spacing from vertex i to k
        // Spacing2 is a counter to find the spacing from vertex k to i
        // Spacing  is the highest spacing eathier spacing1 or spacing2
        // Intimate is the difference between the convergence and spacing

        for (int i = 0; i < Graph_Size; i++)
        {
                for (int j = i+1; j < Graph_Size; j++)
                {
                        for (int k = 0; k < Attributes_Size; k++)
                        {
                                for (int l = 0; l <Attributes_Size; l++)
                                {
                                        if (NodeV[i].Like[k] == NodeV[j].Like[l])
                                                like_convergence = like_convergence + Attributes_constant;
                                        if (NodeV[i].Dislike[k] == NodeV[j].Dislike[l])
```

```
                                                Dislike_convergence = Dislike_convergence + 
Attributes_constant;
                                        if (NodeV[i].Like[k] == NodeV[j].Dislike[l])
                                                spacing1 = spacing1 + Attributes_constant;
                                        if (NodeV[i].Dislike[k] == NodeV[j].Like[l])
                                                spacing2 = spacing2 + Attributes_constant;
                                }
                        }
                        convergence = like_convergence + Dislike_convergence;
                        if (spacing1 >= spacing2)
                                spacing = spacing1;
                        else
                                spacing = spacing2;
                        if (convergence == 2)
                                intimate = 2;
                        else
                                intimate = convergence - spacing;


                        //  Finding the total relationships of the givin graph
                        clusterN[count].ver_num = counter;

                        clusterN[count].num1 = NodeV[i].ver_num;
                        clusterN[count].ver_name1 = NodeV[i].ver_name;

                        clusterN[count].num2 = NodeV[j].ver_num;
                        clusterN[count].ver_name2 = NodeV[j].ver_name;

                        clusterN[count].convergence = convergence;
                        clusterN[count].spacing = spacing;
                        clusterN[count].intimate = intimate;


                        count = count + 1;
                        counter = counter + 1;

                       // Reset all locations in the memory before a new iteration starts
                        convergence = 0.0;
                        spacing = 0.0;
                        spacing1 = 0.0;
                        spacing2 = 0.0;
                        like_convergence = 0.0;
                        Dislike_convergence = 0.0;
                        intimate = 0.0;
                }
        }



        int x = 0, y = 0;
        int con = 0;

        // from the cluster graph, find the closest vertex for each vertex in the cluster graph
        // based on intimate flag

        for (int i = 0; i < Graph_Size*(Graph_Size - 1) / 2; i++)
        {

                           x =  clusterN[i].num1;
                           y =  clusterN[i].num2;
                           char* str = &clusterN[i].ver_name2[0];
                           char* str1 = &clusterN[i].ver_name2[0];

                           //cout << "--------------------------" << endl;
                           //cout << x << " " << y << endl;
                           closestN[x].ver_num = (float) x;
                           closestN[x].ver_name = clusterN[i].ver_name1;
```

```
                            closestN[x].friend_num[y] = (float)  y;
                            closestN[x].friend_name[y] = str;
                            //cout << "str = " << str << endl;
                            closestN[x].intimate[y] = clusterN[i].intimate;
                            // cout << closestN[x].ver_num <<"  "<<closestN[x].friend_num[y]<<
                             "  Intimate is:"<< closestN[x].intimate[y]<< endl;
                            // cout << closestN[x].friend_name[y] << "  " << closestN[x].ver_name << "  " <<
endl;
                            // cout << y << " " << x;
                            closestN[y].ver_num = (float) y;
                            closestN[y].ver_name = clusterN[i].ver_name1;

                            closestN[y].friend_num[x] = (float)x;
                            closestN[y].friend_name[x] = str;
                            //cout << "str =" << str << endl;
                            closestN[y].intimate[x] = clusterN[i].intimate;
                            // cout << closestN[y].ver_num << "  " << closestN[y].friend_num[x]
                            << "  Intimate is:" << closestN[y].intimate[x] << endl;
                            // cout << closestN[y].ver_name << "  " << closestN[y].friend_name[x] << endl;
        }

        /*
        x = 0; y = 0;
        for (int i = 0; i < Graph_Size*(Graph_Size - 1) / 2; i++)
        {

                x = clusterN[i].num1;
                y = clusterN[i].num2;
                cout << "============" << endl;
                cout << closestN[x].ver_num << "  " << closestN[x].friend_num[y] <<
                "  Intimate is:" << closestN[x].intimate[y] << endl;
                cout << closestN[x].friend_name[y] << "  " << closestN[x].ver_name << "  " << endl;

                cout << closestN[y].ver_num << "  " << closestN[x].friend_num[y] <<
                "  Intimate is:" << closestN[y].intimate[x] << endl;
                cout << closestN[x].ver_name << "  " << closestN[x].friend_name[y] << endl;


        }
        */


        // To print out the list of relationships among the vertices
        cout << endl;
        cout << "Finding the relationships...." << endl;
        cout << endl;
        cout << "=====================================================================" << endl;
        for (int i = 0; i < Graph_Size*(Graph_Size - 1) / 2; i++)
        {
                cout << clusterN[i].ver_num <<"==>(" <<clusterN[i].num1 <<")"<< clusterN[i].ver_name1 <<"AND
"
                        << " (" << clusterN[i].num2 << ") " << clusterN[i].ver_name2;
                printf("    Convergence is: %.2f", clusterN[i].convergence);
                printf("    Spacing is: %.2f", clusterN[i].spacing);
                if (clusterN[i].intimate == 2)
                    cout << "    Intimate  is: " << clusterN[i].intimate <<"   <== Perfect"<<endl;
                else
                        cout << "    Intimate  is: " << clusterN[i].intimate << endl;

        }
        cout << "======================================================================" << endl;


        //To print out the closest vertices
        cout << endl;
        cout << "Finding the closest vertices..." << endl;
        cout << endl;

        float max = 0.0;
```

```
	string name;
	int num;
	for (int x = 0; x < Graph_Size; x++)
	{
		max = closestN[x].intimate[y];
		for (int y = 0; y < Graph_Size; y++)
		{
			//cout << closestN[x].intimate[y] << endl;
			if (closestN[x].intimate[y] > max)
			{
				max = closestN[x].intimate[y];
				name = closestN[y].ver_name;
				num = closestN[y].ver_num;
			}

		}
		cout << "(" << closestN[x].ver_num << ") " << closestN[x].ver_name <<
		" cloased to (" << num << ") " << name << " Intimate = " << max << endl;
		cout << "-------------------------------------------------" << endl;
		max = 0.0;
	}

	// Get the total execution time and finalize
	cout << '\n';
	cout <<"The total Execution Time... "<< endl;
	double end = omp_get_wtime();
	printf("Time = %.16g", end - start);
	cout << endl;
	system("pause");
	return 0;
}

// This function to read the data of the graph from given file
// Save the data in the memory based on the main structure NodeData

NodeData *ReadFile()
{
	NodeData *Node = new NodeData[Graph_Size];


	try {
		ifstream infile;
		infile.open("GraphFile.txt");
		cout << "Reading the global graph from the given file" << endl;
		cout << "Read the file ......" << endl;
		cout << endl;
		string ch;
		for (int i = 0; i < Graph_Size; i++)
		{
			getline(infile, ch);

			//cout << ch<< endl;

			std::istringstream iss(ch);
			std::string sub;
			int count = 0;
			int count_like = 0;
			int count_Dislike = 0;
			int count_chct = 0;
			while (iss >> sub)
			{
				if (count == 0)
				{
					Node[i].ver_num = stoi(sub);
					count = count + 1;
					continue;
				}
				if (count == 1)
				{
```

```
                                        Node[i].ver_name = sub;
                                        count = count + 1;
                                        continue;
                                }
                                if (count >= 2 && count <= 6)
                                {
                                        Node[i].Like[count_like] = sub;
                                        count = count + 1;
                                        count_like = count_like + 1;
                                        continue;
                                }
                                if (count >= 7 && count <= 11)
                                {
                                        Node[i].Dislike[count_Dislike] = sub;
                                        count = count + 1;
                                        count_Dislike = count_Dislike + 1;
                                }

                        }
                }
                infile.close();
        }

        catch (std::exception& e)
        {
                cout << "Please check the source file" << endl;
        }

        // To print out the data for the checking
        /*
        int j = 0;
        for (int i = 0; i < 5; i++)
        {
                cout << Node[i].ver_num << "==>" << Node[i].ver_name << endl;
                for (int j = 0; j < 5; j++)
                {
                        cout << "Like " << Node[i].Like[j] << endl;
                }
                for (j = 0; j < 5; j++)
                {
                        cout << "Dislike " << Node[i].Dislike[j] << endl;
                }
                for ( j = 0; j < 5; j++)
                {
                        cout<<"Chct " << Node[i].chct[j] << endl;

                }
                cout << "-------------------------------" << endl;
        }
        */

        return Node;
}
```