Friend Suggestion Algorithm

Before

After

Introduction

Social networking platforms have become an important part of our lives and online friendships have now become similarly appealing as offline friendships if not more. People tend to enjoy the companionship of their real-life friends in the virtual world and at the same time, are ready to form new friendships online. But, with increasing online crimes it is hard to guess whom to trust online. The recent surge of research in the friend recommendation algorithms is not surprising. Historically there have been two main recommendation algorithms, content based and collaborative based. Content based algorithm requires textual information as its name suggests and recommends websites, news paper articles, blogs and other contents. Collaborative based algorithms recommends products to a user which it believes has been liked by similar users. Both of these algorithms have yielded unsatisfactory results in friend recommendation systems. In real-life scenarios, people tend to trust strangers who are friends of their friends as they trust in the judgement of their friends. Higher the number of mutual friends between them, higher can be the trust between strangers meeting for the first time. This behavior can be mimicked by the friend suggestion system where strangers can be introduced to each other where they have some mutual friends, making them trustworthy to each other

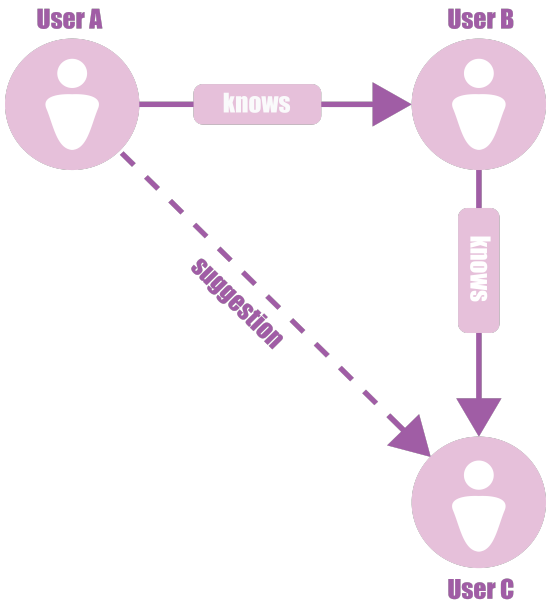
Methodology

The algorithm suggests users to other users based on the number of mutual friends between them. The algorithm uses an undirected graph, where each node is a user and the edges of the graph indicates the relation between them i.e. is there is an edge between two nodes then those two users are friends.

This algorithm has been developed for social networks which prefer bidirectional friendships i.e. if user A is a friend of user B then user B is also a friend of user A unlike some social networks which implement directional graphs, having the follow relationship rather friendship where user A follows user B but user B may or may not

follow user A. The graph used by the algorithm is unweighted which means there are not stronger or weaker friendships in the network and all the friendships are equivalent. The algorithm also doesn't allow self-loops where there can be an edge from and to the same node

The algorithm works on the property of triadic closure. Triadic closure is the property of social networks where if node A is connected with node B and node B is connected with node C, then the nodes A and C will be attracted to each other to form an edge between them. Or we can say that if two nodes have one or more common nodes between them, then they tend to be attracted towards each other to form a bond.



The algorithm closely follows this principle and tries to close triangles in the network. For any user, the algorithm first computes potential friends. To do so, it first finds the current friends on the user, then one by one for each friend of the user it finds the friends of the friend and adds to the acquaintance list. This acquaintance list has a counter for the number of times an acquaintance is appearing in the lists of friends of friends of the user. Every time the algorithm encounters a user in the list of friends of any friend of the user, it first checks if this new person is already in the acquaintance list, if it is then it would increment the counter for that user else if it is not in the acquaintance list, it will add it to the list and set its counter value as one. Once all the friends of the friends of the user have been traversed, then the algorithm starts suggesting the user other users in the decreasing order of the counter values, implying any user in the network with the highest number of mutual friends with the host user will be suggested first, the user with the second highest number of mutual friends would be suggested second and so on.

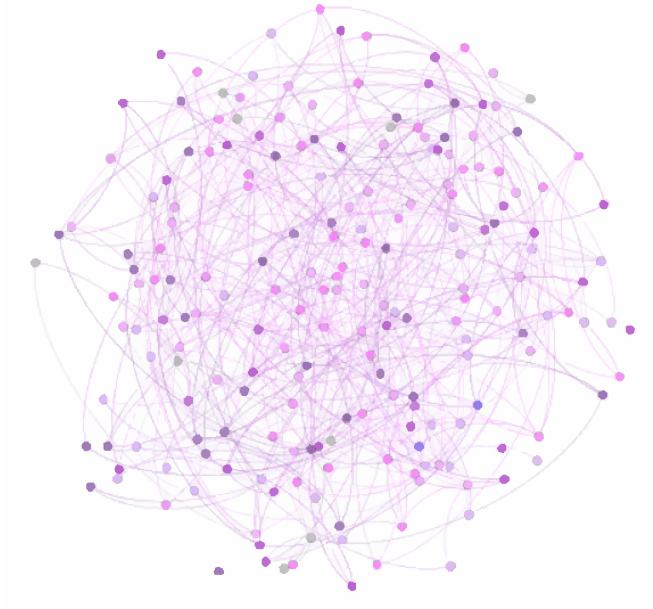
The user which the algorithm is suggesting as the first suggestion for the host user, will close the most number of open triangles in the network. The algorithm is essentially trying to close all possible triangles and will only stop suggesting friends when the user is connected with each and every user in the network. After one computation cycle, if the user connects with all the suggested users then it is not the end of possible suggestions for the user. Every new edge in the network, every new connection mane by any user opens new doors for multiple other users in the network to connect to and discover other new users in the network who are now or were always closer to them in the network.

Results

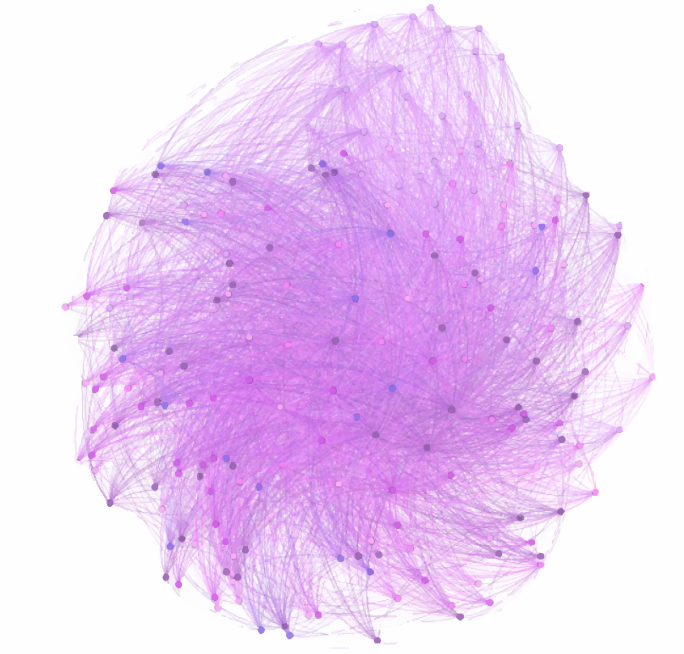
Below are the statistics of 500 runs of the algorithm

	Before	After
Average Degree	39	87
Average Clustering Coefficient	0.0019	0.425
Triangles	1065	937476
Nodes	2000	2000
Edges	40000	90000
Diameter	3	2
Path Length	2.45	1.95

Below is the representation of the graph before and after 500 runs of the algorithm

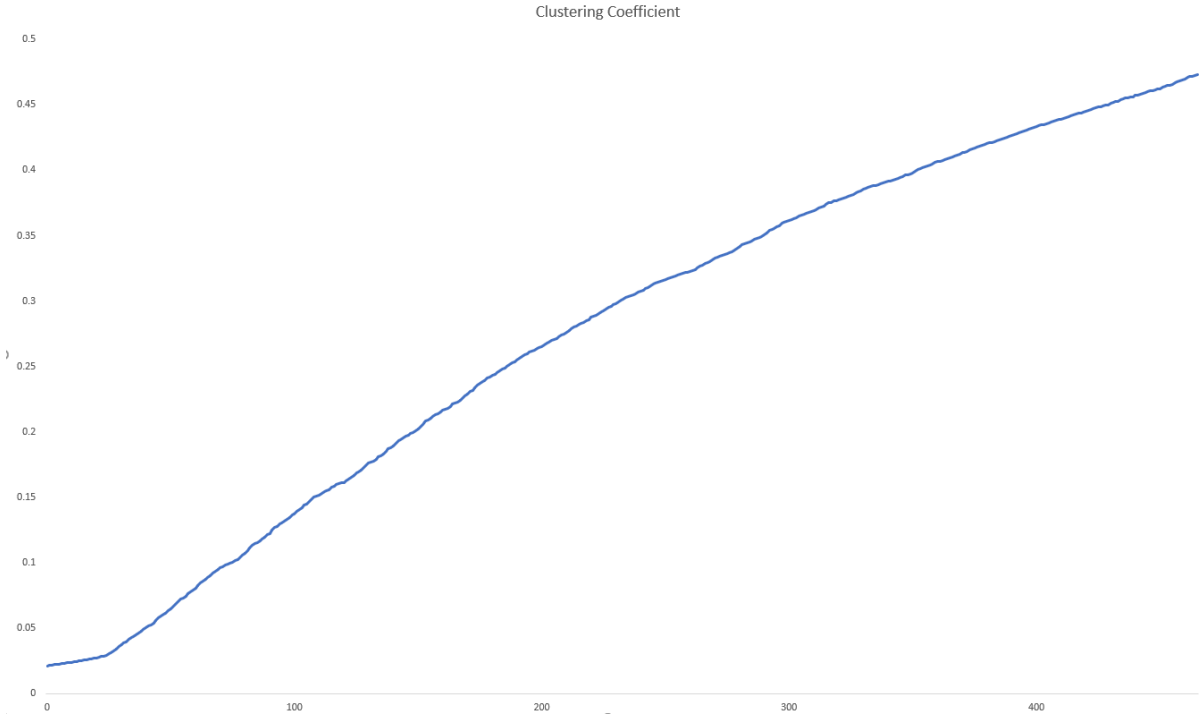


Before
2,000 Nodes
40,000 Edges



After 500 runs
2,000 Nodes
90,000 Edges

Below is the plot of the average clustering coefficient of the social graph after each run



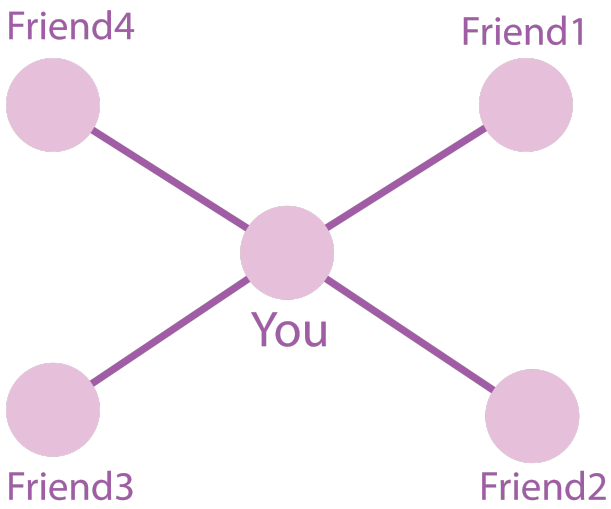
Code Explanation

AddFriends:

We get the connection string to connect to database from config file. We use a SQL Query to get the list of users from our Database. We use our second query to get the list of the friendships from the network.

```
string connString =
System.Configuration.ConfigurationManager.AppSettings["connstring"];
string queryEdges = "Select * from edges";
string queryUsers = "Select * from users";
```

A user is represented by a node in a graph. The edge between the two nodes is represents that they are friends.



Here we write the code to establish a social network, which will be represeneted by a graph. We represent the social network via DataTable

```
DataTable edges = new DataTable();
DataTable users = new DataTable();
```

We use the function *AddFriends()* to select a random person and find another person with most mutual friends between them and add them as friends. Below is the basic structure of *AddFriends()* function.

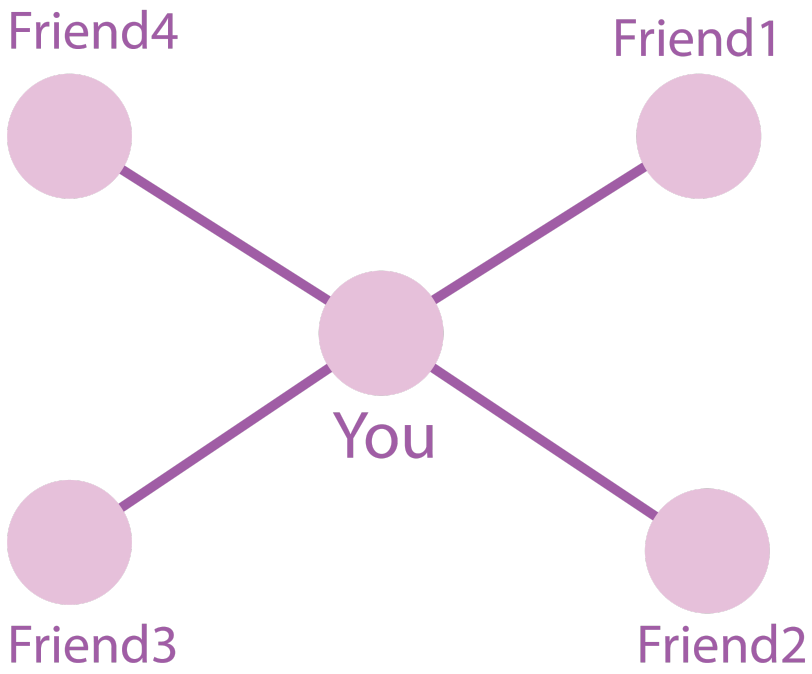
```
// Add Friend function
       private void AddFriends()
        {
                        // We get the connection string to connect to database
from config file.
            string connString =
System.Configuration.ConfigurationManager.AppSettings["connstring"];
            string queryEdges = "Select * from edges";
            string queryUsers = "Select * from users";

                        //friendsmade stores the specified number of friends to
be made.
            int friendsmade = 0;
                        //Index used for Output purposes
            int k = 0;

            string[] output = new string[100];
            Array.Clear(output, 0, 100);

                        DataTable edges = new DataTable();
            DataTable users = new DataTable();
```

The following code executed the above mentioned queries to fetch the data.

```
try
{
    using (SqlConnection conn = new SqlConnection(connString))
    {
        conn.Open();

        SqlDataAdapter da = new SqlDataAdapter(queryEdges, conn);
        da.Fill(edges);

        da = new SqlDataAdapter(queryUsers, conn);
        da.Fill(users);
```

For demonstration purposes we try to simluate a friend being suggested for 100 times. We pick a random user from the network, and for our understanding let's call him **host user**, referred as 'You' in the diagram

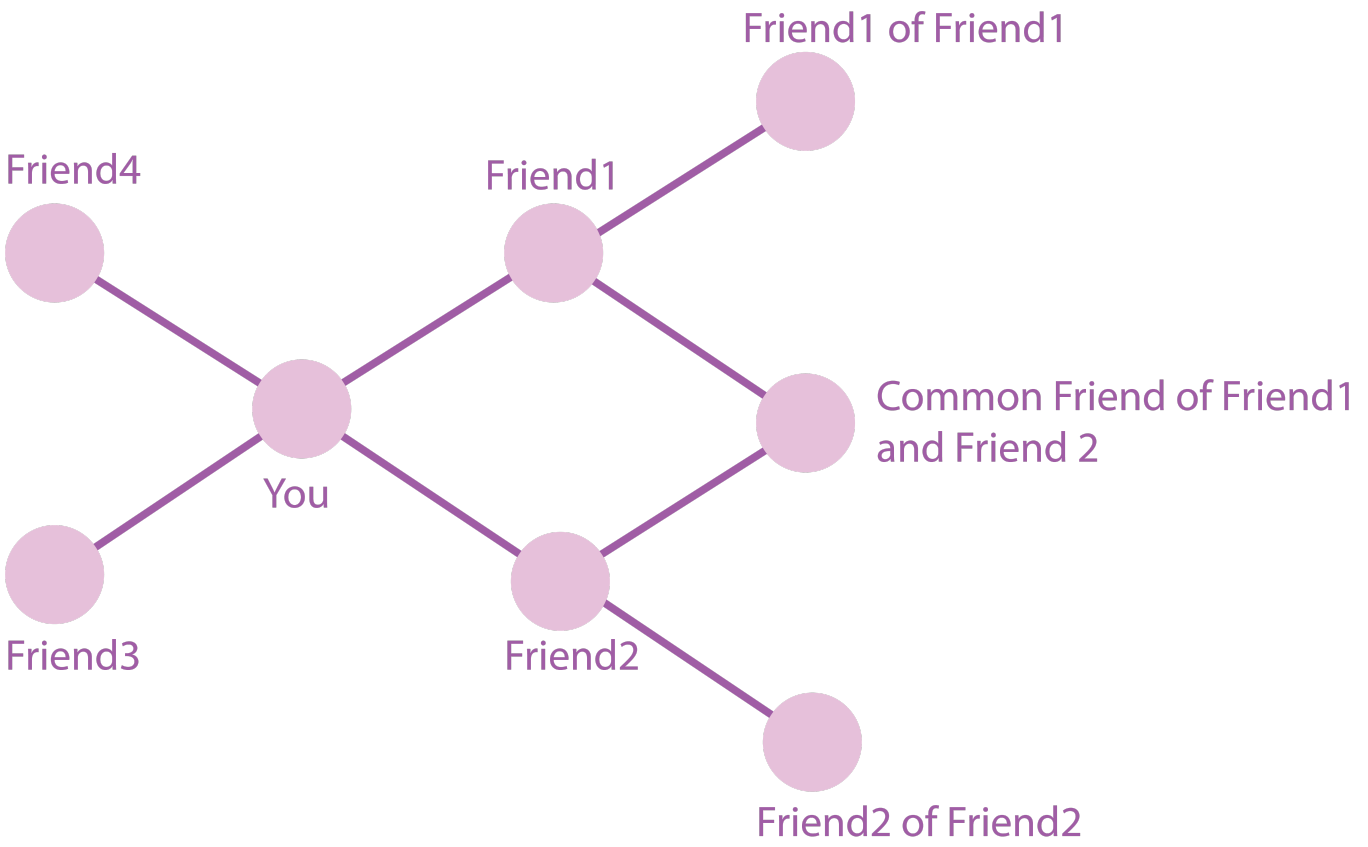
below.
A user's friends of friends can potentially be friends with the host user. We use *potentialFriends* object to store that.



Friends object stores the friends of the host user.

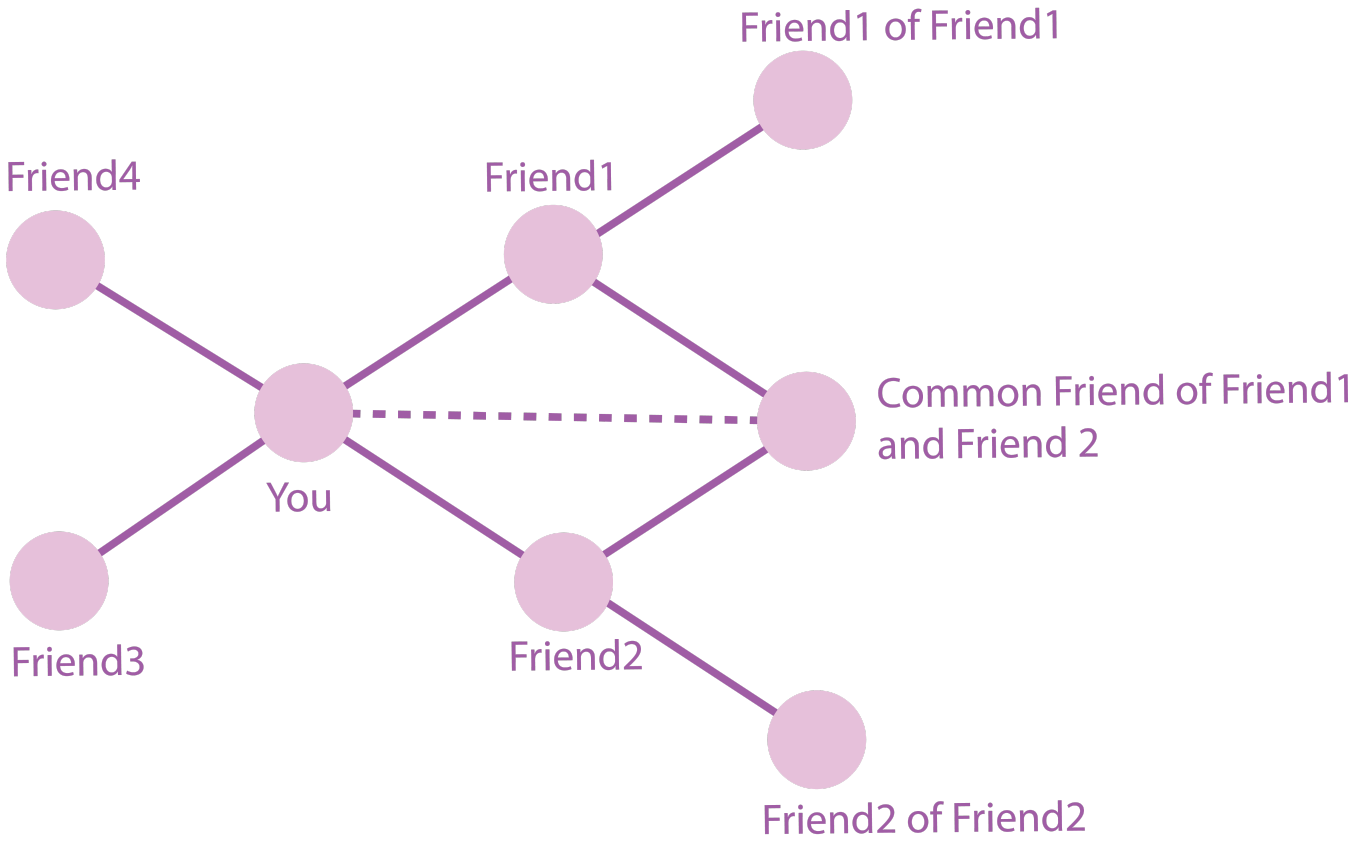
```
while (friendsmade < 100)
{
    Random rnd = new Random();
    int person = rnd.Next(1, 2001);
    DataRow[] friends;
    DataRow[] friendsoffriends;

    Dictionary<int, int> potencialFriends = new Dictionary<int, int>();

    friends = edges.Select("A=" + person);
```

Now we process each friend in the list of host user's friends one by one, let's consider the first friend and call him **friend1**.
Now we get all the friends of friend1, let's call the first friend of friend1 **acquaintance**.
Now we check if the acquaintance and the host users are already friends. If they're not then we add them to *potentialfriends*.
We maintain a count of occurences of this acquaintance in *potentialfriends*.




```
foreach (DataRow row in friends)
    friendsoffriends = edges.Select("A=" + row.ItemArray[1]);

    foreach (DataRow acquaintance in friendsoffriends)
    {
        if ((edges.Select("A=" +person+ "AND B="
+acquaintance.ItemArray[1])).Length > 0)
        {
            continue;
        }
        if
(potencialFriends.ContainsKey(Convert.ToInt32(acquaintance.ItemArray[1])))
        {
            potencialFriends[Convert.ToInt32(acquaintance.ItemArray[1])]++;
        }
        else
        {
            potencialFriends.Add(Convert.ToInt32(acquaintance.ItemArray[1])
        }
    }
}
```

Now we get the Potential Friend with maximum occurence and suggest him to our host user.



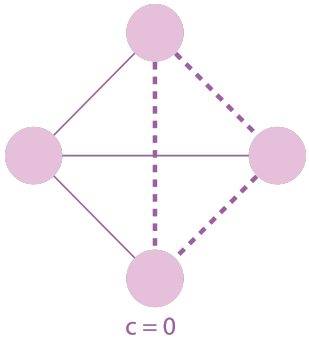

```
potencialFriends = potencialFriends.OrderByDescending(x =>
x.Value).ToDictionary(pair => pair.Key, pair => pair.Value);
```

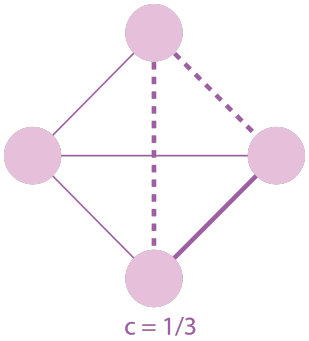
We run the following code to add this new friend edge in database.

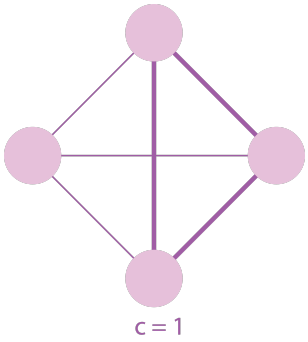
```
string queryNewEdges = "INSERT INTO EDGES VALUES (" + person + ", " +
potencialFriends.First().Key + ")";
SqlCommand command = new SqlCommand(queryNewEdges, conn);

try
{
    friendsmade = friendsmade + command.ExecuteNonQuery();
    output[k++] = users.Rows[users.Rows.IndexOf(users.Select("id=" + person)[0])]
["first_name"]
    + " is now friends with "
    + users.Rows[users.Rows.IndexOf(users.Select("id=" +
potencialFriends.First().Key)[0])]["first_name"]
    + " and completed " + potencialFriends.First().Value
    + " triangles";
    System.Diagnostics.Debug.WriteLine(friendsmade);
}
catch (Exception ex)
{ }
```

Clustering Coefficient:

As explained above, we simply query relevant information from the database.







Clustering coefficient explain here

```
private void CalculateClusteringCoefficient()
{
    string connString =
System.Configuration.ConfigurationManager.AppSettings["connstring"];
    string queryEdges = "Select * from edges";
    string queryUsers = "Select * from users";
    string queryCoefficient = "Select * from clusteringcoefficient";

    DataTable edges = new DataTable();
    DataTable users = new DataTable();
    DataTable chart = new DataTable();

    try
    {
        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();

            SqlDataAdapter da = new SqlDataAdapter(queryEdges, conn);
            da.Fill(edges);

            da = new SqlDataAdapter(queryUsers, conn);
            da.Fill(users);

            DataRow[] friends;
            int maxEdges;
            int common;
            Dictionary<int, double> coefficient = new Dictionary<int, double>();
```

Now we process each user in the network one by one, let's call this user as **host user**.
Now we get all the friends of host user, let's call the first friend as **friend1**.
Now we check if friend1 is friends with all other friends of the host user and keep the count of these friendships.
Now we try to check maximum possible friendship edges amongst all the friends of host users.
We calculate clusterig coefficient for the given host user as

Clustering Coefficient = Count of Friendships / Max Edges

```
foreach (DataRow user in users.Rows)
{
    common = 0;

    friends = edges.Select("A=" + user.ItemArray[0]);
    maxEdges = friends.Length * (friends.Length - 1);

    foreach (DataRow friend in friends)
    {
        foreach (DataRow otherfriend in friends)
        {
            if (otherfriend.ItemArray[1].ToString() !=
friend.ItemArray[1].ToString())
            {
                DataRow[] frienship =
                    edges.Select("A=" + friend.ItemArray[1] + " AND B=" +
otherfriend.ItemArray[1]);
                if (frienship.Length > 0)
                    common++;
            }
        }
    }
    double ClusteringCoefficient =

Convert.ToDouble(common.ToString()+".0")/Convert.ToDouble(maxEdges.ToString()+".0"
);

    coefficient.Add(Convert.ToInt32(user.ItemArray[0]), ClusteringCoefficient);
}
```

We calculate the average clustering coffecient for the entire network. And store it in the database.

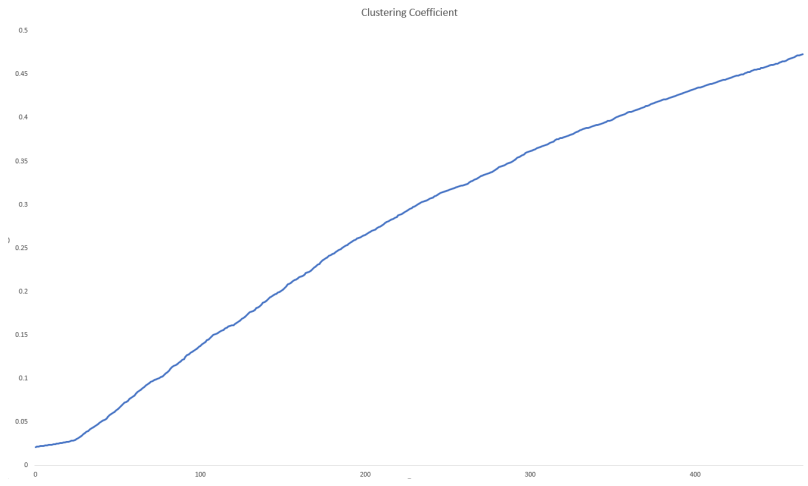
Avg Clustering Coefficient = Sum of Clustering Coefficient / Number of Clustering Coefficient

```
            double avg = coefficient.Values.Sum() / coefficient.Count;
            string queryclustering="INSERT INTO CLUSTERINGCOEFFICIENT VALUES
("+avg+")";
            SqlCommand command = new SqlCommand(queryclustering, conn);
            command.ExecuteNonQuery();
            lbl_clustering.Text = "The clustring coefficient is now " + avg;
            da = new SqlDataAdapter(queryCoefficient, conn);
            da.Fill(chart);
        }
    }
    catch (Exception ex){}
}
```

Load Chart

Now we use the below function to fetch the average clustering coefficient for each run and plot them in a chart.

```
private void LoadChart()
{
    string connString =
System.Configuration.ConfigurationManager.AppSettings["connstring"];
    string queryCoefficient = "Select * from clusteringcoefficient";
    DataTable chart = new DataTable();
    try
    {
        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();
            SqlDataAdapter da = new SqlDataAdapter(queryCoefficient, conn);
            da.Fill(chart);
        }

        ch_clustering.ChartAreas["ChartArea1"].AxisX.MajorGrid.Enabled = false;
        ch_clustering.ChartAreas["ChartArea1"].AxisY.MajorGrid.Enabled = false;

        ch_clustering.DataSource = chart;
        ch_clustering.Series["Series1"].YValueMembers = "Coefficient";
        ch_clustering.DataBind();
    }

    catch (Exception ex)
    { }
}
```



Summary of the project.

This application implements a friends suggestion system based on mutual friends
The algorithm is based on the property of triadic closure where if there is are common nodes between two nodes then the common node tends to attract towards each other completing a triangle. The algorithm fetches the freinds of a node and for each friend node, fetches the friends of friends. It maintains a counter for the number of times a friend of friend appeared which means that particular friend of friend has that many mutual friends with the host node.

This application computes the one friend of friend which has the maximum mutual friends with the host node and adds them as friends. The application selects a node at random and computes as above and adds a new friend. Each function call does the above 100 times for acheiving analyzable results, which can be adjusted as desired.

The application also calculated the average clustering coefficient of the graph by averaging the clustering coefficient of each node. Clustering coefficient is the measure of the degree to which nodes in a graph tend to cluster together.
With each run, the algorithm should increase the clustering coefficient. The database also contains a table for clustering coefficient which stores the coefficient value after each run, where each run adds customizable number of friends, 100 at the time of testing. As a proof of the algorithm working correctly, the clustering coefficient should rise after each run and also after each new friend made.

This application as of now just computes the friends to be matched based on the number of mutual friends. As it visits each friend of the user and each friend of firend of the user, it needs just slight modification to not just count number of occurrences of a friend of friend but number of friends of friends having a particular property. For example, it can find the friend of firend who has the most mutual friends and has the most matching interests like music, sports, etc. It can suggest friends based on maximum number of matching interests like favourite sports team/player, favourite musicians, etc.

For this application, an undirected and unweighted graph has been used. The graph has been stored in a database which contains two tables, users and edges. The users table contains the details of the user. The edges table contains the edges of the graph where an edge between two nodes is represents friendship

Below is the entire code for reference

Back End

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace OwlBook
{

    //Programmed by Nimit Johri | nimit.johri@temple.edu

    public partial class _Default : Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void btn_Run_Click(object sender, EventArgs e)
        {
                AddFriends();
                //CalculateClusteringCoefficient();
                //LoadChart();
        }

        //Function to add friends based on mutual friends
        private void AddFriends()
        {
            string connString =
System.Configuration.ConfigurationManager.AppSettings["connstring"];

            //The graph i used contained 2000 nodes and 40000 edges which fit in
memory
            string queryEdges = "Select * from edges";

            //For larger graph or for memory efficiency, these values can be not
            //prefetched but whenevery required and only for the nodes in scope
            string queryUsers = "Select * from users";

            //Counter to run until the specified number of friends are made
            int friendsmade = 0;

            //counter for indexing output array
            int k = 0;

            string[] output = new string[100];
```

```
            Array.Clear(output, 0, 100);

            //Datatable to store edges
            DataTable edges = new DataTable();

            //Datatable to store users
            DataTable users = new DataTable();

            try
            {
                using (SqlConnection conn = new SqlConnection(connString))
                {
                    conn.Open();

                    SqlDataAdapter da = new SqlDataAdapter(queryEdges, conn);
                    da.Fill(edges);

                    da = new SqlDataAdapter(queryUsers, conn);
                    da.Fill(users);

                    while (friendsmade < 100)
                    {
                        Random rnd = new Random();

                        //select a random person to find another user with most
mutual friends
                        int person = rnd.Next(1, 2001);

                        DataRow[] friends;
                        DataRow[] friendsoffriends;

                        //Dictionary to store potencial friends
                        Dictionary<int, int> potencialFriends = new
Dictionary<int, int>();

                        //Find all the friends of the host node
                        friends = edges.Select("A=" + person);

                        //Loop to process each friend of the host node
                        foreach (DataRow row in friends)
                        {
                            //Find all the friends of the friend of host node
                            friendsoffriends = edges.Select("A=" +
row.ItemArray[1]);

                            //Each friend of friend will be called acquaintence
                            foreach (DataRow acquaintance in friendsoffriends)
                            {
                                //If the acquaintence is already a friend of the
host node then continue
                                if ((edges.Select("A=" +person+ "AND B="
+acquaintance.ItemArray[1])).Length > 0)
                                {
                                    continue;
```

```
                                }

                                //If the dictionary of potencial friends already
                                //contains this acquaintence then increment it'
counter

                                //This signifies that this acquaintence appeared
again
                                //as a friend of another firend of the host node
                                //In other words this counter suggests the number
of mutual friends
                                //between the host node and this node

                                if
(potencialFriends.ContainsKey(Convert.ToInt32(acquaintance.ItemArray[1])))
                                {

potencialFriends[Convert.ToInt32(acquaintance.ItemArray[1])]++;
                                }
                                else
                                {
                                    //If we came across this acquaintence for the
first time
                                    //then add it to the potencial friends
dictionary with counter value 1

potencialFriends.Add(Convert.ToInt32(acquaintance.ItemArray[1]), 1);
                                }
                            }
                        }
                        //At the end of the loop, order the dictionary by the
counter value to find
                        //the acquaintence node with most mutual friends
                        potencialFriends=

potencialFriends.OrderByDescending(x=>x.Value).ToDictionary(pair=>pair.Key,pair=>p
air.Value);

                        //Add the friendship edge between the acquaintence node
and host node,
                        //which had the most mutual friends
                        string queryNewEdges=
                        "INSERT INTO EDGES VALUES (" + person + ", " +
potencialFriends.First().Key + ")";
                        SqlCommand command = new SqlCommand(queryNewEdges, conn);

                        try
                        {
                            //Increment the friends made counter
                            friendsmade = friendsmade + command.ExecuteNonQuery();
                            output[k++] =
                            users.Rows[users.Rows.IndexOf(users.Select("id=" +
person)[0])]["first_name"]
                            + " is now friends with "
```

```
                            + users.Rows[users.Rows.IndexOf(users.Select("id=" +
potencialFriends.First().Key)[0])]
                            ["first_name"]+ " and completed "
                            + potencialFriends.First().Value
                            + " triangles";
                            System.Diagnostics.Debug.WriteLine(friendsmade);
                        }
                        catch (Exception ex)
                        { }


                    }

                    lbl_friendsmade.Text = friendsmade + " new friends made";

                    gv_Display.DataSource = output;
                    gv_Display.DataBind();
                }
            }
            catch (Exception ex)
            {
            }
        }

        protected void btn_Clustering_Click(object sender, EventArgs e)
        {
            CalculateClusteringCoefficient();
        }

        //Method to calculate the average clustering coefficient of the graph
        //by averaging the clustering coefficient of each node
        private void CalculateClusteringCoefficient()
        {
            string connString =
System.Configuration.ConfigurationManager.AppSettings["connstring"];
            string queryEdges = "Select * from edges";
            string queryUsers = "Select * from users";
            string queryCoefficient = "Select * from clusteringcoefficient";

            DataTable edges = new DataTable();
            DataTable users = new DataTable();
            DataTable chart = new DataTable();

            try
            {
                using (SqlConnection conn = new SqlConnection(connString))
                {
                    conn.Open();


                    SqlDataAdapter da = new SqlDataAdapter(queryEdges, conn);
                    da.Fill(edges);

                    da = new SqlDataAdapter(queryUsers, conn);
```

```
                    da.Fill(users);

                    DataRow[] friends;
                    int maxEdges;
                    int common;

                    //Dictionary for clustering coefficient
                    Dictionary<int, double> coefficient = new Dictionary<int,
double>();

                    //Loop to process each node
                    foreach (DataRow user in users.Rows)
                    {
                        common = 0;

                        //Find all friends of this host user
                        friends = edges.Select("A=" + user.ItemArray[0]);

                        //Find maximum number of edges possible between this
user's friends
                        //which is n(n-1) where n is the number of friends of this
host user
                        maxEdges = friends.Length * (friends.Length - 1);

                        //Loop to process each friend of this user
                        foreach (DataRow friend in friends)
                        {
                            //Loop to process each mutual friend between the host
user and this friend user
                            foreach (DataRow otherfriend in friends)
                            {
                                //Check if it is the same friend in both firend
and otherfriend
                                if (otherfriend.ItemArray[1].ToString() !=
friend.ItemArray[1].ToString())
                                {
                                    //Check if this friend is friends with another
friend of the host node
                                    DataRow[] frienship = edges.Select("A=" +
friend.ItemArray[1]
                                    + " AND B=" + otherfriend.ItemArray[1]);

                                    //If friendship found then increment counter
                                    if (frienship.Length > 0)
                                        common++;
                                }
                            }
                        }
                        //Calculate the clustering coefficient for this host user
                        //by dividing the number of edges found between it's
friends and the maximum possible edges
                        double ClusteringCoefficient =
Convert.ToDouble(common.ToString()+".0")
                        /Convert.ToDouble(maxEdges.ToString()+".0");
```

```
                        //Add clustering coefficient for this host node in
dictionary
                        coefficient.Add(Convert.ToInt32(user.ItemArray[0]),
ClusteringCoefficient);


                    }

                    //Calculate the average of the clustering coefficent
                    //of all the clustering coefficient of all the nodes
                    double avg = coefficient.Values.Sum() / coefficient.Count;

                    string queryclustering = "INSERT INTO CLUSTERINGCOEFFICIENT
VALUES (" + avg + ")";
                    SqlCommand command = new SqlCommand(queryclustering, conn);
                    command.ExecuteNonQuery();

                    lbl_clustering.Text = "The clustring coefficient is now " +
avg;


                    da = new SqlDataAdapter(queryCoefficient, conn);
                    da.Fill(chart);

                }
            }
            catch (Exception ex)
            {


            }
        }

        protected void btn_loadchart_Click(object sender, EventArgs e)
        {
            LoadChart();
        }

        //This method is used to fetch the clustering coefficent values
        // from the database and display a chart
        private void LoadChart()
        {
            string connString =
            System.Configuration.ConfigurationManager.AppSettings["connstring"];
            string queryCoefficient = "Select * from clusteringcoefficient";

            DataTable chart = new DataTable();

            try
            {
                using (SqlConnection conn = new SqlConnection(connString))
                {
                    conn.Open();
                    SqlDataAdapter da = new SqlDataAdapter(queryCoefficient,
```

```
conn);
                    da.Fill(chart);
                }

                ch_clustering.ChartAreas["ChartArea1"].AxisX.MajorGrid.Enabled =
false;
                ch_clustering.ChartAreas["ChartArea1"].AxisY.MajorGrid.Enabled =
false;

                ch_clustering.DataSource = chart;
                ch_clustering.Series["Series1"].YValueMembers = "Coefficient";
                ch_clustering.DataBind();
            }

            catch (Exception ex)
            { }

        }
    }
}
```

Front End

```
<%@ Page Title="Home Page" Language="csharp" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="OwlBook._Default" %>

<%@ Register Assembly="System.Web.DataVisualization, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35"
Namespace="System.Web.UI.DataVisualization.Charting" TagPrefix="asp" %>

<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent" runat="server">
        <div class="row">
            <div class="col-md-6">
    <div class="jumbotron">
        <h1>OwlBook</h1>

        <p>
            <asp:Button ID="btn_Run" runat="server" Text="Suggest Friends"
OnClick="btn_Run_Click" /></p>
            <p><asp:Label ID="lbl_friendsmade" runat="server" Text=""></asp:Label>
</p>
            <p><asp:GridView ID="gv_Display" runat="server">
            </asp:GridView>
        </p>
    </div>
        </div>
```

```
        <div class="col-md-6">
            <p>
                <asp:Button ID="btn_loadchart" runat="server"
OnClick="btn_loadchart_Click" Text="Plot Clustering Coefficient" /><asp:Button
ID="btn_Clustering" runat="server" Text="Calculate Clustering Coefficient"
OnClick="btn_Clustering_Click" /></p>
            <p>
                <asp:Label ID="lbl_clustering" runat="server" Text=""></asp:Label>
</p>
            <p>
                <asp:GridView ID="gv_Clustering" runat="server"></asp:GridView>
            </p>
            <p>
                <asp:Chart ID="ch_clustering" runat="server" Height="600px"
Width="600px">
                    <Series>
                        <asp:Series Name="Series1" ChartType="Line"></asp:Series>
                    </Series>
                    <ChartAreas>
                        <asp:ChartArea Name="ChartArea1"></asp:ChartArea>
                    </ChartAreas>
                </asp:Chart>
            </p>
        </div>
    </div>

</asp:Content>
```