# IcesiTinder Enunciado V1.0

Un grupo de ingenieros de sistemas y telemática de la universidad Icesi se preocuparon por la falta de comunicación de su comunidad universitaria, ya que debido a la pandemia del Covid-19 las relaciones, reuniones y competencias de esta se tuvieron que eliminar en vista de que la presencialidad en todas estas actividades no está permitida hasta el momento. Este grupo de ingenieros caleños diseñaron un sistema de Software que permite que los individuos se conozcan por medio de alguien en común. Lo que la comunidad espera que la solución de software brinde las siguientes funcionalidades:

El programa funciona con cuentas personales que se crean al momento del registro de cada usuario, donde se solicita que ingrese información tal como sus nombres completos, número de teléfono, correo electrónico, género, género de interés y facultad a la que pertenece, además se solicita un nombre de usuario que será con el que podrá realizarse funciones tales como búsqueda dado que este es el que lo representa dentro del software y una contraseña para poder ingresar a la cuenta.

Se espera que el Software pueda permitir a los usuarios del programa conocerse con otros usuarios por medio de las relaciones de las personas que sigue, las relaciones pueden ser de conocidos, amigos o mejores amigos y de esto facilitará la manera en cómo se conozcan las personas. Los usuarios que el usuario vaya a conocer se espera que tengan relación de mejores amigos con las personas que este sigue, para que de esta manera sea más cómodo para ambos usuarios.

Además, se espera que cuando una persona crea su cuenta, el programa le muestre de manera aleatoria usuarios que podría seguir que tengan o no intereses en común.

Los usuarios pueden definir qué tipo de relación tienen siempre y cuando se sigan mutuamente, como previamente se mencionó pueden ser de tipo **conocido, amigos o mejores amigos.**

El programa debe realizar operaciones donde se pueda buscar, modificar y eliminar.

*Buscar*

Permite al usuario buscar a otro usuario que se encuentre registrado en el programa por medio del usuario y mostrar la información básica tal como nombres y género únicamente. Si desea, puede seguir el usuario buscado para ver más sobre este. Si el usuario buscado si lo sigue la persona que busca, se le mostrará toda la información de la cuenta de este usuario y podrá dejar de seguirlo.

*Eliminar*

Permite al usuario eliminar de manera definitiva su cuenta del programa, donde para poder hacerlo debe confirmar con su contraseña para que verifique su identidad, al eliminarse la cuenta no podrá recuperarla y perdería toda la información y relaciones que tenga en el programa.

*Modificar*

Permite al usuario cambiar su nombre de usuario por otro que se encuentre disponible o cambiar la contraseña de su cuenta por una nueva, la demás información de la cuenta no es posible cambiarla.

Para manejar el sistema de cuentas, por lo tanto, todos los datos del programa deben ser persistentes (es decir, si se cierra el programa, deben seguir allí una vez se inicie nuevamente).

El programa debe manejar una interfaz gráfica donde el usuario pueda iniciar sesión con su nombre de usuario único y contraseña, en cuento el usuario ingrese a su cuenta la interfaz debe contar con todas las funcionalidades anteriores para que de este modo el usuario pueda hacer uso de cualquiera de estas que desee. Si el usuario no tiene una cuenta aún, la interfaz debe contener una opción de registro donde se le solicite la información necesaria y la cuenta quede en el sistema.

**Functional requirements**

| **Name:** | R. #1. Find a path that takes to a new person who can be met |
|---|---|
| **Summary:** | Find a path that takes to a new person who can be met and this path is the best one |
| Input: | |
| | |
| Results: | Path found |
| | Path not found |

| **Name:** | R. #2. Model the force that unites a person with another |
|---|---|
| **Summary:** | This refers to the fact that they can be Friends, best friends or acquaintances |
| Input: | Force that user chose |
| | |
| Results: | |
| | |

| **Name:** | R. #3. Show all the people that user follows |
|---|---|
| **Summary:** | Show the username of all the people that user follows |
| Input: | |
| | |
| Results: | User follows with username |
| | User doesn't follow anyone |

| **Name:** | R. #4. Show all the user followers |
|---|---|
| **Summary:** | Show the username of all the user followers |
| Input: | |
| | |
| Results: | User followers with username |
| | User doesn't have followers |

| **Name:** | R. #6. Remove an specific follow |
|---|---|
| **Summary:** | User can unfollow users |
| Input: | String username |
| | |
| Results: | |
| | |

| **Name:** | R. #7. Modify user information |
|---|---|
| **Summary:** | Modify user information (username or password) to change the password user need to enter actual password before change it |
| Input: | String newInformation |
| | |
| Results: | Information change correctly |
| | Information could not be change |

| **Name:** | R. #8. Register a new user |
|---|---|
| **Summary:** | Register a new user that doesn't have a account, user have to enter name and last name, gender, faculty, username and password |
| Input: | String name, String lastname, char gender, String faculty, String username, String password |
| | |
| Results: | User account was created correctly |
| | User account could not be created |

| **Name:** | R. #9. Allow sign-in |
|---|---|
| **Summary:** | Allow sign-in in accounts already create if password and username match |
| Input: | String username, String password |
| | |
| Results: | Sign-in correctly |
| | Username or password doesn't match |

| **Name:** | R. #10. Delete account |
|---|---|
| **Summary:** | User can delete his account, this action Will remove all followers and won't be able to sign-in again. To delete account user have to enter password |
| Input: | String password |
| | |

| Results: | Account deleted correctly |
| --- | --- |
| | Password doesn’t matc, account couldn’t be deleted |

**Non-functional requirements**

- Implement a graph (adjacent matrix and list)
- Implement graph methods like Dijkstra, BFS, DFS and others
- Must have a graphical interface

**FASE 1: IDENTIFICACIÓN DEL PROBLEMA**

*Definición del problema*

La comunidad universitaria requiere desarrollo de un Software que facilite la conexión social entre un grupo de personas, dado que actualmente por la pandemia las personas no pueden interactuar presencialmente y por esto las conexiones sociales se han visto disminuidas entre la comunidad universitaria, afectando de esta forma el aura social.

*Identificación de necesidades y síntomas*

- Encontrar la forma más eficiente de conocer a una persona mediante una secuencia de lasos sociales que empieza en el individuo y finaliza en la persona que desea conocer (Dijkstra)
- Verificar si es posible contactar a alguien mediante un grafo de relaciones, es decir, si alguno de mis amigos o conocidos tiene relación con la persona que deseo conocer (BFS)
- Listar las relaciones sociales de una persona por categorías tales como mejores amigos, amigos y conocidos
- Mostrar todos los contactos con la información de cada uno tal como correo, nombre de usuario, nombre de la persona, número telefónico, genero, genero de interés, edad y facultad
- Sugerir personas de manera aleatoria a las personas que apenas se registren y no cuenten con contactos aún
- Buscar entre los contactos una persona y mostrar la información de esta
- Eliminar una relación de una persona con otra persona que estaba dentro de sus relaciones.
- Modificar la información del perfil propio del usuario
- Crear una interfaz de registro e inicio de sesión de los usuarios
- Registrar la cuenta de una nueva persona por los campos requeridos (nombre completo, nombre de usuario, número de teléfono, genero, genero de interés, edad y facultad)
- Eliminar la cuenta definitivamente, esto quiere decir que la persona pierde los contactos que tenía y no podrá volver a ingresar nunca más a esta cuenta.

**FASE 2: RECOPILACIÓN DE LA INFORMACIÓN NECESARIA**

*Requerimientos para la solución del problema*

RF1. Encontrar la secuencia que conlleve a un individuo y que sea la secuencia más prometedora.

RF2. Modelar la fuerza que une a un individuo con otro, esto se refiera a que puede ser conocido, amigo y mejores amigos.

*RF3. M*ostrar todos los contactos con la información de cada uno tal como correo, nombre de usuario, nombre de la persona, número telefónico, genero, genero de interés, edad y facultad.

RF4. Sugerir de manera aleatoria un usuario a los usuarios que recién crean sus cuentas y aún no siguen a nadie.

RF5. Realizar la búsqueda entre los seguidores y mostrar la información del seguidor solicitado

RF6. Permitir que el usuario deje de seguir a otro usuario

RF7. Modificar la información del perfil propio del usuario, tal como contraseña o nombre de usuario

RF8. Registro de un nuevo usuario

RF9. Permitir inicio de sesión de los usuarios ya registrados.

RF10. Eliminar la cuenta definitivamente, esto quiere decir que la persona pierde los contactos que tenía y no podrá volver a ingresar nunca más a esta cuenta eliminada.

*Restricciones para la solución del problema*

- *La solución debe* poseer *una interfaz gráfica de usuario.*
- *El modelamiento de la base de datos de las personas debe estar implementada con una estructura de dato tipo Grafo.*
- *Debe implementarse al menos dos métodos funcionales de la estructura de datos tipo Grafo*
- *Debe implementarse todos los métodos, aunque no sean funcionales de la estructura de datos tipo Grafo*
- *Se debe tener dos implementaciones de las estructuras de datos tipo Grafo, una de lista y matriz de adyacencia*
- *Entre cada par de vértices existe una única arista con peso variable de 1 a 3.*

*Definiciones*

1. Vértice

   Son puntos o nodos con los que están conformado los grafos. Llamaremos grado de un vértice, al número de aristas de las que es extremo. Se le dice vértice “par” o “impar” según sea su grado.

2. Arista

   Una arista es una relación entre dos vértices de un grafo.

3. Grafo

   Conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.

4. Caminos

   En Teoría de Grafos, se llama camino a una secuencia de vértices dentro de un grafo tal que exista una arista entre cada vértice y el siguiente. Se dice que dos vértices están conectados si existe un camino que vaya de uno a otro, de lo contrario estarán desconectados. Dos vértices pueden estar conectados por varios caminos. El número de aristas dentro de un camino es su longitud

5. Dijkstra

   Algoritmo de Dijkstra. También llamado algoritmo de caminos mínimos es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo con pesos en cada arista.

6. Deep first search

   Una Búsqueda en profundidad (en inglés DFS o Depth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir

expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

7. Breadth first search

   En Ciencias de la Computación, Búsqueda en anchura (en inglés BFS - Breadth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

8. Algoritmo de Floyd-Warshall

   En informática, el algoritmo de Floyd-Warshall, descrito en 1959 por Bernard Roy, es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución.

9. Árbol de recubrimiento mínimo

   Dado un grafo conexo y no dirigido, un árbol recubridor, árbol de cobertura o árbol de expansión de ese grafo es un subgrafo que tiene que ser un árbol y contener todos los vértices del grafo inicial. Cada arista tiene asignado un peso proporcional entre ellos, que es un número representativo de algún objeto, distancia, etc.; y se usa para asignar un peso total al árbol recubridor mínimo computando la suma de todos los pesos de las aristas del árbol en cuestión. Un árbol recubridor mínimo o un árbol de expansión mínimo es un árbol recubridor que pesa menos o igual que todos los otros árboles recubridores. Todo grafo tiene un bosque recubridor mínimo.

10. Algoritmo de Prim

    El algoritmo de Prim es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

    En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

11. Algoritmo de Kruskal

    El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).

12. Grado del vértice

    Es el número de aristas que inciden sobre un vértice.

**FASE 3: BÚSQUEDA DE SOLUCIONES CREATIVAS**

Para la generación de ideas se realizó una revisión secuencial a conceptos previamente adquiridos relacionados con las necesidades del problema anteriormente expuesto. Partiendo de lo anterior, las posibles soluciones encontradas se exponen a continuación:

**Implementación estructura de datos**

a. Grafo ponderado
b. Grafo no ponderado
c. LinkedList
d. Árbol n-ario

**Tipo de estructura de datos**

a. Grafo dirigido
b. Grafo no dirigido
c. Multígrafo dirigido
d. Multígrafo no dirigido
e. Ninguno de los anteriores (en caso de que no se elijan los grafos, si no la opción c o d)

**Forma de implementación de estructura de datos**

a. Matriz de adyacencias
b. Lista de adyacencias
c. Ninguna de las anteriores (en caso de que no se elijan los grafos, si no la opción c o d)

**FASE 4: TRANSICIÓN DE LA FORMULACIÓN DE IDEAS A LOS DISEÑOS PRELIMINARES**

En esta fase hemos descartado las peores alternativas que no brindan una solución adecuada a los requerimientos, tales como: LinkedList, árbol n-ario, grado dirigido y multígrafo dirigido.

- **Implementación estructura de datos**

  Analizando a profundidad lo requerido, las estructuras tales como LinkedList y árbol n-ario no son las más eficientes ni nos permiten resolver los requerimientos que se piden en el enunciado.

- **Tipo de estructura de datos**

  En este caso, no seleccionamos la opción ninguno de los anteriores dado que la estructura LinkedList y árbol n-ario se descartaron. Se descarta además el grafo dirigido y multígrafo dirigido dado que el tipo de grafo que mejor se acomoda a nuestros requerimientos son los grafos no dirigidos.

- **Forma de implementación estructura de datos**

  En este caso, no seleccionamos la opción ninguno de los anteriores dado que la estructura LinkedList y árbol n-ario se descartaron. Sin embargo, la opción a y b siguen siendo muy factibles por lo que no se descarta ninguna de las dos.

**FASE 5: EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN**

- **Criterios implementación de estructura de datos**
    a. Permita asignar los tipos de relación entre los seguidores del usuario
    b. Permita encontrar el camino más conveniente a el usuario para conocer a una persona (esto se realiza por medio de los tipos de relación)
- **Criterios tipo de estructura de datos**
    a. Permita modelar de la mejor manera el programa por medio de las funciones seguir (tanto del usuario hacía del seguidor, como del seguidor hacia el usuario, es decir que ambos pueden seguirse mutuamente)
- **Forma de implementación de estructuras de datos**

    No tiene criterios, dado que se decide realizar la implementación de ambos tipos de grafo.

**Selección de alternativas de implementación de estructuras de datos**

| | Criterio a | Criterio b | Total |
|---|---|---|---|
| Alternativa a | 5 | 5 | 10 |
| Alternativa b | 0 | 0 | 0 |

Dado que obtiene el mayor puntaje la alternativa a, descartamos la alternativa b.

**Selección de alternativas de tipo de estructuras de datos**

| | Criterio a | Total |
|---|---|---|
| Alternativa b | 2 | 2 |
| Alternativa d | 5 | 5 |

Dado que obtiene el mayor puntaje la alternativa b, descartamos la alternativa a.

**Webgrafía**


- Definición de vértice tomada de: https://sites.google.com/site/matematicasmoralesgalindo/6-1-elementos-y-caracteristicas-de-los-grafos/6-1-1-componentes-de-un-grafo-vertices-aristas-lazos-valencia
- Definición de arista tomada de: https://sites.google.com/site/matematicasmoralesgalindo/6-1-elementos-y-caracteristicas-de-los-grafos/6-1-1-componentes-de-un-grafo-vertices-aristas-lazos-valencia
- Definición de vértice tomada de: https://sites.google.com/site/matematicasmoralesgalindo/6-1-elementos-y-caracteristicas-de-los-grafos/6-1-1-componentes-de-un-grafo-vertices-aristas-lazos-valencia
- Definición de grado de vértice tomada de: https://sites.google.com/site/matematicasmoralesgalindo/6-1-elementos-y-caracteristicas-de-los-grafos/6-1-1-componentes-de-un-grafo-vertices-aristas-lazos-valencia
- Definición de grafo tomada de: https://es.wikipedia.org/wiki/Grafo
- Definición de camino tomada de: https://es.wikipedia.org/wiki/Camino_%28teor%C3%ADa_de_grafos%29
- Definición de Dijkstra tomada de: https://www.ecured.cu/Algoritmo_de_Dijkstra
- Definición DFS tomada de: https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad

- Definición de BFS tomada de: https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura
- Definición de algoritmo de Floyd-Warshall tomada de: https://es.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall
- Definición de árbol de recubrimiento mínimo tomada de: https://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo
- Definición algoritmo de Prim tomada de: https://es.wikipedia.org/wiki/Algoritmo_de_Prim
- Definición de algoritmo de Kruskal tomada de: https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal

**TAD Graph:**

| **TAD:** | <WeightedGraph> |
|---|---|
| **Description:**<br>A Graph is a non-linear collection containing vertices and edges connecting vertices. This ADT does not specify if the edges are directed, leaving that to an implementation. The edges have non-negative weights. | |
| **Invariantes:** | • 1. Empty graph: number of vertices is 0; number of edges is 0.<br>• 2. Self-loops are not allowed.<br>• 3. Edge weights must be >= 1 |
| **Operaciones Primitivas:** | **Graph**()<br>**addEdge**( Vertex v1 , double w, Vertex v2 )<br>**addEdge**( Vertex v1 , Vertex v2 )<br>**addVertex**( Vertex v )<br>**getEdgeWeight**( Vertex v1 , Vertex v2 )<br>**setEdgeWeight**( Vertex v1 , double newWeight, Vertex v2 )<br>**removeVertex**( Vertex v )<br>**removeEdge**(Vertex v1 , Vertex v2 )<br>**getNeighbors**( Vertex v )<br>**getNumberOfVertices**()<br>**getNumberOfEdges**() |

| **graph**() |
|---|
| "Initialize the WeightedGraph." |
| **Pre-condition:** none<br>**Responsibilities:** initializes the graph attributes.<br>**Post-condition:** number of vertices is 0. number of edges is 0 (1.0). |

| **addEdge**( Vertex v1 , double w, Vertex v2 ) |
| --- |
| "Add edge in the WeightedGraph." |
| **Pre-condition:** v1 and v2 are Vertices in this graph and aren't already connected by an edge; w is >= 0.<br>**Responsibilities:** connect Vertices v1 to v2 with weight w; if this is an undirected graph, this edge also connects v2 to v1.<br>**Post-condition**: an edge connecting v1 and v2 with weight w is added to this Graph.<br>*number of edges* is incremented by 1<br>**Exception:** if v1 or v2 are not in the graph, are already connected by an edge,<br>or w < 0.<br>**Returns:** nothing. |

| **addEdge**( Vertex v1 , Vertex v2 ) |
| --- |
| "Add edge in the WeightedGraph." |
| **Pre-condition:** v1 and v2 are Vertices in this graph and aren't already connected by an Edge.<br>**Responsibilities:** connect Vertices v1 to v2; if this is an undirected graph, this edge also connects v2 to v1.<br>**Post-condition:** an edge connecting v1 and v2 is added to this graph *number of edges* is incremented by 1.<br>**Exception:** if v1 or v2 are not in the graph or are already connected by an edge<br>**Returns:** nothing. |

| **addVertex**( Vertex v ) |
| --- |
| "Add the vertex in WeightedGraph." |
| **Pre-condition:** v is not already in the graph.<br>**Responsibilities:** insert a Vertex into this graph.<br>**Post-condition:** a Vertex is added to this graph *number of vertices* is incremented by 1.<br>**Exception:** if Vertex v is already in this graph.<br>**Returns:** nothing. |

| **getEdgeWeight**( Vertex v1 , Vertex v2 ) |
| --- |
| "show the weight." |
| **Pre-condition:** v1 and v2 are Vertices in this graph and are connected by an Edge.<br>**Responsibilities:** get the weight of the edge connecting Vertices v1 to v2.<br>**Post-condition:** the graph is unchanged.<br>**Exception:** if v1 or v2 are not in the graph or are not connected by an Edge.<br>**Returns:** the weight of the edge connecting v1 to v2. |

| **setEdgeWeight**( Vertex v1 , double newWeight, Vertex v2 ) |
| --- |
| "Change the weight." |
| **Pre-condition:** v1 and v2 are Vertices in this graph and are connected by an edge; newWeight is >= 0.<br>**Responsibilities:** set the weight of the edge connecting Vertices v1 to v2 to newWeight.<br>**Post-condition:** the graph is unchanged.<br>**Exception:** if v1 or v2 are not in the graph, are not connected by an edge, or newWeight < 0.<br>**Returns:** nothing. |

| **removeVertex**( Vertex v ) |
| --- |
| "remove the vertex of WeightedGraph. " |
| **Pre-condition:** v is a Vertex in this graph<br>**Responsibilities:** remove Vertex v from this graph.<br>**Post-condition:** Vertex v is removed from this graph, All edges incident on v are removed<br>*number of vertices* is decremented by 1<br>*number of edges* is decremented by degree( v ).<br>**Exception:** if v is not in this graph.<br>**Returns:** nothing. |

| **removeEdge**(Vertex v1 , Vertex v2 ) |
| --- |
| "remove the edge of WeightedGraph." |
| **Pre-condition:** v1 and v2 are vertices in this graph and an edge exists from v1 to v2.<br>**Responsibilities:** remove from this graph the edge connecting v1 to v2; if this is an undirected graph, there is no longer an edge from v2 to v1. |

**Post-condition:** the edge connecting v1 and v2 is removed from this graph *number of edges* is decremented by 1.
**Exception:** if v1 or v2 are not in this graph, or if no edge from v1 to v2 exists.
**Returns:** nothing.

| **getNeighbors**( Vertex v ) |
| --- |
| "return all terms adjacent to v." |
| **Pre-condition:** v is a Vertex in this graph.<br>**Responsibilities:** get the neighbors of Vertex v from this graph.<br>**Post-condition:** the graph is unchanged.<br>**Exception:** if v is not in this graph.<br>**Returns:** a collection containing the Vertices incident on v. |

| **getNumberOfVertices**() |
| --- |
| "Allows get number of vertex." |
| **Pre-condition:** none.<br>**Responsibilities:** get the number of vertices in this graph.<br>**Post-condition:** the graph is unchanged.<br>**Returns:** the number of vertices in this graph. |

| **getNumberOfEdges**() |
| --- |
| "Allows get number of edges." |
| **Pre-condition:** none.<br>**Responsibilities:** get the number of edges in this graph.<br>**Post-condition:** the graph is unchanged.<br>**Returns:** the number of edges in this graph. |