UMAG Universidad de Magallanes | Facultad de Ingeniería

DevGraph: Simulación de red social

Franco Aguilar[†], **Iván Mansilla**[†], **Diego Sanhueza**[†], **Duvan Figueroa**[†] y **Miguel Maripillan**[†]

[†] *Universidad de Magallanes*

Informe fue compilado el día 15 de diciembre de 2024

Resumen

DevGraph es una simulación de una red social, enfocada en la creación de usuarios, la conexión entre ellos, la visualización y creación de publicaciones, y la posible sugerencia de amistades y publicaciones. Para esto se utilizaron grafos, listas enlazadas, tablas hash, colas de prioridad, algoritmos de similaridad, etc.

Keywords: *C, Grafos, Red Social, Simulación, Estructuras de Datos*

Índice

1. Introducción

Una red social es una aplicación diseñada para conectar a las personas y permitirles compartir información a través de publicaciones. Estas plataformas toman en cuenta los intereses y preferencias de los usuarios, ayudándoles a formar comunidades, establecer amistades y ver contenido relevante basado en sus gustos o conexiones.

DevGraph es una simulación de una red social que se enfoca en la creación de usuarios, la conexión entre ellos, la visualización y creación de publicaciones, y la posibilidad de compartirlas. Además, incluye algoritmos que simulan sugerencias de nuevos amigos y publicaciones relevantes para los usuarios.

Para su implementación en C, se utilizaron algoritmos y estructuras de datos para simular DevGraph como:

- **Estructuras de datos:**
  - **Grafos:** En representación de las conexiones entre usuarios, donde cada persona corresponde a un vértice y cada conexión una arista en un grafo dirigido.
  - **Tablas Hash:** Para una busqueda eficiente de datos en la red social.
  - **Colas de prioridad (Heap):** Para crearon recomendaciones de usuarios y publicaciones en base a nuestros intereses o amigos.
  - **Listas enlazadas:** Para manejar publicaciones como listas enlazadas de nodos.
- **Algoritmos:**
  - **Algoritmo de caminos cortos (Dijkstra):** Para calcular la distancia de un nodo a otro en un grafo y de ésta manera encontrar usuarios cercanos en la red.
  - **Similaridad de Jaccard:** la cual mide el peso del enlace en base a los intereses comunes entre dos usuarios.
- **Funcionalidades adicionales:**
  - Generación de usuarios aleatorios.
  - Almacenamiento y recuperación de datos.
  - Perfil psicológico.

2. Objetivo Principal

El objetivo principal del proyecto es crear una simulación de una red social eficiente y rápida, combinando armónicamente los distintos tipos de estructuras de datos vistos durante el curso de Estructuras de Datos (grafos, listas, hash tables, colas de prioridad, etc.).

2.1. Objetivos secundarios

- Implementar algoritmos de búsqueda eficientes, priorizando el tiempo de respuesta.
- Implementar buenas prácticas de codificación.
- Reforzar habilidades de trabajo en equipo y coordinación de tareas entre pares.
- Reforzar habilidades de investigación y análisis de datos.

3. Planteamiento del desarrollo del proyecto

3.1. Usuarios y Posts

Los usuarios y los posts resultan el elemento más importante del programa, pues prácticamente todo el funcionamiento se centra en ellos, por lo que era fundamental que estos estuvieran constituidos de la forma más correcta y eficiente posible. Ambos elementos de forma automática se optó por crearlos como estructuras, que contengan la información pertinente a cada uno, estructuras que se definieron como tipo de dato con typedef, para así en el código tener una mejor comprensión y orden.

3.1.1. Usuarios

Para definir la información de los usuarios que estarían en su estructura se observaron y analizaron los elementos de distintas redes sociales, de los cuales se concluyó que en su mayoría poseen:

- Nombre
- Contraseña
- Usuario
- Lista de seguidos
- Lista de seguidores
- Posts

Por lo que, estos elementos fueron considerados. Además de esto, a medida de que el proyecto fue avanzando y las necesidades evolucionando, se fueron agregando más variables a esta estructura, tal y como sus intereses.

De la estructura planteada anteriormente resalta la palabra “Lista”. Se sabe que las redes sociales en sí las conexiones entre usuarios funcionan como un grafo, en donde un usuario puede tener enlaces con múltiples personas; por lo tanto, estos enlaces de dicho grafo en este caso serían representados por la lista de seguidos y seguidores. Entonces, la estructura de un usuario pasaría a ser el nodo de un gráfico, y las listas, las listas de adyacencia de este. Con esto claro, se unificaron los usuarios con la estructura de datos de los grafos, que estaba siendo trabajada en paralelo.

Sin embargo, este enfoque trajo como consecuencia un problema importante: la búsqueda. El tener que recorrer el grafo en el peor de los casos en orden O(n) para encontrar un único usuario resulta sumamente ineficiente y costoso, sobre todo si hay un gran flujo de datos. Por lo tanto, para resolver esto se pensó en la estructura de datos que tiene un orden de búsqueda O(1): las tablas hash. Entonces, además de los usuarios encontrarse dentro del grafo, a estos se les ve asignado un ID según su nombre de usuario, ID que representa su posición en una tabla hash global en donde están representados todos los usuarios de la red. Este agregado dio al programa la velocidad de búsqueda buscada eficientemente.

3.1.2. Posts

Los posts (o publicaciones) inmediatamente se asumieron como una lista enlazada, en donde la inserción es al principio de ella para mantener un orden de “más reciente a más antiguo”. Al igual que los usuarios, se investigaron las características de otras redes sociales, de donde resaltó que los posts usualmente suelen tener:

- Fecha
- Contenido del post

Por lo que estos elementos fueron añadidos a su estructura.

Con esto hecho, fue lógico, que para que un post sea relacionado a un usuario en específico, que cada usuario tenga en su estructura su lista de posts propia

3.2. Intereses y Similaridad

Los algoritmos de similaridad son técnicas para verificar que tan iguales son dos conjuntos. En el caso de una red social, esto resulta una característica importante para los algoritmos de recomendación de usuarios y publicaciones, así manteniendo enganchado al usuario con contenido que realmente le interese.

Una característica que fue en cierto sentido difícil de plantear fueron cómo representar los intereses de los usuarios de la forma más eficiente, y que sea fácil de obtener su similaridad con los intereses de otro. Desde un principio se eligió trabajar con el **índice de Jaccard**, el cual básicamente habiendo dos conjuntos A y B, su similaridad estará dada por la intersección entre estos dividida por su unión. O sea:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

Esto es un cálculo matemático bastante fácil, el problema era cómo aplicarlo a un grafo y cómo representar los intereses de tal forma que coincidiera con esto.

Entonces, después de una investigación, se acordó utilizar para los intereses atributos binarios. O sea, que cada usuario tuviese una tabla en donde cada celda representa un interés en específico, almacenando un valor “0” si no tiene el interés o “1” si lo tiene. Este enfoque hizo que el índice de jaccard fuese sumamente fácil de calcular, pues es tan simple como sumar la cantidad de celdas en los que ambos tienen un “1” y dividirlo por la cantidad total de intereses presentes.

usuario 1 / usuario 2	0	1
0	M_{00}	M_{01}
1	M_{10}	M_{11}

Figura 1. Tabla comparativa de intereses entre 2 usuarios

M_{00} es el N° de intereses donde ambos tienen valor 0, M_{10} N° de intereses donde el usuario 1 tiene valor 1 y el usuario 2 0, y así sucesivamente. Por lo tanto, la formula para calcular la similitud entre dos usuarios es:

$$J(U1, U2) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}} \tag{2}$$

Sin embargo, esto de igual forma trajo de duda si se estaba ocupando la memoria eficientemente, ya que se estaría guardando la información de qué interés es cada celda repetidas veces. Por lo tanto, se creó una tabla global de intereses, el cual simplemente almacena qué interés corresponde a cada celda, así los usuarios únicamente almacenando en su tabla un puntero a dicha tabla y el valor que le corresponde.

3.3. Grafos

Un grafo es una estructura de datos utilizada para modelar relaciones entre entidades. En este caso, se utilizaron para representar la red social, donde cada nodo, como se dijo anteriormente, representa un usuario. Los grafos se encuentran presentes como estructura de datos fundamental en practicamente todas las redes sociales, pues cada usuario tiene diversas conexiones salientes y entrantes con otros usuarios. Por lo que su implementación fue considerada desde un principio.

En un principio se tuvo la disyuntiva si para aplicar el grafo utilizar una matriz de adyacencia, o bien una lista de adyacencia, y de hecho lo que fue desarrollado en las primeras versiones del programa fue una matriz de adyacencia. Sin embargo, esto no era eficiente ni correspondiente, ya que presentaba una limitación en cuanto a la cantidad de usuarios como en la cantidad de conexiones que se podrían crear; además, los usuarios habían sido pensados que tuvieran una lista de seguidos y seguidores, que eran listas enlazadas simples. Por lo tanto, se deicidió utilizar listas de adyacencias, que permitían la representación de relaciones entre usuarios de manera más eficiente, y que se adaptaba bien a grafos dispersos.

La utilización de grafos con listas de adyacencia, como se dijo anteriormente, de todas formas trajo algunos problemas, tales como la búsqueda, que fue resuelta con tablas hash, recorrer el grafo, y la busqueda de caminos cortos.

Para recorrer el grafo en su totalidad, se decidió hacer que los usuarios no solo sean nodos de un grafo, sino que también nodos de una lista enlazada. O sea, que al crear un grafo, lo que se crea en realidad es una lista enlazada simple, en donde cada nodo es también un nodo de un grafo con diversas conexiones salientes y entrantes.

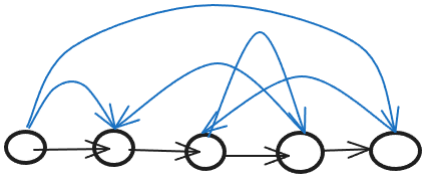

Figura 2. Representación del grafo de usuarios

Por otro lado, para la búsqueda de caminos cortos, se decidió utilizar el algoritmo de Dijkstra, que se utiliza para encontrar el menor camino entre dos nodos en un grafo. Se tuvo la duda si utilizar otros algoritmos como por ejemplo el algoritmo de BFS, pero finalmente fue descartado por una mayor simplicidad.

3.4. Generación de usuarios aleatorios

La generación de usuarios aleatorios es un proceso común en simulaciones y pruebas de software. Al ser este programa una simulación de una red social más que una red social real, este aspecto es importante para comprobar si el programa funciona correctamente.

Se decidió diseñar un algoritmo que generara nombres de usuarios aleatoriamente, de una lista de nombres predefinidas, para que estos tuvieran coherencia. Sin embargo, esto significó la existencia del problema de que si los usuarios no se pueden repetir, entonces habría un máximo de usuarios. Para tratar de mitigar este problema, se optó por agregar un número de 0 a 100 al final de cada usuario, así disminuyendo significativamente la probabilidad de colisión.

Otro aspecto necesario en la generación de usuarios aleatorios era que estos tuviesen intereses y preferencias aleatorias. Esto resulta fácil, ya que es simplemente generar un número aleatorio entre 0 y la cantidad total de intereses, que defina la cantidad de intereses que tendrá el usuario en total, para luego aleatoriamente asignar un interés en cada iteración.

Con los usuarios aleatorios creados, fue importante posteriormente que estos tuviesen conexiones y publicaciones aleatorias. Para las conexiones, se utilizó una lógica similar a la de los intereses, y para las publicaciones aleatorias, se diseñó un algoritmo que creara frases aleatorias según los intereses de los usuarios definidos anteriormente, así manteniendo la coherencia de los usuarios.

3.5. Recomendación de amistades y publicaciones

Para lograr crear recomedaciones de usuario y publicaciones para cada usuario según sus intereses se decidió utilizar colas de prioridad (heaps) para guardar las recomendaciones de amistades y las publicaciones, y ordenarlas de manera que cuando extraigamos un elemento de la cola, este sea el máximo o el mínimo, dependiendo del tipo de cola. Al principio se tuvo la duda si implementar esta estructura de datos con listas enlazadas o con arreglos, pero finalmente después de una investigación, por temas de eficiencia se optó por los arreglos.

Para simular las recomendaciones de amistades, se planteó utilizar Dijkstra o BFS para calcular la distancia de un nodo a otro en un grado y así simular sugerencia de amigos de amigos; finalmente se optó por el algoritmo de Dijkstra como se comentó con anterioridad. Además, se planteó utilizar Jaccard para calcular la similitud entre los usuarios, para poder calcular cuántos usuarios comparten un interes común con otro del total de usuarios.

Para simular las publicaciones, se planteó que primero se mostrarían las publicaciones de los usuarios a los que seguimos, seguidos de estas publicaciones, vendrán aquellas publicaciones recomendadas en base a los usuarios con los que tenemos intereses comunes (jaccard).

3.6. Base de datos

La necesidad de la persistencia de los datos de usuarios en el tiempo hizo que se debiera recurrrir a una forma de almacenamiento como seria una base de datos, la cual recopilaria los datos de los usuarios registrados en la aplicación.

Algo que fue difícil de plantear en un primer intento fue el cómo almacenar la información de los usuarios en el directorio, que fuese eficiente y organizada, pero que no ocupara mucho espacio. Al principio se planteó en utilizar un archivo XML o archivo de texto que almacenara toda la información, pero rápidamente al pensar en un flujo de usuarios grande esta idea fue descartada, ya que el archivo sería demasiado grande, costoso de leer y difícil de implementar su carga. Luego, se pensó en tener una carpeta 'database', en donde cada usuario guardado le correspondiera un archivo de texto con toda su información. Esta idea fue más fácil de implementar, pero costosa de cargar, pues se presentaron problemas en la carga de seguidores, seguidos y publicaciones.

Finalmente, la idea que prosperó fue que cada usuario tuviera su carpeta propia en la carpeta 'database', en este directorio habiendo un archivo para la información de su perfil, otro para sus seguidos, otro para sus seguidores, y una carpeta para sus posts, cada post siendo un archivo de texto con su contenido. Esta idea hizo tanto como la carga como el código mucho más simple y eficiente.

4. Implementación

La implementación del proyecto fue realizada en lenguaje C, utilizando librerías estándar. El programa se ejecuta en un entorno de terminal, donde el usuario debe escribir los comandos necesarios para realizar las acciones deseadas. El código fue escrito de manera modular, o sea, las funciones fueron separadas en diferentes archivos según su funcionalidad. Esto permitió una mayor manera de organización y mantenimiento del código, así como una mejor comprensión del funcionamiento del programa.

4.1. Estructura de directorios

En el directorio principal se encuentran los archivos `Makefile`, con el cual es posible compilar el programa de manera automática, y el archivo `README.MD`, que contiene información sobre el proyecto y su funcionamiento. Además, se encuentra el archivo `subtopics`, que contiene una lista de subtópicos modificable, que se utilizará para el funcionamiento de la aplicación.

Dentro del directorio `src` se encuentran los archivos fuente del programa, que son los siguientes:

- `main.c`: Contiene el flujo principal del programa.
- `database.c`: Contiene las funciones para la gestión y guardado de la base de datos.
- `graph.c`: Contiene las funciones para gestionar los grafos y relaciones entre usuarios.
- `hash_table.c`: Contiene las funciones para gestionar tablas hash.
- `users.c`: Contiene las funciones para gestionar usuarios y publicaciones.
- `utilities.c`: Contiene las funciones de utilidad y testing para el programa.
- `heaps.c`: Contiene las funciones para gestionar las colas de prioridad (heaps) para publicaciones y sugerencias de amistades.

Por otro lado, en el directorio `incs` se encuentran los archivos de cabeceras correspondiente a cada uno de los archivos fuente.

4.2. Usuarios y Posts

4.2.1. Usuarios

Los usuarios se implementaron como los **nodos de un grafo**, donde cada nodo representa un usuario y las aristas indican la relación entre ellos. Las relaciones entre usuarios serían las listas de seguidos y seguidores, que se implementaron como **listas enlazadas simples** que guardan los punteros a los usuarios que están en las respectivas listas, o sea, se utilizaron **listas de adyacencia**. Además estas relaciones son **dirigidas**, es decir, si un usuario A sigue a un usuario B, entonces A será la entrada de B en la lista de seguidores de A y B será la entrada de A en la lista de seguidores de B, y contienen un peso o distancia, el cual representa que tan similares son los usuarios A y B.

Por otro lado, para una búsqueda más eficiente, cada vez que un usuario es creado, se almacena un puntero a este en una tabla hash de usuarios global, hash que es obtenido a partir de su no,bre de usuario, el cual es único y no debe de repetirse en ningún otro usuario. El puntero al usuario se almacena en la celda correspondiente a dicho hash, y, para evitar colisiones, cada celda tiene en ella una lista enlazada, donde es realmente donde se almacenan los punteros de los usuarios que tienen el mismo hash.

4.2.2. Posts

Por otro lado, los posts se implementaron como **listas enlazadas simples**, contenidas en la estructura de cada usuario. Esto permitió una mayor manera de organización y mantenimiento del código, así como una mejor comprensión del funcionamiento del programa. Esta lista posts contiene un nodo centinela, el cual almacena tanto la cantidad de publicaciones que tiene el usuario, como la fecha en la que el usuario fue creado. Además la inserción de una nueva publicación se realiza en el nodo centinela, o sea, al se inserta al principio de la lista, lo que permite que las publicaciones se ordenen de la fecha más reciente a la más antigua.

4.2.3. Intereses

Como se comentó con anterioridad, los intereses de los usuarios fueron almacenados en una tabla binaria, donde cada celda representa un interés en específico, almacenando un valor '0' si no tiene el interés o '1' si lo tiene. Para esto, se utilizó una tabla global de intereses y un archivo de entrada 'subtopics', que contiene una lista de subtópicos modificable. Entonces, cuando el programa se ejecuta, carga este archivo, lee la cantidad de intereses y crea dinámicamente una tabla global de intereses con la misma cantidad de celdas, que luego será la misma cantidad para las tablas de intereses de cada usuario; luego, carga el nombre de cada interés en la tabla global. Como cada usuario en su tabla cada celda tiene un puntero a la celda correspondiente en la tabla global, se ahorra memoria en el proceso de reconocer qué interés tiene cada celda.

Para que un usuario pueda escoger los intereses que este desea tener, debe de realizarlo en el momento en donde se registra, en donde se le preguntará esto. Después estos intereses se almacenan en su tabla de intereses y se guardan en la base de datos. Si un usuario quiere ver los intereses que tiene, se le muestran al momento de visualizar su perfil.

4.3. Índice de similaridad

El índice se similaridad de Jaccard fue implementado como una función que recibe dos usuarios, realiza un cálculo con los intereses de ambos, y devuelve un valor entre 0 y 1. Este valor fue aplicado como el peso de la conexión entre los usuarios, por lo tanto, es calculado cada vez que el programa se ejecuta y se crean las conexiones entre usuarios. Sin embargo, finalmente no se utilizó el índice de jaccard como tal, sino, la **distancia de jaccard**, que en vez de la similaridad, indica que tan diferentes son los usuarios A y B, o sea, que mientras más diferentes son los usuarios, mayor será su distancia.

4.4. Grafos

Los grafos fueron implementados como una lista enlazada simple, en donde cada nodo tiene 2 listas de adyacencia. Cada nodo contiene punteros hacia estas listas, que a su vez permiten mantener un registro de las conexiones asociadas a un usuario determinado. La lista de seguidos gestiona las aristas dirigidas hacia los usuarios que un nodo sigue, mientras que la lista de seguidores contiene las conexiones entrantes, permitiendo representar el grafo dirigido.

Para representar las aristas se creó una estructura `Edge`, que contiene un puntero hacia el usuario destino, así como el peso de la conexión, volviendo el grafo ponderado. Esta estructura es una lista enlazada simple, entonces además contiene un puntero hacia la conexión siguiente.

```
1
2  struct _edge{
3      User dest; /*!< Usuario destino */
4      double weight; /*!< Peso de la arista */
5      Edge next; /*!< Siguiente en la lista de aristas */
6  };
7
8  struct _graph{
9      GraphList graphUsersList; /*!< Lista de usuarios en el grafo */
10     int usersNumber; /*!< Numero de usuarios en el grafo */
11 };
```

Código 1. Estructura que representa una arista y un grafo

Esta forma de implementación es ideal para grafos dirigidos y ponderados. Usar listas enlazadas para las conexiones hace que el uso de memoria sea eficiente, especialmente en grafos dispersos donde no todos los nodos están conectados entre sí. Además, este diseño hace que agregar o eliminar nodos y conexiones sea sencillo, sin necesidad de realizar cambios complicados en la estructura del grafo. Esto la hace una solución práctica y funcional para aplicaciones donde la estructura del grafo puede cambiar con frecuencia.

4.5. Colas de prioridad (heaps)

La cola de prioridad, como se dijo anteriormente, es una estructura de datos que se utilizó para almacenar sugerencias de amistades con respecto a sus intereses y amigos de amigos, y publicaciones en base a nuestros intereses y/o amigos. Esta fue implementada con un arreglo.

Para las sugerencias de amigos de amigos se utilizó el algoritmo de Dijkstra para calcular la distancia de un nodo a otro en un grafo, y de ésta manera lograr encontrar usuarios cercanos en la red con amigos en común. Una vez calculada la distancia, ésta se utilizará como prioridad para la sugerencia de amistad, para que los usuarios con amigos en común se ordenen de menor a mayor distancia.

Para la sugerencia de amistades según los intereses de los usuarios, se utilizó el algoritmo de similaridad de Jaccard, que mide la proporción entre la intersección y la unión de los conjuntos de intereses o conexiones compartidas entre dos usuarios. Este algoritmo se utilizó para calcular la similitud entre los usuarios donde mientras menor sea el resultado, más similar serán sus intereses.

Para la visualización de publicaciones se considerarán prioridad las publicaciones de nuestros seguidos, y luego de éstas las publicaciones de aquellos usuarios con los que tengamos intereses comunes calculado previamente con jaccard como lo hicimos con la sugerencia de amistades según los intereses.

Esta cola de prioridad (heap), es una estructura de datos que organiza elementos de manera que siempre se pueda tener acceso rápido al elemento de menor o mayor prioridad, dependiendo del tipo de heap (mínimo o máximo). En nuestro caso trabajamos con el máximo, es decir que el elemento con mayor prioridad será el primero en la cola (se encuentra en la raiz). Para mantener las propiedades de ésta estructura, se implementaron funciones para insertar, extraer máximos y mínimos, funciones para ordenar la cola de prioridad.

Las principales funciones que fueron implementadas para mantener estas propiedades de la cola de prioridad (heap) fueron las de heapify_up y heapify_down.

- **heapify_up:** Esta función ordena la cola prioridad de manera que, al insertar un nuevo elemento al final de la cola, este es comparado con su padre, intercambiándolos si la prioridad del nuevo elemento es mayor. Esto se repite hasta que se cumpla la propiedad o el elemento llegue a la raiz de la cola.
- **heapify_down:** Esta función ordena la cola de prioridad de manera que, después de eliminar o extraer el máximo elemento de la cola, el elemento al final de la cola es ubicado en la raiz de la cola y se compara con sus hijos, se intercambia si la prioridad de del hijo mayor es mayor a la del elemento en la raiz (padre). Esto continúa hasta que se cumpla la propiedad o el elemento no tenga más hijos.

4.6. Base de Datos

Para la base de datos se hicieron 3 funciones principales:

- **load_all_users:** Esta función carga los perfiles de los usuarios, así como sus publicanes
- **load_connections:** Esta función carga las conexiones entre los usuarios
- **save_database:** Esta función guarda toda la información de los usuarios de la base de datos, así como sus conexiones y publicaciones.

Las funciones de carga son ejecutadas en cada llamada del programa, si es que es necesario en el comando que se le pase el programa. Esto lo hace recorriendo primero cada carpeta del directorio 'database' (que como se dijo anteriormente, cada directorio dentro de él representa un usuario), y se guarda el perfil del usuario que está en el archivo 'data.dat' que tiene cada uno. Una vez todos los perfiles guardados, las conexiones son cargadas, ya que no es posible cargar las conexiones sin antes los usuarios cargados.

Para el guardado de la base de datos, se utiliza una lógica inversa, recorriendo cada usuario del grafo y guardando su perfil en los archivos correspondientes mencionados anteriormente.

En resumen, cada usuario tiene un directorio con la siguiente estructura:

- **data.dat:** Contiene el perfil del usuario, que incluye su ID, nombre, contraseña, categoría, y su lista de intereses.
- **following.dat:** Contiene una lista de usuarios que sigue al usuario actual.
- **followers.dat:** Contiene una lista de usuarios que el usuario actual sigue.
- **posts:** Carpeta con los posts del usuario actual.

4.7. Tablas Hash

Las tablas hash, como se dijo anteriormente, fueron utilizadas para la búsqueda eficiente de usuarios en el grafo, con orden $O(1)$, por lo que esto es un aspecto fundamental para la rapidez del programa.

La estructura `HashNode` fue creada para organizar y manejar los datos en una tabla hash. Esta es una lista enlazada así manejando los casos de colisiones. Un aspecto a importante a destacar es que esta estructura almacena un dato de tipo `void*`, lo que permite almacenar datos de cualquier tipo, como usuarios, publicaciones, intereses, etc.

```

  struct _hashnode {
    char *key;
    void *data;
    Hashnode *next;
};

struct _hashtable {
    Hashnode *buckets[HASH_TABLE_SIZE];
};
```

Código 2. Estructura que representa una celda de una tabla hash

5. Gestión del trabajo

El equipo de trabajo constó de 5 personas, uno de ellos designado por el profesor como líder del grupo, con la responsabilidad de coordinar el trabajo y la gestión del proyecto. Debido a la cantidad de personas involucradas, fue fundamental establecer mecanismos de comunicación, coordinación y orden para asegurar la colaboración y la coordinación de los diferentes miembros del equipo. Algunos de estos mecanismos fueron:

sectionNormas de codificación Las normas de codificación fueron establecidas para el proyecto, con el objetivo de garantizar la coherencia y la consistencia en la codificación. Estas normas incluían:

- **Constantes**: Para las constantes se utilizó la convención de `SCREAMING_SNAKE_CASE`, con el objetivo de diferenciarlas de las variables y funciones.
- **Variables:** Para las variables comunes se utilizó la convención `camelCase`
- **Funciones:** Para las funciones se utilizó la convención `snake_case`, así diferenciándose de las variables y constantes.
- **Tipos de datos:** Para los tipos de datos creados con `typedef` se utilizó la convención `PascalCase`, con el objetivo de diferenciarlos de los tipos de datos nativos de C.
- **Llaves de apertura:** Se estableció la convención de que las llaves de apertura se escribieran en la misma línea que el código que las contiene

```
#define MAX_CHAR 256 // Constante
typedef _User *User; // Tipo de dato
Graph newGraph; // Variable
void print_all_users(Graph graph); // Funcion
```

Código 3. Ejemplo de código, utilizando las normas de codificación

5.1. Comunicación

Un aspecto de gran importancia para el proyecto es la comunicación entre los miembros del equipo, ya que sin ella, no sería posible garantizar la colaboración y la coordinación en el desarrollo del proyecto. Debido al tamaño del equipo y la complejidad del proyecto esto fue un desafío, debido a las diferencias en horarios y responsabilidades académicas de cada miembro. Para resolver este problema fue escencial establecer un medio de comunicación remoto, que permitiera a cada uno comunicarse de manera rápida y en el lugar en que estuviese.

Se creó un chat en línea con cada integrante del equipo, el cual se utilizó para discutir sobre el desarrollo del proyecto y su organización, así como para compartir información y recursos.

Esto resultó exitoso, ya que las reuniones presenciales resultaron ser extremadamente dificiles de coordinar. La comunicación remota nos permitió a los miembros ver y discutir los detalles del proyecto en el momento del día en que estuviesen disponibles. Sin embargo, esto también implicó consecuencias, ya que dio paso a que fuera fácil que algún miembro del equipo "desapareciese"; sin embargo, estas situaciones fueron manejadas a través de la comunicación por privado con éste para consultar su situación.

5.2. División del trabajo

Al inicio del proyecto se decidió establecer **objetivos a corto plazo** para cada miembro del equipo, según las estructuras de datos que se deseaban implementar en el proyecto. A cada uno se le asignaron según afinidad, capacidad y gusto propio.

Una vez las estructuras de datos hechas, se decidió crear una **lista de tareas** a realizar, cada miembro con al menos una tarea a realizar. Esta lista fue creada en base a las ideas discutidas por los integrantes del equipo, y se fue actualizando constantemente en función de las necesidades del proyecto.

A continuación, se presenta una descripción detallada de las tareas asignadas y realizadas por cada miembro del equipo:

- **Duvan Figueroa:**
  Duvan se encargó de la implementación de las **tablas hash** que permiten el almacenamiento eficiente de datos en la red social. Además, trabajó en la **organización de publicaciones**, gestionando cómo se almacenan y visualizan las publicaciones en la plataforma.
- **Diego Sanhueza:**
  Diego fue responsable de la implementación de la **estructura de grafos** en conjunto con Miguel, lo que facilita la representación de las conexiones entre usuarios. También desarrolló el **algoritmo de Dijkstra** para encontrar usuarios cercanos basados en la ubicación o afinidades. Además, trabajó en la creación de perfiles psicológicos de los usuarios y en la implementación de **usuarios aleatorios** para probar el sistema.
- **Iván Mansilla:**
  Iván implementó las **listas enlazadas** para manejar las publicaciones, así como la creación de **estructuras de usuarios**, las cuales incluyen la lista enlazada de publicaciones. También se encargó de agregar **temas/tópicos** en la estructura de usuarios y publicaciones, y de implementar el índice **Jaccard** para medir las similitudes entre usuarios y publicaciones. Además, fue responsable de la asignación de intereses a los usuarios y de la implementación de funcionalidades para **iniciar sesión** y **publicar** según el usuario iniciado. Por último, implementó los comandos para publicar, borrar, ver usuarios, perfil, publicaciones, entre otros.
- **Franco Aguilar:**
  Franco fue responsable de la implementación de **colas de prioridad**, que permiten gestionar las tareas de manera eficiente y priorizar las acciones dentro de la red social.
  como lo son las sugerencias de amistades según interes o

- **Miguel Maripillan:**
  ami Miguel trabajó en conjunto con Diego en la implementación de la **estructura de grafos** y los **algoritmos BFS/DFS** para explogos de amigos y la creación y sugerencia de publicaciones en base a sus intereses y/o usuarios seguidos por el usuario actual.rar las conexiones entre usuarios de manera eficiente. Además, se encargó del **almacenamiento de la información de los usuarios** en la nube y de la **carga de usuarios** a través de un archivo `txt` de entrada.

6. Posibles mejoras

En este punto se presentarán las posibles mejoras que se podrían realizar en este programa, las cuales pueden ser varias dado el tiempo y complejidad asignado a este proyecto.

- Mejoras en la interacción con el usuario: al visualizar las publicaciones recomendadas y sugerencias de amigos se podría incluir una función para guardar publicaciones que le gustaron al usuario y además agregar una opción para seguir a otro usuario.
- Inclusión de más topicos: En este proyecto para los intereses se utilizaron intereses comunes con respecto a la programación, pero existen muchos otros temas de interes generales que podrían ser agregados para simular una red social más amplia.
- Implementación de un sistema de mensajería: En este proyecto se utilizó una interfaz de línea de comandos para interactuar con el programa, pero se podría implementar un sistema de mensajería para permitir a los usuarios comunicarse entre sí y con otros usuarios.
- La creación de grupos o comunidades: Para este tipo de programas es posible crear grupos o comunidades en la red social con respecto a un tema o usuario en particular, para que los usuarios puedan conectarse con otros usuarios de la misma comunidad.
- Búsqueda de publicaciones eficiente: Se pudo haber implementado una tabla hash para las publicaciones de cada usuario, así este pudiendo por ejemplo, editar o borrar una publicación sin un costo grande en tiempo de ejecución.
- Se pudo haber mejorado los tiempos de respuesta de la aplicación, ya que cuando se tiene un flujo grande de usuarios (más de 50.000) el programa tarda mucho en cargar los datos de los usuarios y publicaciones, lo que afecta la experiencia de usuario. Probablemente el algoritmo de la carga de datos se puede mejorar.

7. Ejemplo de uso

Para utilizar este programa es sencillo, en el terminal se debe escribir el comando `make` para compilar el programa, y posterior a esto se debe escribir el comando `./build/devgraph.out` o `./devgraph.out` en el caso de haber ingresado a la carpeta build, asegurandose de tener el documento de entrada `subtopics` en la misma carpeta.

Para comenzar a utilizar el programa, podemos acompañar a nuestro ejecutable con un `-h` o `-help` para ver las opciones de uso del programa.

Si se desea empezar a utilizar las diferentes opciones que otorga el menu de ayuda, se debe partir primero por la opción `-g <cantidad>` o `-generate <cantidad>`, que permite generar la cantidad de usuarios que se deseen, a su vez preguntando si se quieren generar publicaciones para cada uno de ellos.

Luego de generar los usuarios, sin haber iniciado sesión, se pueden utilizar comandos como `-a` o `-all` que lista los usuarios registrados y `-c` o `-clear` para vaciar/borrar la base de datos.

Para utilizar los comandos que requiere de inicio de sesión, puedes utilizar los datos de usuarios generados, y luego utilizar el comando `-l <usuario>` o `-login <usuario>` para iniciar sesión, o bien `-r` o `-register` para registrarse con sus propios datos y luego iniciar sesión.

Una vez iniciado sesión se pueden empezar a utilizar los siguientes comandos:

- `-o` o `-logout`: cierra sesión del usuario actual.
- `-d` o `-delete`: borra el usuario actual.
- `-e` o `-edit`: permiso para editar el perfil del usuario actual.
- `-p` o `-post`: postea una publicación.
- `-m` o `-me`: muestra el perfil del usuario actual.
- `-u <usuario>` o `-user <usuario>`: muestra el perfil del usuario entregado.
- `-f <usuario>` o `-follow <usuario>`: seguir al usuario otorgado.
- `-n <usuario>` o `-unfollow <usuario>`: dejar de seguir al usuario otorgado.
- `-w` o `-followerlist`: imprime la lista de usuarios que sigue al usuario actual.
- `-v` o `-followlist`: imprime la lista de usuarios que sigue el usuario actual.
- `-q` o `-feed`: muestra publicaciones de los usuarios que sigue en base a sus seguidos e intereses.
- `-x` o `-connect`: muestra sugerencias de amigos de amigos y además sugiere amigos según los intereses de los usuarios.

Nota: No olvidar que para utilizar el programa se debe ejecutar de la siguiente manera: `./build/devgraph.out <COMANDO>` o `./devgraph.out <COMANDO>` en el caso de haber ingresado a la carpeta build.

8. Conclusiones

En este proyecto se logro implementar con creces una red social simulada que cumplió con todas las funcionalidades requeridas. Logramos simular varias de las funcionalidades de una red social, como la creación, edición y conexión de usuarios, gestionar publicaciones y compartir contenido en función de los intereses y relaciones de amistad, además de incorporar características avanzadas como la recomendación de nuevos usuarios y publicaciones, la generación aleatoria de diversos elementos rotundamente necesarios en una red social, lo cual añadió una mejoría en lo que respecta a la experiencia ofrecida por el programa.

En su gran mayoría se puede destacar la base de datos, las colas de prioridad, los grafos y las funciones para el usuario que fueron desarrolladas de manera significativa que nos impulsó a lograr nuestros objetivos, utilizando diversas estructuras de datos y algoritmos esenciales para mantener la estabilidad del programa, como tablas hash, las ya mencionada colas de prioridad (heaps), listas enlazadas y listas de adyacencia. También aplicamos el algoritmo de similaridad de jaccard y el algoritmo de Dijkstra para optimizar las recomendaciones y búsquedas de usuarios y/o publicaciones.

Durante el desarrollo del proyecto, a pesar de la dificultad e inconvenientes sufridos, se logró salir adelante con este proyecto con un gran resultado a nuestro parecer, logrando en su mayoría los objetivos planteados al inicio de este trabajo y logramos aplicar de manera práctica conceptos visto en clase y a su vez conceptos investigados por nuestra propia cuenta, fortaleciendo así nuestras habilidades en programación.

9. Referencias

- **Pasky**. Funciones opendir, readdir y closedir en C (2009). Disponible en: https://pasky.wordpress.com/2009/08/05/funciones-opendir-readdir-y-closedir-en-c/. Consultado el 30 de noviembre de 2024.
- **R. Pasniuk**. Directorios y ficheros en C (2013). Disponible en: https://www.programacion.com.py/escritorio/c/directorios-y-ficheros-en-c-linux. Consultado el 5 de diciembre de 2024.
- **OdeN**. Arboles de HEAP y colas de prioridad (2020). Disponible en: https://www.youtube.com/watch?v=udZcgpg-Ss0. Consultado el 2 de diciembre de 2024.
- **GRAPHeverywhere**. Algoritmo de similaridad de Jaccard (202X). Disponible en: https://www.grapheverywhere.com/algoritmo-de-similaridad-de-jaccard/. Consultado el 2 de diciembre de 2024.
- **Estefania Cassingena Navone**. Algoritmo de la ruta mas corta de Dijkstra - Introduccion grafica y detallada (2022). Disponible en: https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/. Consultado el 3 de diciembre de 2024.