**UNIVERSIDAD SIMÓN BOLÍVAR**

**DEPARTAMENTO DE COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN**

**CI2693 – LABORATORIO DE LENGUAJES DE PROGRAMACIÓN I**

**Trimestre:** Abril - Julio 2022

**Profesor:** Fernando Torre

**Informe del Proyecto #3**

**Estudiantes:**

José González 15-10627

Ana Santos. 17-10602

Sartenejas, Julio del 2022

**Introducción**

1. **Decisiones de diseño:**
   1. Diseño general de la solución:

Para este problema se decidió usar como base los grafos no dirigidos y aristas y la implementación recursiva de *búsqueda en profundidad (DFS)*. La idea del diseño del programa está basada en usar clases para definir el grafo y sus componentes, como arista y nodo, y DFS. Por otra parte, usar las funciones de las clases para obtener la información necesario del grafo, como las listas de adyacencias, la lista de nodos y la lista de aristas; la información necesaria de cada arista, como los nodos que la componen; la información necesaria de cada nodo, como su identificador y la información de este que el usuario desea guardar, y finalmente, las funciones recursivas para realizar la búsqueda deseada y para encontrar la información de un nodo.

Por último, se decidió realizar toda la interacción con el cliente, como todas las operaciones relacionadas a la información aportada por el mismo, en el archivo principal que contiene al programa.

* 1. Organización del código y representaciones escogidas para el grafo.

El código se organizó en 5 archivos, *main.rb, nodo.rb, arista.rb, grafoNoDirigido.rb* y *DFS.rb.* Como se indicó anteriormente, se decidió implementar tanto las estructuras grafo, nodo y arista como DFS en clases, aprovechando la amigabilidad de estas y sus funciones para guardar, representar y encontrar la información deseada. Las representaciones escogidas para cada estructura:

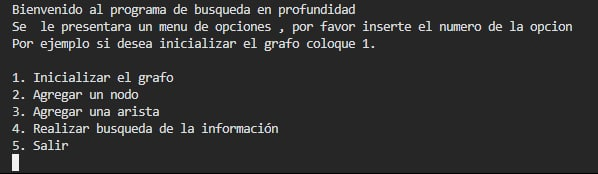
* + 1. **Nodo:**Esta clase fue implementada como una especie de campo, tomando como entrada un String que representa la información aportada por el usuario y asignándole un identificador. Este identificador aumentará cada vez que se haga una instancia de la clase Nodo. Esta clase cuenta con las funciones:
* **get\_X:** retorna el identificador del nodo.
* **get\_Info*:*** retorna la información del nodo.
* **to\_S:** retorna el nodo en forma de String.
  + 1. **Arista:**Esta clase fue implementada para representar un par no ordenado de nodos, ya que los lados de un grafo no dirigido no cuentan con una dirección al conectarse. La clase toma como entrada dos nodos e inicializa sus identificadores por se parado para poderlos usar en las siguientes funciones:
* **cualquieraDeLosNodos:** retorna el identificador del nodo que ha sido ingresado primero (por ninguna razón en especial).
* **elOtroNodo:**toma como entrada el identificador de uno de los nodos pertenecientes a la arista y retorna el identificador del otro nodo.
* **to\_S:** retorna la arista como un Strings, dando a conocer los nodos con sus identificadores e informaciones respetivas.
  + 1. **GrafoNoDirigido:** Esta clase fue implementada para representar un grafo con listas de adyacencias a partir de un arreglo de listas de aristas. Por ello, cada nodo cuenta con un identificador que representa el índice donde se encontrarán todas las listas de aristas relacionadas a ese nodo. Además, la clase cuenta auxiliarmente con un arreglo para todos los nodos y un arreglo para todas las aristas. La entrada de esta clase es un Entero que representa la cantidad de nodos iniciales que va a introducir el usuario. Las funciones de la clase son:
* **agregarNodo:** toma como entrada un nodo y si este no esta incluido en la lista de nodos, entonces añade el nodo a la lista de nodos en la posición de su identificador y en la misma posición, añade una lista vacía en el arreglo del grafo. Retorna un String en caso de que el nodo se encuentre ya dentro de la lista de nodos, indicando que el nodo ya existe.
* **get\_Nodos:** retorna el arreglo de nodos.
* **get\_Grafo:**retorna el arreglo de listas que representa al grafo.
* **get\_Nodo:**toma como entrada el identificador del nodo y retorna el nodo completo.
* **get\_Aristas:**retorna el arreglo que contiene a todas las aristas del grafo.
* **agregarArista:**toma como entrada una arista y verifica que esta no pertenezca al arreglo de aristas para añadirla en las posiciones del arreglo según los identificadores de sus respectivos nodos y añadirla en el arreglo que contiene a todas las aristas. Retorna un String indicando que la arista se encuentra en el grafo, si este es el caso antes de hacer la adición antes mencionada.
* **adyacentes:** toma como entrada un nodo y retorna la lista de aristas guardades en la posición del arreglo según el identificador del nodo.
  + 1. **DFS:** Esta clase toma como entrada el grafo y la información que se desea buscar en el mismo. Para su implementación hace uso de un arreglo de predecesores, un arreglo de sucesores, un arreglo que indica el color y/o estado de los nodos según las visitas, un arreglo de tiempo inicial, un arreglo de tiempo final y una variable para guardar el nodo que se está buscando. Se uso esta implementación recursiva debido a que ha sido previamente estudiada con amplitud por los programadores de este proyecto. Esta clase cuenta con las siguientes funciones para la correcta implementación de búsqueda en profundidad:
* **dfs\_Base:** esta función itera sobre todos los nodos pertenecientes al grafo y por cada nodo que su identificador en el arreglo de color sea blanco, llama a la función recursiva dfsVist.
* **dfs\_Visit:** toma como entrada un entero, como identificador de un nodo, registra el tiempo inicial de este nodo en el respectivo arreglo, así como también se registra como visitado en color al asignar gris a su posición en dicho arreglo. Posteriormente, por cada arista en la lista de adyacencia donde el color del otro nodo de la arista contrario al actual visitado sea blanco, se le asigna el actual nodo como predecesor, al actual se le asigna su adyacente como sucesor y se llama recursivamente a dfs\_Visit con el identificador de este adyacente. Al salir, de las iteraciones de la lista de adyacencia, se entiende que se cerró la visita al nodo, por tanto se registra su tiempo final y se le asigna negro en sus respectivos arreglos. Finalmente, se verifica si la información del nodo actual es igual a la buscada y se guarda en la variable respectiva.
* **predecesores:** toma como entrada un nodo y devuelve el identificador del nodo predecesor.
* **get\_All\_Predecesores:** retorna el arreglo de predecesor.
* **obtenerTiempo:** toma como entrada un nodo y retorna un par que indica el tiempo inicial y final del nodo.
* **get\_NodoEncontrado:** retorna el nodo encontrado, si la variable que lo guarda es distinta a *nil*. En caso contrario, retorna *nil*.
  + 1. **Main*:*** para el programa principal se decidió realizar la interacción con el usuario a través de un while en el que se le solicita instrucciones al cliente para inicializar el grafo, agregar un nodo, agregar una arista, encontrar un nodo según su información y salir del programa. Para disminuir la complejidad y aumentar la eficiencia del programa *switch case* para según cada acción del cliente poder realizar la acción consecuente. Para mayor entendimiento, se debe leer la siguiente sección *Manual del Usuario.*

1. **Manual del Usuario:**

Para la implementación del programa debe seguir los siguiente pasos:

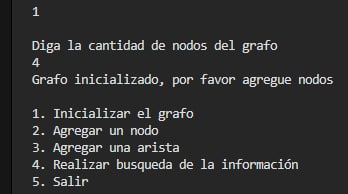
1. Abrir la consola del computador.
2. Buscar el directorio donde se encuentre guardado el programa.
3. Ingresar el comando:

A continuación, el programa le solicitará que introduzca un número según la acción que desea realizar:

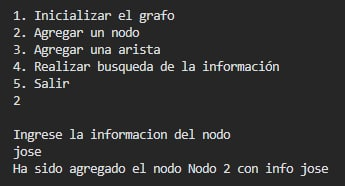


Como respuesta se espera que el usuario ingrese en número y luego presione la tecla enter.

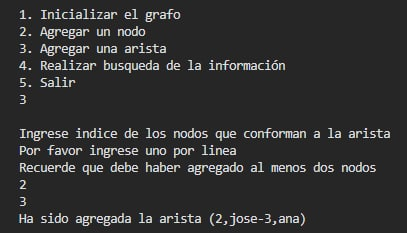
En caso de seleccionar ***“1. Inicializar el grafo”***, el programa le pedirá que indique el número de nodos que desea que el grafo tenga inicialmente. Posteriormente, el programa solicitará que ingrese los nodos. Para ello, recordará nuevamente el menú de acciones.



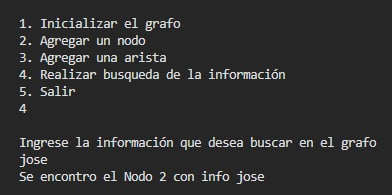
En caso de seleccionar ***“2. Agregar un nodo”***, el programa le solicitará que ingrese la información del nodo que se quiere ingresar (la asignación de identificadores lo hace el programa internamente). Para ello, debe colocar la información y presionar la tecla enter. Se le sugiere al usuario agregar la cantidad de nodos con lo que inicializó el grafo, antes de agregar aristas, aunque sólo será necesario tener ingresado al menos dos nodos antes de agregar una arista. Después de ingresar el nodo, el programa despliega otra vez el menú.



En caso de seleccionar ***“3. Agregar una arista”***, el programa solicita que inserte una arista nodo por nodo, es decir, debe colocar el identificador del nodo y presionar enter para ambos nodos que componen la arista. Una vez la arista es agregada, se despliega el menú de acciones.



En caso de seleccionar ***“4. Realizar la búsqueda de la información”,*** el programa solicita que inserte la información que desea buscar entre los nodos del grafo. Para ello, debe escribir la información y presionar la tecla enter. El programa devolverá la información completa del nodo, es decir, el identificador con su información respectiva que ha sido encontrada aplicando la implementación recursiva de búsqueda en profundidad.



En caso de seleccionar ***“5. Salir”***, el programa emitirá el mensaje *“Gracias por usar el programa”* hacia el usuario y terminará el programa.

Este programa permite realizar las acciones ***1, 2, 3*** y ***4*** tantas veces como el usuario lo desee por cada llamada al programa.

1. **Dificultades en la implementación:**

Durante la implementación de este programa en Ruby, se encontraron las siguiente dificultades:

* A pesar de que la sintaxis de la creación de clases, módulos y las importaciones al archivo principal del programa se logra entender con el tiempo de una manera sencilla, existe una curva de aprendizaje innegable para Ruby y no se contó con el tiempo suficiente para entender esta sintaxis a mayor profundidad y explotar las capacidades del lenguaje.
* No se encontró una implementación correcta para búsqueda en profundidad donde se pudiera, parar la búsqueda al encontrar el primer nodo con la información deseada, devolverlo y retomar la búsqueda desde el punto que se dejó y realizar el procedimiento con cada nodo que comparta la información deseada.

**Conclusiones y Recomendaciones**

**Referencias**