MANUAL DE ALGORITMOS DEL PROYECTO FINAL

Estructuras Computacionales Avanzadas

Asignatura:

"Estructuras Computacionales Avanzadas"

Maestro:

Miguel Ángel Meza de Luna

Alumnos:

- Luis Pablo Esparza Terrones
- Luis Manuel Flores Jiménez
- Juan Francisco Gallo Ramírez

Ingeniería en Computación Inteligente

3er Semestre

Índice

Introducción

En el ámbito de las estructuras computacionales avanzadas, el presente documento presenta los resultados y el análisis del proyecto final que abordó la implementación y optimización de diversos algoritmos fundamentales en el manejo de grafos. Este proyecto se enfocó en el desarrollo de soluciones eficientes para problemas clásicos utilizando tanto la representación de grafos mediante matrices como a través de listas de adyacencia.

Entre los algoritmos abordados se encuentran la Búsqueda en Anchura y la Búsqueda en Profundidad, técnicas esenciales para la exploración de grafos. Además, se abordaron problemáticas como la determinación de la conectividad de un grafo, la identificación de caminos más cortos entre nodos, y la aplicación de algoritmos de pareo tanto en grafos bipartitos como en grafos normales.

Cabe destacar que el éxito de este proyecto se debe en gran medida al enfoque colaborativo y al destacado trabajo en equipo, que permitió combinar habilidades individuales para alcanzar soluciones integrales y eficientes. Además, la plataforma GitHub se erigió como una herramienta central para la gestión del código fuente, la colaboración y el control de versiones, facilitando la integración continua y la entrega de un producto final robusto.

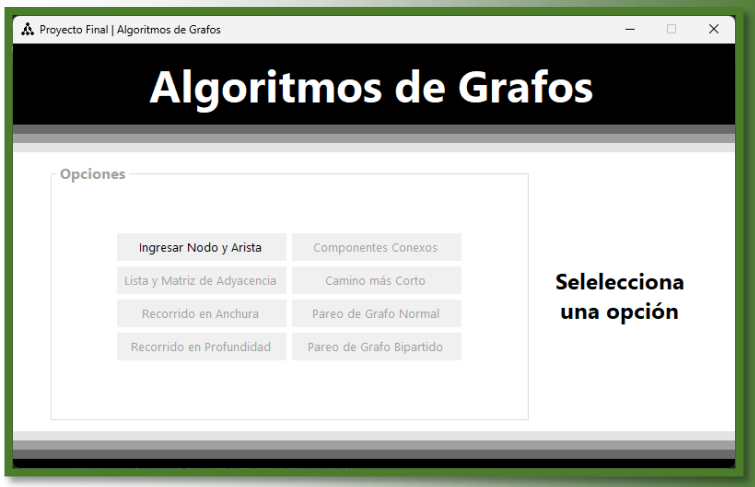
El proyecto no solo se enfocó en la implementación de algoritmos, sino que también hizo hincapié en la importancia de la programación orientada a objetos (POO), destacando la modularidad y la reutilización de código como principios fundamentales. El uso de conceptos de POO permitió construir soluciones más mantenibles y escalables, facilitando la extensión y adaptación de los algoritmos a diferentes contextos y problemas.

En resumen, este proyecto no solo representa un ejercicio técnico en el dominio de las estructuras computacionales avanzadas y la teoría de grafos, sino también una experiencia enriquecedora en cuanto a trabajo colaborativo, herramientas modernas de desarrollo y buenas prácticas de programación. Los resultados presentados a continuación reflejan el esfuerzo conjunto de un equipo comprometido con la excelencia técnica y la aplicación práctica de los conceptos aprendidos.

Algoritmos

Aplicación realizada

Para la realización del proyecto se elaboró una aplicación utilizando Windows Forms con C# en el contexto de estructuras computacionales avanzadas. En esta aplicación, se capitalizó la capacidad de C# para la programación orientada a objetos (POO), implementando diversos algoritmos fundamentales para grafos. La aplicación ofrece un menú intuitivo que permite la selección y ejecución de algoritmos, proporcionando una interfaz amigable para interactuar con las estructuras de datos y obtener resultados de manera eficiente.



Este enfoque modular no solo permitió una implementación más clara y estructurada, sino que también facilitó la incorporación y prueba de nuevos algoritmos en el futuro. La combinación de la versatilidad de C#, la interfaz gráfica de Windows Forms y la aplicabilidad de la programación orientada a objetos resulta en una solución robusta y accesible para el análisis de grafos en diversos escenarios.

La clase realizada para el proyecto es la clase llamada **`Grafo`**, esta tiene como atributos **`Dictionary <String, Dictionary<String, int>> listaAdyacencia`**, **`int[,] matrizAdyacencia`** y **`List<String> nodos`**. Estos atributos son fundamentales para trabajar con el grafo ingresado.

1. Ingresa Nodos y Arista

Esta opción es la que permite ingresar los nodos y sus respectivas aristas para crear un grafo, bien pueden ser aristas o arcos, ya que podemos elegir la opción mediante un CheckBox para especificar.

La función **`AgregarArista`** en la clase **`Grafo`** permite añadir una arista entre dos nodos especificados, estableciendo una conexión con un peso asociado. Inicia creando nodos en la lista de adyacencia si no existen, evitando duplicados. Además, valida la inexistencia de la arista para evitar conexiones duplicadas. Si el grafo es no dirigido, asegura la simetría de la conexión. Posteriormente, llama a **`GenerarMatriz`** para actualizar la representación de la matriz de adyacencia. La función retorna true si la arista se agrega con éxito, y false en casos de duplicados o conexiones no permitidas, proporcionando así una gestión eficiente y coherente de las relaciones en el grafo.

2. Lista y matriz de adyacencia

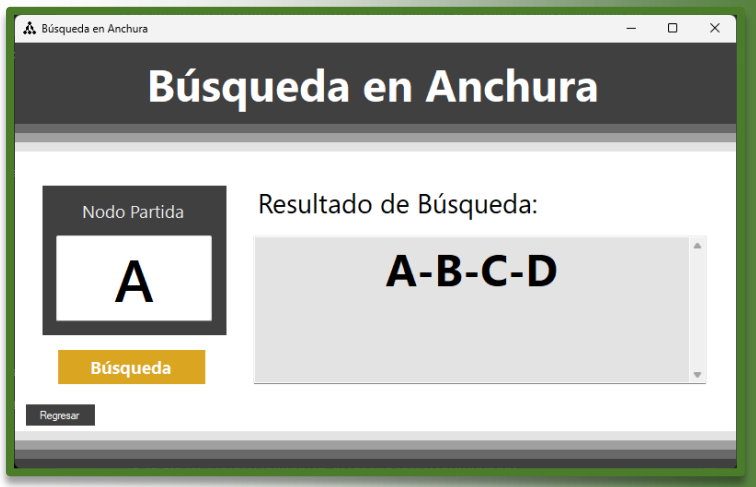
Esta opción es la que nos muestra la lista y matriz de adyacencia respectiva. Para representar los datos que contiene fueron necesarias dos tablas para la lista y la matriz, las cuales toman los datos de los atributos de la clase **`Grafo`**.

3. Recorrido en Anchura

Esta opción realiza el recorrido en anchura correspondiente al grafo ingresado y utiliza la función **`BusquedaAnchura`**. La función **`BusquedaAnchura`** implementa el algoritmo de búsqueda en anchura (BFS) en un grafo representado mediante listas de adyacencia. La función toma como parámetro

un nodo inicial v y realiza una búsqueda en anchura a partir de este nodo, explorando todos los nodos alcanzables desde v en un orden nivel por nivel. El resultado es una lista que contiene los nodos visitados en el orden en que fueron descubiertos durante la búsqueda. El algoritmo es el siguiente:

a) Se inicia con la creación de una lista anchura para almacenar los nodos visitados en orden de anchura y una cola queue para gestionar el orden de exploración.
b) El nodo inicial v se agrega tanto a la lista anchura como a la cola queue.
c) Se inicia un bucle mientras la cola no esté vacía:
    a. Se extrae un nodo v de la cabeza de la cola.
    b. Se recorren todos los nodos adyacentes a v.
    c. Si un nodo adyacente no está en la lista anchura, se encola y se agrega a la lista.
d) El bucle continúa hasta que la cola esté vacía, y finalmente, la lista anchura se devuelve como resultado de la búsqueda en anchura.

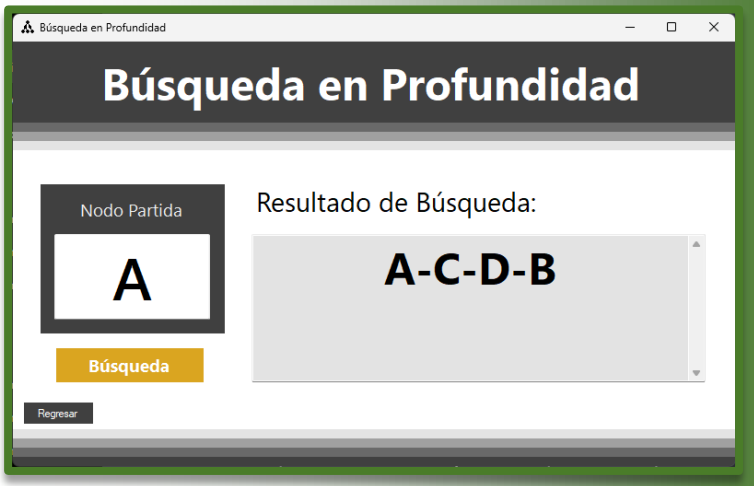


4. Recorrido en Profundidad

Esta opción realiza el recorrido en anchura correspondiente al grafo ingresado y utiliza la función **`BusquedaProfundidad`**. La función **`BusquedaProfundidad`** implementa el algoritmo de búsqueda en profundidad (DFS) en un grafo representado mediante listas de adyacencia. La función recibe como parámetro un nodo inicial v y realiza una búsqueda en profundidad a partir de este nodo, explorando tan profundamente como sea posible a lo largo de cada rama antes de retroceder. El resultado es una lista

que contiene los nodos visitados en el orden en que fueron descubiertos durante la búsqueda en profundidad. El algoritmo es el siguiente:

a) Se inicia con la creación de una lista profundidad para almacenar los nodos visitados en orden de profundidad y una pila stack para gestionar el orden de exploración.
b) El nodo inicial v se inserta en la pila stack.
c) Se inicia un bucle mientras la pila no esté vacía:
    a. Se extrae un nodo v de la cima de la pila.
    b. Si el nodo v no está en la lista profundidad, se agrega a la lista.
    c. Se recorren todos los nodos adyacentes a v.
    d. Si un nodo adyacente no está en la lista profundidad, se inserta en la pila.
d) El bucle continúa hasta que la pila esté vacía, y finalmente, la lista profundidad se devuelve como resultado de la búsqueda en profundidad.



5. Componentes Conexos

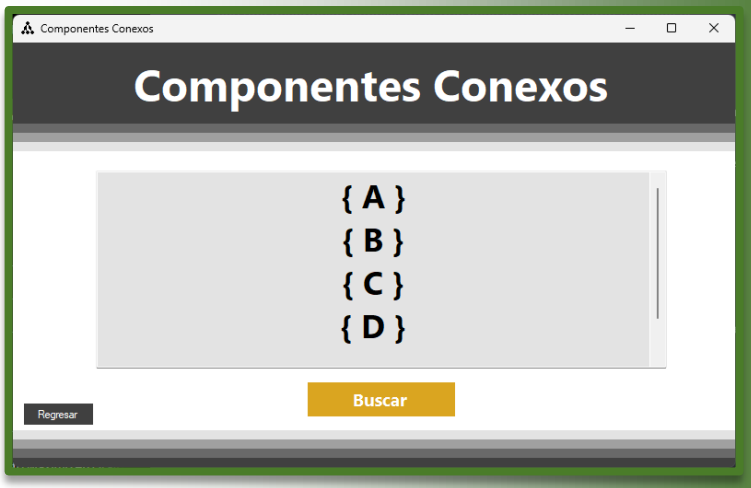
Esta opción es la que hace una búsqueda de componentes conexos en el grafo ingresado. Para la realización de este método fueron necesarios más de un método. Estos métodos forman parte de un proceso para obtener los componentes conexos en un grafo representado mediante una matriz de adyacencia. Aquí hay una breve explicación de cada uno:

- `MatrizCaminos`: Este método realiza la clausura transitiva en la matriz de adyacencia, es decir, busca caminos indirectos entre nodos para

asegurar que la relación de adyacencia sea reflexiva y simétrica. Utiliza el algoritmo de Warshall para actualizar la matriz y garantizar la conectividad de los nodos.

- `OrdenarMatriz`: Este método ordena la matriz y el arreglo de nodos en función de la suma de sus filas. Cuanto mayor sea la suma de una fila, más conexiones tiene el nodo correspondiente. Se utiliza para facilitar la identificación de componentes conexos al organizar los nodos en un orden descendente de acuerdo con su grado de conectividad.
- `SumaFila`: Calcula la suma de los valores en una fila específica de la matriz de adyacencia. Esto se utiliza en el método OrdenarMatriz para comparar las sumas de filas y ordenar en consecuencia.
- `SwapFila`: Intercambia dos filas en la matriz de adyacencia. Es utilizado en el método OrdenarMatriz para reorganizar la matriz según las sumas de las filas.
- `SwapString`: Intercambia dos elementos en un arreglo de strings. Este método se usa en conjunto con SwapFila para mantener la correspondencia entre los nodos y sus respectivas filas en la matriz durante el proceso de ordenamiento.
- `ComponentesConexos`: Este método utiliza los resultados de los métodos anteriores para construir y devolver una lista de strings que representa los componentes conexos del grafo. Organiza los nodos en grupos conectados y los coloca en conjuntos entre llaves, generando así una representación visual de los componentes conexos.

En conjunto, estos métodos proporcionan funcionalidades esenciales para analizar la conectividad de un grafo, identificar componentes conexos.

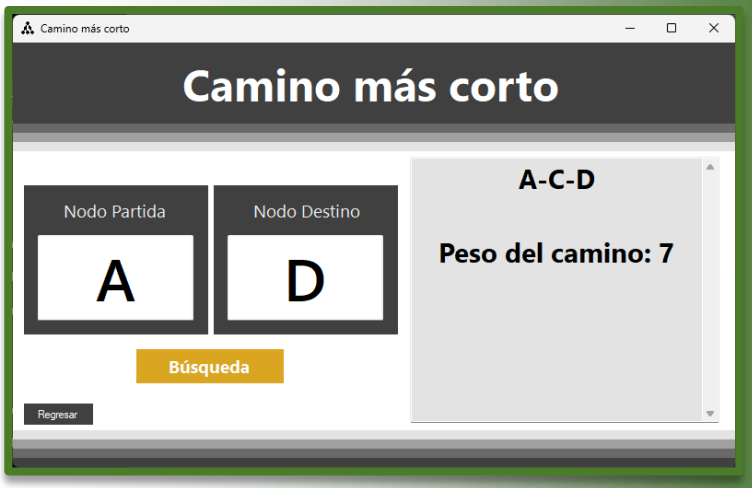


6. Camino más corto

Esta opción es la encargada de calcular el camino más corto con el algoritmo de Dijkstra.

- `Dijkstra`:
  a) Inicia creando diccionarios para las distancias y los predecesores, así como una lista de nodos no visitados.
  b) Inicializa las distancias con valores infinitos y establece la distancia desde el origen hasta él mismo en cero.
  c) Itera hasta que todos los nodos hayan sido visitados.
  d) Encuentra el nodo no visitado con la distancia más corta, actualiza las distancias y predecesores según los vecinos del nodo actual.
  e) Retorna el camino más corto desde el origen hasta el destino utilizando el método `ConstruirCamino`.
- `EncontrarNodoConMenorDistancia`: Busca y retorna el nodo no visitado con la distancia más corta.
- `ConstruirCamino`: Crea y retorna un camino reconstruido desde el origen hasta el destino utilizando los predecesores almacenados.
- `PesoCamino`: Calcula y retorna el peso total de un camino dado, sumando los pesos de las aristas en el grafo.

En resumen, estos métodos trabajan en conjunto para aplicar el algoritmo de Dijkstra en un grafo ponderado. La función principal, `Dijkstra`, calcula el camino más corto y el método `PesoCamino` proporciona el peso total del camino encontrado. Estos métodos son útiles para resolver problemas de búsqueda de rutas óptimas en grafos ponderados.

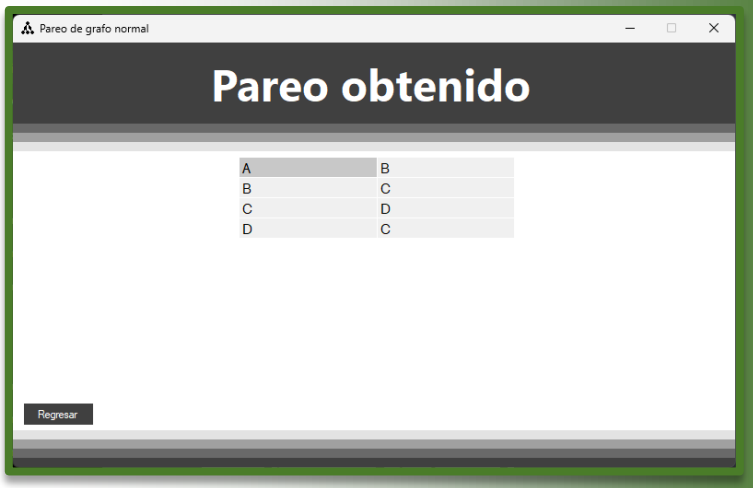


7. Pareo Grafo Normal

Esta opción es la que nos realiza un pareo para cualquier grafo ingresado. Consta de varios métodos, estos métodos están destinados a realizar el pareo de nodos en un grafo, lo que significa emparejar nodos de tal manera que no haya aristas cruzadas entre los pares emparejados. Aquí hay una descripción concisa de cada método:

- `PareoNormal`: Inicia creando un diccionario emparejamiento para almacenar los pares emparejados de nodos. Divide los nodos en dos listas: nodosIzquierda para nodos que aún no han sido emparejados, y nodosDerecha para nodos que ya tienen un emparejamiento. Itera sobre los nodos no emparejados (nodosIzquierda) y busca caminos de aumento utilizando el método EncuentraCaminoAumento. Retorna el diccionario emparejamiento que representa el pareo encontrado.
- `EncuentraCaminoAumento`: Utiliza la técnica de búsqueda en profundidad para encontrar un camino de aumento en el grafo bipartito. Recibe un nodo y un diccionario de emparejamiento. Intenta encontrar un camino de aumento desde el nodo dado, actualizando el emparejamiento en el proceso. Retorna true si se encuentra un camino de aumento, false en caso contrario.
- `ObtenerVecinos`: Obtiene los vecinos de un nodo específico en el grafo, utilizando la representación de listas de adyacencia.

En resumen, estos métodos implementan el algoritmo de búsqueda de caminos de aumento para encontrar el pareo máximo en un grafo bipartito. El resultado es un diccionario que representa la relación de emparejamiento entre los nodos del grafo.

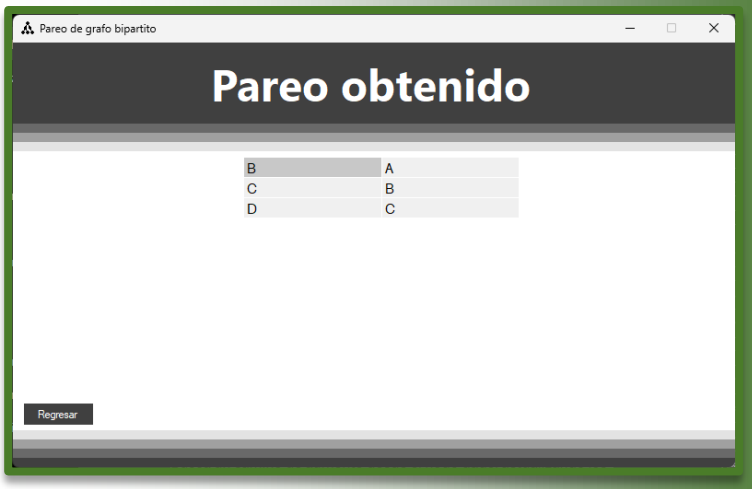


8. Pareo Grafo Bipartito

Esta opción es la que realiza el pareo correspondiente a grafos bipartitos. Los métodos que usa están diseñados para encontrar el pareo máximo en un grafo bipartito utilizando el algoritmo de aumentar caminos. A continuación, se presenta una descripción breve de cada método:

- `PareoBipartito`: Inicializa una matriz `grafoResidual` que representa el grafo bipartito en forma de matriz de adyacencia. Inicializa un arreglo pareja para almacenar los emparejamientos. Inicialmente, todos los elementos son -1, indicando que ningún nodo tiene pareja. Utiliza el método `AumentarCamino` para encontrar el pareo máximo.
- `AumentarCamino`: Implementa la técnica de aumentar caminos para encontrar el pareo máximo en el grafo bipartito. Recibe un nodo, la matriz grafo Residual, el arreglo pareja, un arreglo visitado y el diccionario emparejamientos. Busca un camino de aumento desde el nodo dado, actualizando los emparejamientos y retornando true si se encuentra un camino de aumento.

En resumen, estos métodos proporcionan una implementación eficiente del algoritmo de aumentar caminos para encontrar el pareo máximo en un grafo bipartito. La función principal, `PareoBipartito`, inicializa la matriz residual y luego utiliza la recursión del método `AumentarCamino` para encontrar y almacenar los emparejamientos máximos en el diccionario emparejamientos.



Conclusión

En el ámbito de las estructuras computacionales avanzadas, este proyecto ha abordado la implementación y optimización de algoritmos fundamentales para el manejo de grafos. Se han explorado diversos enfoques, desde la representación mediante matrices hasta el uso de listas de adyacencia, destacando la versatilidad y eficiencia de cada método. Los algoritmos cubiertos incluyen la Búsqueda en Anchura, la Búsqueda en Profundidad, la determinación de componentes conexos, el cálculo de caminos más cortos con el algoritmo de Dijkstra, y técnicas de pareo para grafos normales y bipartitos.

La colaboración y el trabajo en equipo fueron elementos clave para el éxito del proyecto, evidenciando la importancia de la colaboración en la programación y la utilización efectiva de herramientas como GitHub para la gestión del código fuente y el control de versiones.

El enfoque modular y orientado a objetos se destacó en la implementación de los algoritmos, subrayando la importancia de la programación orientada a objetos para lograr soluciones mantenibles y escalables. La aplicación desarrollada con Windows Forms en C# proporciona una interfaz amigable para interactuar con los algoritmos, capitalizando la capacidad de C# para la programación orientada a objetos y facilitando la incorporación de nuevos algoritmos en el futuro.

Cada algoritmo ha sido desglosado y explicado, desde la adición de aristas y la representación del grafo hasta los procesos de búsqueda y análisis de componentes conexos. La aplicación ofrece una herramienta completa para la manipulación y análisis de grafos, proporcionando resultados claros y visuales.

En resumen, este proyecto no solo es un ejercicio técnico en el ámbito de las estructuras computacionales avanzadas y la teoría de grafos, sino también una experiencia enriquecedora en cuanto a trabajo colaborativo, herramientas modernas de desarrollo y buenas prácticas de programación. Los resultados presentados reflejan el esfuerzo conjunto de un equipo comprometido con la excelencia técnica y la aplicación práctica de los conceptos aprendidos.