



FACULTAD DE INGENIERÍA
UNIVERSIDAD DE CONCEPCIÓN

INGENIERÍA CIVIL INFORMÁTICA

Matemáticas Discretas

INFORME

Profesor:
Arturo Antonio Zapata Cortés

Cristóbal González, Pablo Villagrán, Sebastián Vega, Vicente Moranda

Introducción

El uso de la tecnología está presente en diversas áreas, como las redes sociales, los sistemas de transporte y, especialmente, en infraestructuras como las redes eléctricas. En este contexto, se requiere analizar la robustez de estas redes frente a condiciones climáticas extremas, que pueden causar fallos en las conexiones y afectar su funcionamiento. Para evaluar la resistencia de estas infraestructuras, es fundamental verificar su k -conexividad.

Este concepto se refiere a la capacidad de una red para mantener su conectividad incluso cuando se eliminan hasta $k-1$ nodos (en este caso, nodos clave, como las centrales eléctricas). Si, tras la eliminación de estos nodos, la red sigue siendo conexa, esto indica que la infraestructura es robusta frente a fallos. Por lo tanto, la k -conexividad permite analizar y garantizar la estabilidad de la red eléctrica bajo situaciones de estrés, como tormentas o eventos extremos que puedan afectar varias de sus instalaciones.

A continuación, se mostrará una breve demostración de por qué la k -conexividad y la eliminación de conjuntos de $k-1$ nodos están relacionadas con la problemática planteada.

Definición de k -conexidad: Un grafo $G = (V, E)$ se dice k -conexo si cumple las siguientes dos condiciones:

1. $|V| > k$, es decir, el número de vértices del grafo es mayor que k .
2. Para todo conjunto de vértices $X \subseteq V$ tal que $|X| < k$, el subgrafo $G - X$ (el grafo resultante de eliminar los vértices en X) es conexo.

Demostración

1. **Suposición:** Sea $G = (V, E)$ un grafo k -conexo. Esto significa que G satisface las condiciones mencionadas anteriormente: $|V| > k$ y, para cualquier conjunto de vértices $X \subseteq V$ tal que $|X| < k$, el grafo $G - X$ es conexo.
2. **Conjunto de eliminación:** Considerese un conjunto $X \subseteq V$ de tamaño $|X| = k-1$, es decir, se eliminan $k-1$ vértices de G .
3. **Conectividad después de la eliminación:** Dado que G es k -conexo, la segunda condición de la k -conexividad implica que, para cualquier conjunto $X \subseteq V$ con $|X| < k$, el grafo resultante $G - X$ sigue siendo conexo. En este caso, como $|X| = k-1$, se cumple que $|X| < k$. Por lo tanto, el subgrafo $G - X$ debe ser conexo.
4. **Conclusión:** Como se ha demostrado que eliminar $k-1$ vértices del grafo G no desconecta el grafo, se concluye que eliminar un conjunto de $k-1$ vértices de un grafo k -conexo deja el grafo conexo.

Como solución, se plantea la idea de verificar la conectividad de estas redes; para ello, se emplearán algoritmos de verificación de k -conectividad, los cuales permiten evaluar la resistencia de dichas redes ante fallos aleatorios en sus nodos.

Para llevar a cabo esta verificación, se abordarán tres enfoques distintos que permitirán comprobar la k -conectividad de un grafo: el algoritmo de eliminación de vértices, el Teorema de Karl Menger y el algoritmo de Yefim Dinitz (Dinic).

Algoritmo eliminación vértices

En la demostración anterior sobre la k -conexividad, se ha establecido que un grafo k -conexo se mantiene conexo incluso después de eliminar hasta $k - 1$ vértices. Esta propiedad se puede emplear para desarrollar un algoritmo que verifique si un grafo es k -conexo, siendo k un número entre 1 y 4, según lo planteado.

En este apartado del informe, se explicará cómo diseñar el algoritmo de eliminación de vértices, basado en la definición de k -conexividad y en lo demostrado anteriormente.

1. **Inicialización de datos:** Al inicio del programa, se inicializan las estructuras de datos necesarias para representar el grafo, como el arreglo de visitados y la matriz de adyacencia que representa las conexiones entre nodos.
2. **Lectura del archivo de entrada:** El algoritmo lee un archivo de entrada que contiene la información sobre las conexiones entre nodos de la red eléctrica. En cada línea del archivo se define la conectividad de un nodo con otros nodos, lo que se almacena en una matriz de adyacencia.
3. **Comprobación de conectividad:** El algoritmo utiliza un algoritmo de búsqueda en profundidad (*DFS*) para verificar si el grafo es conexo. Esto significa que, desde cualquier nodo, debe ser posible llegar a todos los demás nodos, independientemente de cuántos nodos hayan fallado.
4. **Cálculo de la k -conexividad:** Si el grafo es conexo, el algoritmo procede a verificar la k -conexividad. Para ello, elimina sucesivamente nodos del grafo y comprueba si la red sigue siendo conexa. Si la red sigue siendo conexa después de eliminar $k - 1$ nodos, el grafo es k -conexo. Esto permite analizar la resistencia de la red frente a la pérdida de nodos importantes, como las centrales eléctricas.
5. **Resultados:** Finalmente, el programa muestra si el grafo es conexo y su k -conexividad, proporcionando así información clave sobre la estabilidad de la infraestructura eléctrica ante posibles fallos.

Complejidad del Algoritmo de Eliminación de Vértices

La complejidad del algoritmo de eliminación de vértices depende del número de vértices V en el grafo y el número de vértices que se eliminan en cada iteración ($k - 1$). El proceso de verificación de conectividad tras la eliminación de vértices requiere realizar una búsqueda en profundidad (*DFS*).

La complejidad de la búsqueda en profundidad (*DFS*) es $(V + E)$, donde V es el número de vértices y E es el número de aristas del grafo. En el caso de eliminación de $k - 1$ vértices, el algoritmo tiene que realizar $k - 1$ iteraciones para verificar si el grafo sigue siendo conexo después de eliminar los vértices. En cada iteración, el algoritmo realiza una búsqueda DFS en el grafo resultante.

$$((k - 1) \cdot (V + E))$$

Donde:

- k es el número de vértices a eliminar.

- V es el número de vértices en el grafo.
- E es el número de aristas del grafo.

Algorithm 1 Algoritmo Eliminacion de vertices

Require: Grafo $G = (V, E)$, conjunto de nodos a eliminar $S \subset V$

Ensure: Verificar conectividad después de eliminar S

```

1: Eliminar todos los nodos en  $S$  y sus aristas incidentes de  $G$ 
2: Seleccionar un nodo  $v \in V \setminus S$  como nodo inicial (si existe)
3: Inicializar todos los nodos en  $V \setminus S$  como no visitados
   DFS( $G$ , nodo  $u$ )
4: Marcar  $u$  como visitado
5: for cada vecino  $w$  de  $u$  en  $G$  do
6:   if  $w$  no ha sido visitado then
7:     DFS( $G$ ,  $w$ )
8:   end if
9: end for
10: Llamar a DFS( $G$ ,  $v$ ) para explorar el grafo
11: for cada nodo  $x \in V \setminus S$  do
12:   if  $x$  no ha sido visitado then
13:     return 1
14:   end if
15: end for
16: return 0 = 0

```

Teorema de Carl Menger

1. **Carl Menger** De manera introductoria a este teorema, es importante conocer a su creador, Carl Menger. Carl Menger fue un economista austriaco, uno de los más importantes de su época. Es el fundador de la Escuela Austriaca de Economía y el creador de muchos principios y teoremas, tales como la teoría marginalista y la teoría del valor subjetivo. En este caso, nos centraremos en el Teorema de Menger, teorema probado por Carl Menger en 1927 que caracteriza la conectividad de un grafo.
2. **Teorema:** El Teorema de Menger establece que, en un grafo finito, la cantidad máxima de caminos disjuntos (caminos que no comparten ningún vértice) que pueden unir dos conjuntos de puntos es igual a la cantidad mínima de puntos cuya eliminación desconectaría esos conjuntos. En otras palabras, el Teorema de Menger afirma que, en un grafo, el tamaño de un conjunto de corte mínimo es igual al número máximo de caminos disjuntos que se pueden encontrar entre cualquier par de vértices. En términos más simples, si A y B son dos conjuntos de puntos, el número de caminos disjuntos que conectan A y B es igual al número de vértices que se deben eliminar para perder la conectividad entre el punto A y el punto B.
3. **Ejemplo:** Supongamos que estamos en un punto de la Universidad de Concepción y queremos llegar a un punto de Talcahuano. Los caminos disjuntos en este caso serían las distintas calles por las que podemos llegar a Talcahuano sin que se crucen, es decir, los caminos que no comparten la misma calle. Ahora, si se decidiera cerrar todas las calles que nos servían, el Teorema de Menger indica que el número de calles que se deben cerrar para desconectar la UdeC de Talcahuano es igual al número máximo de calles que existían al principio y que podían conectar estos dos puntos sin cruzarse. Así, si tengo 5 calles que me llevan directamente a Talcahuano, al cerrar 4 de ellas, todavía quedará una calle que me permitirá llegar a mi destino.

Demostración

Comenzamos analizando algunos casos particulares. Sea $V' \subseteq V \setminus \{u, v\}$ un u - v -corte mínimo, donde u y v son vértices distintos y no adyacentes del conjunto de vértices V , y $|V'| = k$. Sea G un grafo.

- Si $k = 0$, entonces el grafo G es desconexo, ya que no existe un camino entre u y v .

Ahora, según el Teorema de Menger, no deben existir más de k caminos disjuntos. Procedemos por contradicción, suponiendo que $\{C_1, \dots, C_j\}$ son caminos disjuntos u - v con $j > k$. Entonces, si V' es cualquier u - v -corte mínimo, $|V'| = k$. Por lo tanto, en $G - V'$ se han eliminado a lo sumo k de los j caminos. Como $j > k$, algún camino no ha sido modificado, lo que implica que sigue existiendo un camino entre u y v en $G - V'$, lo que genera una contradicción. Así, no pueden existir más de k caminos disjuntos.

Ahora debemos probar que existen k caminos disjuntos entre u y v con $k \in \mathbb{N}$. Esto lo lograremos mediante inducción sobre el tamaño n de G .

Base $n = 0$

En el caso trivial, no existen aristas, por lo que $k = 0$.

Hipótesis de inducción

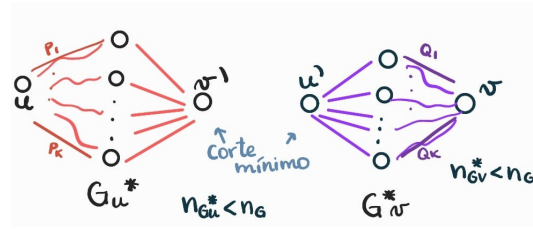
Demostraremos que si se cumple para un grafo H con $n_H < n_G$, entonces el teorema se cumple también para G . Para esto, es necesario analizar una serie de casos.

Caso 1: Si u y v están a distancia 2, separados por un único vértice w , entonces todo V' u - v -corte mínimo debe contener a w . Por lo tanto, $|V' \setminus \{w\}| = k - 1$ y $V' \setminus \{w\}$ es un u - v -corte mínimo en $G - w$. Luego, como $n_G - 1 < n_G$, se cumple el teorema por hipótesis de inducción.

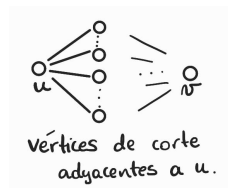


Caso 2: Existe $V' = \{w_1, \dots, w_k\}$ tal que hay algún vértice no adyacente a v y algún vértice no adyacente a u . Sea $G_u \setminus \{v\}$ el grafo de todos los caminos que inician en u y terminan en w_i sin pasar por otro w , y $G_v \setminus \{u\}$ su análogo. Si existen $\{P_1, \dots, P_k\}$ caminos de u a w_i y se crea un vértice v' , tendremos i caminos entre u y v' , lo mismo para v con un u' conectados por caminos $\{Q_1, \dots, Q_k\}$. Luego, por hipótesis de inducción, existen k caminos P_i de u a w_i internos disjuntos, y existen k caminos Q_i de w_i a v internos disjuntos. Por lo tanto, los caminos $\{P_1Q_1, P_2Q_2, \dots, P_kQ_k\}$ (concatenación) son k caminos disjuntos entre u y v .

Caso 3: Para todo V' , u es vecino de todos los w_i o v es vecino de todos los w_i , con $d(u, v) > 2$ y $k \geq 2$. Sea $C = (u, x, y, \dots, v)$ una u - v -geodésica, es decir, un camino más corto entre u y v . Sea $a \in A_G$ tal que $Y(a) = xy$, consideramos el grafo $G - a$. Debemos probar que $G - a$ tiene V'' u - v -corte mínimo con $|V''| = k$.



Supongamos que $|V''| = k - 1$, entonces $V'' \cup \{x\}$ es un $u-v$ -corte mínimo en G . Notemos que x es vecino de u , por lo que todos los vértices en V'' son vecinos de u . Además, $V'' \neq \emptyset$ con $k \geq 2$, y no son vecinos de v . $V'' \cup \{y\}$ es un $u-v$ -corte mínimo en G , por lo que hay vértices de este conjunto vecinos a u y, por hipótesis, todos lo son. En particular, y es adyacente a u , lo que genera una contradicción, ya que, según C , la existencia de x implicaría que habría dos caminos entre u y y , lo cual no es posible porque C es geodésica.



Conclusión

Se han revisado todos los conceptos necesarios para entender el Teorema de Menger, desde su significado hasta su relación con la conectividad de un grafo y su demostración. El Teorema de Menger ha sido de gran ayuda en el mundo moderno y se utiliza en áreas de optimización de redes, teoría de flujos en logística y transporte, teoría de juegos y, por supuesto, en infraestructuras eléctricas, permitiendo diseñar redes de manera que puedan soportar fallos sin perder conectividad.

Pseudocodigo

Algorithm 2 VerificarConectividad(Grafo, u, v)

Entrada: Grafo con vértices y aristas, vértices u y v

Salida: **true** si el grafo es k-conexo, **false** en caso contrario

Definir función **Conectado**(Grafo, u, v)

Si existe arista entre u y v **entonces return true**
 return false

Definir función **Conectividad**(Grafo, vértices, aristas)

 caminos \leftarrow 0

Para cada vértice i del Grafo **hacer**

Si $i \neq u$ y $i \neq v$ **entonces**

Si **Conectado**(Grafo, u, i) y **Conectado**(Grafo, v, i) **entonces**
 caminos \leftarrow caminos + 1

Fin Si

Fin Si

Fin Para

return caminos

Definir función **VerticesCorte**(Grafo, vértices, aristas)

 vert_corte \leftarrow 0

 // Implementación algoritmo para encontrar los vértices de corte

return vert_corte

caminos \leftarrow **Conectividad**(Grafo, vértices, aristas)

vert_corte \leftarrow **VerticesCorte**(Grafo, vértices, aristas)

Si caminos > vert_corte **entonces return true**

return false

=0

Teorema Dinic

El algoritmo de Dinic es un método bastante práctico desarrollado por el científico de la computación Yefim Dinitz, que busca encontrar el flujo máximo en un grafo con una red de flujo, en un tiempo polinomial de $O(V^2 \cdot E)$.

Este algoritmo consta de tres partes principales: la construcción de un grafo de niveles, la búsqueda de bloqueos de flujo y la construcción de un grafo residual. Estas tres fases se desarrollan en ese orden y permiten encontrar la capacidad o el flujo máximo de un grafo.

En la primera parte, que consiste en la construcción del grafo de niveles, el objetivo es encontrar todos los vértices que están a una distancia $n + 1$, siendo n el nivel actual del grafo. Para ello, comenzamos en el vértice fuente y exploramos cuáles otros vértices podemos alcanzar, marcándolos con el nivel 1. Luego, eliminamos todas las aristas que conecten dos vértices del mismo nivel. Este proceso continúa con el resto del grafo, construyendo un grafo con caminos entre el vértice fuente y el sumidero de longitud mínima. Esto se logra mediante una búsqueda en anchura (*BFS*), que no solo explora el grafo, sino que también marca las aristas que pasan de un nivel al siguiente, así como el nivel de cada vértice.

```
for vertice_i in V:
    for vertice_j in V:
        if vertice_j != vertice_i && (vertice_i, vertice_j) in combinaciones_usadas == false:
            //nos aseguramos que sean distintos los vértices y
            //de que no se haya probado antes dicha combinación
            vertice_j = t // seteamos un vértice como sumidero
            vertice_i = s // seteamos un vértice como fuente

            G' = BFSdenivel(G) //transformamos el grafo en un grafo
                               //de nivel haciendo uso de un BFS
                               //y eliminando las aristas que no sirven
```

Luego, el algoritmo busca los caminos que llegan al vértice sumidero, identificando los caminos directos que permiten la mayor cantidad de flujo. Esto se logra mediante una búsqueda en profundidad (*DFS*), que recorre el grafo de niveles enviando la mayor cantidad de flujo posible por las aristas. El algoritmo avanza por los caminos hasta llenar todas las aristas con su respectivo flujo máximo, sumando el total de flujo final, cantidad que se conoce como el bloqueo de flujo del grafo. Durante este proceso, el algoritmo marca las aristas que se han llenado con flujo, de manera que, al buscar el flujo residual, no se vuelvan a utilizar las mismas aristas, ya que no pueden transportar más flujo del que su capacidad máxima permite. Además, el algoritmo registra la cantidad de flujo que ha pasado por las aristas que no se han llenado por completo.

```
flujo_max = encontrar_flujo_maximo_inicial(G')
//encuentra los caminos de s a t, marcando cuales aristas se usaron,
//aplicando un DFS que recorre los caminos,
//y retorna el valor del flujo, el cual sería

Gr = transformar_a_grafo_residual(G')
//toma el grafo G', y busca las aristas marcadas en el paso anterior,
//las cuales bloquea y no permite que se avnze por ellas
```

Por último, se debe construir el grafo de flujo residual, el cual se toma del grafo inicial, pero con las aristas que ya se han llenado en el paso anterior bloqueadas. Las aristas

que no se llenaron tienen su capacidad reducida según el flujo que ha pasado por ellas. Después de construir este grafo residual, se busca un camino desde la fuente hasta el sumidero. En caso de que sea posible encontrar dicho camino, se suma su capacidad al bloqueo de flujo obtenido previamente. Este proceso se repite hasta que el vértice sumidero no se puede alcanzar desde la fuente, lo que indica que se ha alcanzado el flujo máximo del grafo.

```
flujo_max += caminos_finales(Gr)
//busca los caminos que quedan entre s y t
//que no se encontraron en la primera iteración
//y retorna el valor de estos caminos
///##ESTE PASO SE APLICA A GRAFOS DE PESO DISTINTO DE UNO##
///##YA QUE EN GRAFOS DE PESO UNO LAS ARISTAS NO TIENE UN SOBRANTE##
//de otra forma, si el grafo tiene capacidad
```

Corte Mínimo

El algoritmo de Dinic también puede ser utilizado para encontrar el corte mínimo de un grafo. Una vez que se ha obtenido el grafo residual, es posible identificar los vértices que no son alcanzables desde la fuente y los que sí lo son. El corte mínimo se define como la suma de las capacidades restantes de las aristas que conectan los vértices alcanzables desde la fuente con los vértices no alcanzables. De esta manera, el valor del corte mínimo coincide con el valor del flujo máximo.

Relación Conectividad

Finalmente, una de las utilidades del corte mínimo de un grafo es que puede ayudar a conocer la conectividad de un grafo aplicando el algoritmo de Dinic. Esto se debe a que el corte mínimo indica el flujo mínimo que debe ser cortado para desconectar los vértices fuente y sumidero del grafo. De esta forma, si se tienen las capacidades del grafo y el grafo residual generado por el algoritmo al momento de obtener el corte mínimo, es posible identificar las aristas que deben ser eliminadas para que el grafo deje de ser conexo, impidiendo que se pueda llegar de la fuente al sumidero. Aplicado a la problemática de eliminación de vértices, al eliminar aquellos vértices que contienen las aristas del corte mínimo, se perdería la conexidad en el grafo, lo que permite determinar cuáles y cuántos vértices limitan la conectividad.

Aclaraciones en el proyecto

Durante el desarrollo del pseudocódigo, se observó que el algoritmo de Dinic implementado para la búsqueda de la conectividad permite determinar la conectividad de manera sencilla en los grafos a los que está diseñado. Esto se debe a que, si se tiene un grafo en el que se asigna una capacidad de 1 a cada arista, no es necesario generar el grafo residual, ya que las aristas quedan saturadas y no tienen capacidad residual. Además, todos los caminos que salen o llegan al vértice sumidero no pueden repetirse.

```

G = (V,E): grafo entregado

for aristas in E:
    asista.add.peso(1) // seteamos la capacidad de las aristas del grafo a 1,
                        // para poder hacer un dinic que entregue el valor
                        // exacto de la conectividad

flujo_max_minimo = 0
combinaciones usadas []

```

Dicho esto, se observa que el corte mínimo, al ser igual al flujo máximo, estará limitado por los siguientes factores: las vecindades de s y de t , la existencia de aristas puente, o el grado mínimo del grafo. Lo primero afecta la conectividad, ya que, si el grado de entrada de t o de salida de s es menor a la cantidad de caminos disjuntos posibles, el flujo máximo será dicho valor, dado que no se pueden generar más caminos que induzcan a una mayor conectividad. Además, al eliminar los vértices que generan dichas vecindades, el grafo quedará desconexo. El segundo caso se da porque, si se tiene alguna arista puente que divide dos secciones del grafo, al pasar por dicha arista con el algoritmo de Dinic, solo se podrá pasar una vez, ya que se saturará después de eso. Por lo tanto, el flujo máximo será 1. Finalmente, el grado mínimo de un grafo puede determinar el valor del flujo máximo, ya que, para algunas de las posibles orientaciones de los vértices, se obtendrá como flujo máximo dicho grado. Al igual que en el primer caso, esto dará lugar a un corte mínimo igual al grado mínimo del grafo.

```

G = (V,E): grafo entregado

for aristas in E:
    asista.add.peso(1) // seteamos la capacidad de las aristas del grafo a 1,
                        // para poder hacer un dinic que entregue el valor
                        // exacto de la conectividad

flujo_max_minimo = 0
combinaciones usadas []

```

Pseudocodigo

Algorithm 3 Calcular la conectividad del grafo usando el algoritmo de Dinic

Require: Grafo $G = (V, E)$

```
1: Para cada arista  $a$  en  $E$ :
2:   Establecer la capacidad de las aristas del grafo a 1
3:   Esto permite hacer un Dinic que entregue el valor exacto de la conectividad
4:  $flujo\_max\_minimo \leftarrow 0$ 
5:  $combinaciones\_usadas \leftarrow [*]$ 
6: for cada  $vertice1$  en  $V$  do
7:   for cada  $vertice2$  en  $V$  do
8:     if  $vertice1 \neq vertice2$  y  $(vertice1, vertice2) \notin combinaciones\_usadas$  then
9:       Asegurarse que se han visitado todos los v rtices
10:      Evitar combinaciones ya probadas
11:       $vertice1 \leftarrow s$    (Se establece vertice1 como fuente)
12:       $vertice2 \leftarrow t$    (Se establece vertice2 como sumidero)
13:       $G' \leftarrow \text{BFS\_en\_nivel}(G)$    (Transformamos el grafo en uno en niveles con un BFS, eliminando las aristas que no sirven)
14:       $flujo\_max \leftarrow \text{encontrar\_flujo\_maximo\_inicial}(G')$ 
15:      Mientras haya caminos desde  $s$  a  $t$ :
16:        Marcar los nodos  $s$  y  $t$ 
17:        Usar DFS para recorrer los caminos
18:        Eliminar los nodos bloqueados en la  ltima iteraci n
19:         $Gr \leftarrow \text{transformar\_a\_grafo\_residual}(G')$    (Transformamos  $G'$  en un grafo residual, marcando y buscando aristas bloqueadas)
20:         $\text{Obtener\_caminos\_finales}(Gr)$    (Buscar los caminos entre  $s$  y  $t$  que no se encontraron en la primera iteraci n)
21:        if  $flujo\_max < flujo\_max\_minimo$  then
22:           $flujo\_max\_minimo \leftarrow flujo\_max$    (Guardar el m nimo valor de todos los flujos m ximos)
23:        end if
24:         $combinaciones\_usadas \leftarrow (vertice1, vertice2)$ 
25:      end if
26:    end for
27:  end for
28: Imprimir "La conectividad del grafo obtenida a partir del algoritmo de Dinic es de  $flujo\_max\_minimo$ "
    =0
```

Aspectos Técnicos

Los sistemas operativos usados fueron principalmente Windows (WSL) y Linux Mint. A continuación se detallan las especificaciones técnicas de las máquinas utilizadas para llevar a cabo las pruebas:

Especificaciones de las Máquinas Utilizadas

- **Plataforma Windows:**

- Arquitectura: 64 bits
- Núcleos: 8
- Hilos: 12
- Modelo: 12th Gen Intel(R) Core(TM) i5-12450H 2.00 GHz
- RAM: 16,0 GB
- Sistema Operativo: Windows 11
- Herramientas de desarrollo: WSL, GCC

- **Plataforma Linux Mint:**

- Arquitectura: x86_64
- Núcleos: 8
- Hilos: 2
- Modelo: AMD Ryzen 3 5300U with Radeon Graphics
- RAM: 9.6GB
- Sistema Operativo: Linux Mint 22
- Compilador utilizado: GCC 13.2.0

Se recomienda el uso de Linux para la ejecución del código dado las funciones que llaman al sistema.

Descripción Datos

El formato de entrada estándar tiene la siguiente estructura:

- La primera línea contiene el número de vértices.
- Las siguientes líneas están en el formato " n : " conexiones en forma creciente, donde n es el número del vértice, y las conexiones están separadas por coma y espacio, en caso de ser un vertice sin conexiones debe ser solo " n : " :

Ejemplo:

```
3
1: 2
2: 1, 3
3: 2
```

En este caso, el número de vértices es 3 y las conexiones son:

- El vértice 1 está conectado al vértice 2.
- El vértice 2 está conectado a los vértices 1 y 3.
- El vértice 3 está conectado al vértice 2.

El algoritmo procesara el archivo en el siguiente orden:

1. **Lectura de la primera línea:** Lee la primera línea del archivo, que contiene el número de vértices del grafo. Este valor se utiliza para determinar el tamaño de la matriz de adyacencia, la cual será de dimensiones $V \times V$, donde V es el número de vértices.
2. **Creación de la matriz de adyacencia:** Utilizando el número de vértices leído, se crea una matriz de adyacencia de tamaño $V \times V$. Esta matriz se inicializa con ceros.
3. **Lectura y procesamiento de las líneas restantes:** Posteriormente, el algoritmo lee cada una de las líneas siguientes del archivo, que describen las conexiones de los vértices. Si una línea contiene conexiones, se procesan e identifican los vértices conectados. Para cada vértice conectado, el algoritmo marca un 1 en la matriz de adyacencia, indicando que hay una arista entre los vértices correspondientes
4. **Cálculo del grado de cada vértice:** A medida que se procesan las conexiones, también se cuenta el número de conexiones (grado) de cada vértice. Esta información se utiliza para determinar el **grado máximo** (el mayor número de conexiones) y el **grado mínimo** (el menor número de conexiones) entre los vértices.

Los archivos de entrada fueron creados manualmente. El criterio de creación de estos archivos se basó en la casos extremos (ej triviales), simples y que requieran un gran uso de memoria, dentro de los que se probaron estan:

Grafos tipo árbol: Se generaron grafos con una estructura de árbol para evaluar cómo el algoritmo maneja grafos conexos sin ciclos.

Grafos desconexos: Se crearon grafos donde algunos o todos los vértices no están conectados entre sí, permitiendo probar la detección de grafos desconexos y la verificación de la conectividad.

Grafos con órdenes grandes: Para probar el rendimiento del algoritmo en grafos grandes.

Grafos con órdenes pequeños: También se consideraron grafos de orden menor a 4, para evaluar el comportamiento del algoritmo en casos simples.

Implementación de Código

Para comenzar, el programa pedirá que quiere hacer el usuario, tiene 2 opciones

1. Cargar un grafo que esté en la misma carpeta
2. Salir del programa

En caso en que el usuario no elija ninguno de las opciones ofrecidas lanzara una excepcion, en caso de elegir la 1 iniciara todo el proceso de analisis

Lo que primero hara el programa es pedirle el nombre del documento para posteriormente empezar el proceso de lectura, utilizando la librería `stdio.h` y las funciones `fopen` y `fgets`. Con `fopen`, se abre el archivo en modo de solo lectura ("r"), lo que permite acceder a su contenido sin modificarlo. Posteriormente, con `fgets`, se obtiene el contenido línea por línea en forma de cadena de texto.

La primera línea del archivo contiene el número de vértices del grafo, que lee con `fgets`. se utiliza la libreria `stdlib.h` y la funcion `atoi` para poder transformar el valor del numero en la cadena de caracteres a un entero y lo asigna a la variable global `V`, la cual representa el orden del grafo. Usando este valor, crea dinámicamente una matriz de adyacencia de tamaño $V \times V$ con `malloc`. También inicializa una lista llamada `visitado`, que utiliza más adelante en la búsqueda en profundidad (DFS) para realizar el recorrido del grafo. La función `ResetVisitados` se emplea para restablecer todos los elementos de la lista `visitado` a 0, asegurando que todos los nodos estén sin visitar al comenzar un nuevo recorrido. Adicionalmente, inicializa todos los valores de la matriz de adyacencia a 0, en caso de que el orden del grafo se 0 o 1 nos vamos a saltar el analisis ya que al ser caso triviales ya se sabe el resultado.

Para poder obtener las conexiones entre los vértices, usa un bucle `while` que itera sobre las líneas restantes del archivo hasta `n` líneas ya que solo va a realizar lectura de los que contienen los vertices. Define una variable `fila` para identificar el vértice al que se le va a analizar las conexiones. Utiliza `strchr` y `strtok` para analizar las conexiones en cada línea: `strchr` nos permite encontrar el primer espacio entre el numero del vértice y sus conexiones, lo que marca el punto de inicio para `strtok`. Luego, `strtok` divide esta lista de conexiones usando como criterio de separación coma y espacio (", "). Esto permite extraer el número de cada vértice al que el vértice actual está conectado, y con ese numero poder marcar un 1 en la matriz de adyacencia indicando la conexion entre los vertices.

Además, al saber cuántas iteraciones realiza la función `strtok`, determina el grado del vértice, ya que esto indica la cantidad de conexiones que tiene dicho vértice. Para ello, utiliza la variable `Grad`, que cuenta la cantidad de conexiones. Una vez calculado el grado, utiliza `GradMax` y `GradMin` para comparar el valor de `Grad` con los grados de otros vértices y así definir si es mayor o menor que los últimos valores leídos. En caso de no haber ninguna conexión, significa que no hay espacio entre el número del vértice y las conexiones, por lo que no utilizara `strtok` y pasa a la siguiente línea.

Después de extraer todos los datos del grafo, cierra el archivo y comienza a revisar la *K-Conexidad*.

Para poder verificar la K-conexividad usara 3 funciones importantes el algoritmo de DFS Depth-first search que se encargara de recorrer y marcar todos los vertices como visitados de manera recursiva, la funcion `esConexo`, que se encarga de verificar la conexividad gracias a el algoritmo DFS y `resetVisitados()` enacargado de resetear el arreglo visitados.

Función DFS

La función DFS realiza un recorrido en profundidad (*Depth First Search*) en el grafo comenzando desde el nodo v . Esto es usado ya que permite explorar todos los nodos que son accesibles desde un nodo cualquiera.

Detalle del proceso

- **Marcado de nodo como visitado:** La primera línea dentro de DFS establece `visitado[v] = 1`, lo que significa que el nodo v ha sido visitado. Esto para evitar ciclos y que el recorrido no entre en un bucle infinito, ya que no intentará visitar de nuevo los nodos que ya fueron alcanzados.
- **Recorrido de los nodos vecinos:** La siguiente sección de DFS es un bucle `for` que recorre todos los posibles nodos i del grafo (de 0 hasta $V - 1$, donde V es el número de nodos en el grafo). En cada iteración, DFS revisa si:
 - Existe una arista entre el nodo v y el nodo i , es decir, si `grafo[v][i] == 1`.
 - El nodo i no ha sido visitado, es decir, `visitado[i] == 0`.

Si ambas condiciones se cumplen, DFS llama a sí misma recursivamente con el nodo i . Esto significa que la función empieza un nuevo recorrido en profundidad desde el nodo i , marcándolo y visitando a su vez sus vecinos no visitados. Esta estructura recursiva permite explorar todos los nodos conectados al nodo de partida v .

Función `esConexo()`

La función `esConexo` utiliza DFS para determinar si el grafo es conexo. Un grafo es conexo si existe un camino entre cualquier par de nodos; en otras palabras, si se puede llegar desde cualquier nodo hasta cualquier otro nodo del grafo.

Detalle de `esConexo()`

- **Búsqueda de un nodo de inicio no visitado:** Primero, `esConexo` recorre todos los nodos desde 0 hasta $V - 1$ para buscar un nodo no visitado (`visitado[r] == 0`). Si encuentra un nodo r que no ha sido visitado, `esConexo` llama a `DFS(r)`. Esta llamada debería visitar todos los nodos conectados a r .
- **Verificación de conectividad:** Después de ejecutar DFS, `esConexo` verifica si todos los nodos han sido visitados. Esto se hace recorriendo el arreglo `visitado` y comprobando si hay algún nodo i tal que `visitado[i] == 0`. Si existe un nodo no visitado, significa que el grafo no es conexo, ya que desde el nodo r inicial no fue posible alcanzar a todos los demás. En este caso retorna `false`, indicando que el grafo no es conexo.

Sin embargo, si después de la llamada a DFS todos los nodos aparecen como visitados (`visitado[i] == 1` para todos i), entonces `esConexo` retorna `true`. Esto significa que el grafo es conexo, ya que todos los nodos fueron alcanzables desde el nodo inicial, en ambos casos llamara a `ResetVisitados()` para posteriormente probar otra combinacion.

Función `ResetVisitados()`

La función `ResetVisitados` es una funcion que restablece el estado de todos los elementos en el arreglo `visitado` a 0. Esto asegura que todos los nodos vuelvan a estar marcados como no visitados, lo cual permite seguir utilizando DFS varias veces en el mismo grafo, para luego comprobar eliminado conjuntos de $K-1$ vertices.

Detalles de `ResetVisitados`

- Recorre cada índice del arreglo `visitado` (de 0 a $V - 1$) y establece cada elemento a 0. Esto se asegura de que todos los nodos estén listos para una nueva ejecución de DFS.

La funcion `esConexo()` se utiliza 4 veces, la primera vez para comprobar si es conexo por lo cual basta con marcar un vertice cualquiera y empezar a hacer DFS de no ser conexo sera 0-Conexo. Para probar el resto de combinaciones para saber la K-Conexidad usara 3 for, el primero para probar si es 2-Conexo, el segundo para probar 3-Conexo y el tercero para probar 4-Conexo, para poder ir limitando la K-Conexidad usara una variable llamada `Lock` que se sabe si durante las combinaciones se encuentra un nivel de k-conexidad inferior, entonces no es necesario continuar probando combinaciones para niveles de k-conexidad superiores. Esto se debe a que, una vez detectada una k-conexidad inferior, es imposible que el grafo sea más k-conexo. Por eso la variable `Lock` bloqueara el for, evitando pruebas adicionales innecesarias.

Significado de los Valores de K

- $K = 0$: El grafo no es conexo.
- $K = 1$: El grafo es conexo (1-conexo).
- $K = 2$: El grafo sigue siendo conexo después de eliminar un vértice (2-conexo).
- $K = 3$: El grafo es 3-conexo, ya que sigue siendo conexo tras eliminar cualquier par de vértices.
- $K = 4$: El grafo es 4-conexo, permaneciendo conexo incluso tras eliminar tres vértices distintos.

Conexidad básica (1-conexidad):

- La primera vez que se llama a `esConexo()`, se verifica si el grafo es **1-conexo**.
- Esto se realiza desde un vértice cualquiera con una búsqueda en profundidad (DFS). Si el grafo no es conexo, se considera **0-conexo** ($K = 0$), y no es necesario continuar con las pruebas.

2-Conexidad (Eliminar 1 vértice):

- Si el grafo es conexo, se procede a verificar si es **2-conexo**.
- Para esto, se usa el primer ciclo `for (int i = 0; i < V && Lock < 1; i++)`, en el cual se "elimina" cada vértice individualmente (marcándolo como visitado) y se llama a `esConexo()` para ver si el grafo sigue siendo conexo sin ese vértice.
- Si el grafo se desconecta tras eliminar un vértice, entonces es **1-conexo** ($K = 1$), y se detiene la prueba de niveles mayores con `Lock = 1`. Si sigue siendo conexo, se continúa al siguiente nivel.

3-Conexidad (Eliminar 2 vértices):

- Si el grafo es 2-conexo, se verifica si es **3-conexo**.
- En este paso, el segundo `for (int j = 0; j < V && Lock < 2; j++)` elimina pares de vértices (los vértices i y j), y se llama nuevamente a `esConexo()` para comprobar si el grafo sigue siendo conexo.
- Si el grafo se desconecta al eliminar dos vértices, es 2-conexo ($K = 2$), y se detiene el proceso para niveles superiores con `Lock = 2`. Si sigue siendo conexo, se continúa probando para el siguiente nivel de *k-conexidad*.

4-Conexidad (Eliminar 3 vértices):

- Si el grafo es 3-conexo, se verifica si es **4-conexo**.
- El tercer `for (int k = 0; k < V && Lock < 3; k++)` elimina tres vértices (i , j , y k), y se llama a `esConexo()` para comprobar la conectividad.
- Si el grafo se desconecta tras eliminar tres vértices, es 3-conexo ($K = 3$), y se detiene el proceso con `Lock = 3`. Si sigue siendo conexo, se considera 4-conexo ($K = 4$).

Una vez identificada la **K-conexidad** se libera la memoria reservada con `malloc` ya que se obtuvieron todos los datos.

Ahora el programa le preguntará al usuario que quiere saber del grafo, el usuario tiene 6 opciones.

1. **Grado Maximo:** El programa mostrará en la terminal el resultado obtenido de `GradMax`.
2. **Grado Minimo:** El programa mostrará en la terminal el resultado obtenido de `GradMin`.
3. **Conexidad:** El programa mostrará en la terminal si el grafo es conexo o es desconexo, esto basado en el valor de K ya que si ($K \geq 1$) se sabe que el grafo es conexo, el caso de 0 vértices lo tomará conexo por temas convencionales pero este no tiene conectividad.
4. **Conectividad:** El programa mostrará en la terminal la **K-conexidad** del grafo
5. **Cerrar Grafo:** Volverá al menú principal y preguntará al usuario nuevamente si quiere ingresar otro grafo o salir
6. **Salir:** Cerrará el programa

En caso de no elegir ninguna lanzará una excepción.

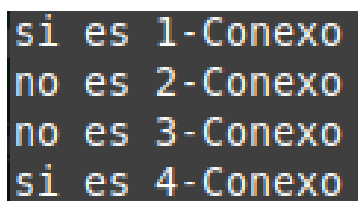
Desafios

El desarrollo del código presento diversos desafíos, desde errores de memoria hasta analisis, pero lograron ser superados con exito a continuacion se mencionaran algunos de estos desafios:

- **Obtencion de conexiones:** Este fue un desafío bastante problemático, ya que el método con el que se trabajó no era muy conocido por el equipo. La función `strtok()` generó conflictos, principalmente relacionados con la memoria. La razón de esto es que `strtok()`, como se explicó anteriormente, separa una cadena de caracteres en partes según un delimitador. Sin embargo, antes de empezar a extraer los datos, necesita saber desde dónde comenzar a separar. Para este propósito, se utilizó la función `strchr()`, que permitía determinar el punto de inicio para `strtok()`, en este caso el primer espacio (" ").

El problema surgía cuando se trabajaba con vértices sin conexiones. Al no encontrar un espacio que separara el número del vértice de sus conexiones, `strchr()` devolvía `NULL`, lo que generaba errores de memoria. Este problema se resolvió añadiendo un condicional que verificaba si el resultado de `strchr()` era distinto de `NULL`, de manera que el caso se omitiera.

- **Analisis de Grafos de orden menor a 4:** El algoritmo utilizado para analizar la K-Conexidad tuvo 2 aproximaciones, de las cuales la primera no producía los resultados esperados. La primera aproximación mostraba las 4-Conexidades e indicaba si los grafos cumplían o no con la K-Conexidad. Sin embargo, las variables empleadas eran de tipo `char` y se inicializaban con "si", lo que señalaba que cumplían con la K-Conexidad correspondiente. El problema surgió al trabajar con grafos de orden menor a 4, ya que los bucles `for` estaban condicionados a ejecutarse únicamente si el grafo tenía un orden específico. En este caso, el último bucle, encargado de analizar las combinaciones de 4-Conexo, no se ejecutaba, por lo que la cadena de caracteres "si" nunca se actualizaba a "no". Por ejemplo, si se tenía un grafo de 3 vértices y era 1-Conexo, el programa mostraba lo siguiente:



```
si es 1-Conexo
no es 2-Conexo
no es 3-Conexo
si es 4-Conexo
```

Lo cual no tenia ningun sentido segun la definicion de K-Conexidad

- **Menu :** Cuando se implementó el algoritmo del menú, el tipo de entrada que recibía el programa era de tipo `int`, por lo que el usuario debía ingresar obligatoriamente un número al elegir una opción. El problema surgía cuando el usuario ingresaba una cadena de caracteres en lugar de un número. En ese caso, la variable `Opcion1` retenía ese valor no válido, y como el programa estaba en un bucle infinito `while(1)`, quedaba atrapado.

Esto sucedía porque, al intentar ingresar un valor no entero en una entrada que esperaba un `int`, las condicionales existentes no podían manejar ese caso, ni siquiera el bloque `else`. Como resultado, el programa se quedaba bloqueado y el usuario no podía continuar, esto sucedia analogamente para la `Opcion2`.

Experimentación y testing

Para la experimentación y testing del algoritmo desarrollado, se planteará una serie de hipótesis a comprobar, más específicamente 3, las cuales estarán relacionadas con el tiempo de ejecución, tamaño del grafo (nodos o aristas) y características de los grafos como vértices de corte, caminos, entre otras que serán nombradas en el desarrollo de la misma. **Los grafos asociados a cada hipótesis se encuentran en carpetas con su nombre distintivo, se deben mover a la carpeta raíz para ejecutar.**

Hipótesis 1:

La existencia de puentes y puntos de corte en un grafo disminuye de manera considerable la k -conectividad del grafo y lo limita. Esto debido a que los puentes y vértices de corte restringen las rutas alternativas entre los nodos, lo que puede provocar que una falla en un nodo o en una arista que desconecte el grafo.

Se examinarán diferentes grafos que tengan vértices de corte y sin vértices de corte para observar cómo su presencia afecta la k -conectividad, usaremos una lista de grafos completos, estrella y de camino, el grafo completo demuestra que la no presencia de vértices de corte indica máxima conectividad, mientras que estrella y camino que cuentan con muchos vértices de corte lo contrario, para ello mostraremos una tabla con la conectividad de cada grafo.

1. Completo: tamaño 5, 10, 15, 20
2. Estrella: tamaño 5, 10, 15, 20
3. Camino: tamaño 5, 10, 15, 20

Nombre del Grafo	k -conectividad
Camino de 5 nodos	1
Estrella de 5 nodos	1
Completo K_5	5
Camino de 10 nodos	1
Estrella de 10 nodos	1
Completo K_{10}	10
Camino de 15 nodos	1
Estrella de 15 nodos	1
Completo K_{15}	15
Camino de 20 nodos	1
Estrella de 20 nodos	1
Completo K_{20}	20

Table 1: Conectividad de los grafos

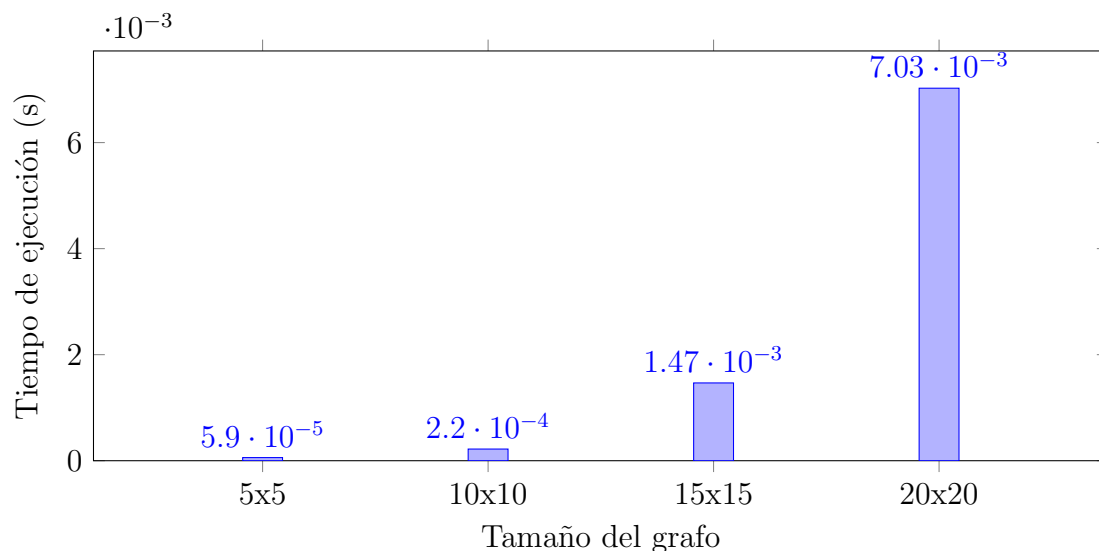
La tabla muestra la k -conectividad de diferentes tipos de grafos para varios tamaños de nodos. Permite concluir que, a medida que un grafo cuenta con vértices de corte o caminos limitados, mayor es la probabilidad de que su k -conectividad esté limitada a el grado de estos vértices críticos.

Hipótesis 2:

Dada la experimentación del código a medida que se fue creando, se hace notar que el tiempo de ejecución en los algoritmos de grafos aumenta a medida que hay una mayor cantidad de nodos y aristas, esto debido a que dependiendo del tamaño del grafo necesita de mas calculos, por ejemplo, el DFS se ejecuta recursivamente hasta recorrer todo el grafo, esto afecta en el tiempo de ejecución a medida que hayan mas nodos y conexiones, por lo que, Suponemos que la cantidad de nodos y aristas afecta en el tiempo de ejecución.

Sea un conjunto de grafos de tamaño variable, se espera observar un crecimiento acelerado en el tiempo de ejecución al aumentarlos. Dado que, en muchos algoritmos de análisis de conectividad o camino, el costo computacional depende de la cantidad de nodos y aristas combinados. En grafos donde el número de aristas crece considerablemente en relación con el número de nodos, es probable que el tiempo de ejecución crezca de manera acelerada, debido al aumento en el número de operaciones necesarias.

Para comprobar esta hipótesis, se realizarán experimentos con grafos de distinto tamaño en específico de tamaño 5, 10, 15 y 20, todos los cuales serán 4-conexos para mantener una estructura consistente y evitar que cambios en la conectividad afecten el tiempo de ejecución. Utilizando una función específica de medición de tiempo en el código.



El diagrama presenta el tiempo de ejecución para los diferentes tamaños de grafos, evidenciando un incremento que podría ser exponencial a medida que se aumentan los nodos y las aristas en los grafos. Basándonos en estos datos, podemos deducir que, en grafos de gran tamaño y alta conectividad, el tiempo de ejecución suele incrementarse de manera exponencial debido al crecimiento en el número de operaciones requeridas para examinar sus características de conectividad.

Hipotesis 3:

Mientras mas cantidad de ciclos a pequeña escala es mayor la probabilidad de que el grafo sea mas robusto, ya que estos ciclos proporcionan múltiples rutas alternativas entre nodos, lo que aumenta la capacidad del grafo para mantener su conectividad incluso ante fallos de nodos o aristas.

Los ciclos pequeños son fundamentales para la conectividad en grafos, ya que proporcionan redundancia en los caminos entre los nodos, lo que hace que el grafo sea más resistente a fallos. Al existir múltiples caminos entre los nodos debido a estos ciclos, un grafo con muchos ciclos pequeños tendrá una mayor tolerancia a la desconexión, lo que se traduce en una mayor robustez.

Para ello, se comprobara usando grafos de diferentes tamaños y con muchos ciclos y otros con menor cantidad para comprobar su conexividad, como ejemplo usaremos los siguientes grafos.

1. Completo: tamaño 5, 10, 15, 20
2. Aciclico: tamaño 5, 10, 15, 20
3. Entre 2 y 3 ciclos: tamaño 5, 10, 15, 20

Tipo de Grafo	Tamaño del Grafo	k -Conectividad
Grafo Completo	5 nodos	5
Grafo Completo	10 nodos	10
Grafo Completo	15 nodos	15
Grafo Completo	20 nodos	20
Grafo Acíclico	5 nodos	1
Grafo Acíclico	10 nodos	1
Grafo Acíclico	15 nodos	1
Grafo Acíclico	20 nodos	1
Grafo con 2 Ciclos	5 nodos	2
Grafo con 2 Ciclos	10 nodos	2
Grafo con 3 Ciclos	15 nodos	1
Grafo con 3 Ciclos	20 nodos	1

Table 2: Tabla de k -conectividad de los grafos

Dada la tabla solo podemos concluir que mientras mas ciclos tenga mayor es su k -conexividad, dado que los grafos completos cuentan con mayor cantidad de ciclos que los otros, los conforme aumenta la cantidad de ciclos pequeños en un grafo, su robustez aumente debido a la redundancia de caminos, lo que permite que el grafo mantenga su conectividad incluso si fallan algunos nodos o aristas. Para poder concluir que afecta en general a todos los grafos los ciclos deben ser acordes al tamaño del grafo, ya que. un grafo con 3 ciclos de tamaño 20 no es similar a un grafo con 3 ciclos de tamaño 7.

Conclusiones

En conclusión, se puede determinar que, gracias a la experimentación realizada en diversos casos de grafos, se ha comprobado que la k -conectividad es un factor clave para evaluar la robustez. La presencia de puentes y vértices de corte en la estructura de un grafo afecta significativamente su capacidad para mantener la conectividad en caso de fallos. A medida que se aumenta el tamaño de los grafos, los tiempos de ejecución de las pruebas crecen, lo cual genera un desafío en términos de tiempo y recursos de cómputo. Esto plantea una dificultad al analizar redes de gran escala, como una red eléctrica nacional, donde las pruebas de conectividad resultan en tiempos de ejecución elevados y un costo asociado considerable.

El problema de la red eléctrica se complica debido a estos factores, ya que comprobar la k -conectividad en grafos grandes requiere no solo de tiempo, sino también de una inversión económica importante en infraestructura computacional y en recursos humanos especializados. En este contexto, optimizar los algoritmos y técnicas de análisis de conectividad se convierte en una necesidad para realizar estudios de gran escala con mayor eficiencia.

La importancia de la k -conectividad en la actualidad es fundamental, especialmente en el diseño y mantenimiento de infraestructuras críticas, como redes de transporte de energía y comunicaciones. Una alta k -conectividad en estas redes garantiza redundancia y resiliencia frente a fallos, lo que resulta esencial para minimizar el riesgo de apagones y fallas masivas. En conclusión, el análisis de k -conectividad no solo contribuye a mejorar la estabilidad de una red eléctrica, sino que también promueve un enfoque más seguro y robusto en la planificación de infraestructuras complejas que dependen de una conectividad fiable.

Anexo

Repositorio donde se subio el codigo (github).