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

Eliminar todos los nodos en S y sus aristas incidentes de G
Seleccionar un nodo $v \in V \setminus S$ como nodo inicial (si existe)
Inicializar todos los nodos en $V \setminus S$ como no visitados
DFS(G, nodo u)
Marcar u como visitado
for cada vecino w de u en G **do**
  if w no ha sido visitado **then**
    DFS(G, w)
  end if
end for
Llamar a DFS(G, v) para explorar el grafo
for cada nodo $x \in V \setminus S$ **do**
  if x no ha sido visitado **then**
    return 1
  end if
end for
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 tenemos que G es disconexo, ya que no existe un camino entre u y v.

Ahora, según el Teorema de Menger, no deberían existir más de k caminos disjuntos. Procedemos por contradicción, suponiendo que $\{C_1, \ldots, C_j\}$ 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 más k de los j caminos. Como $j > k$, algún camino no ha sido modificado, por lo 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.

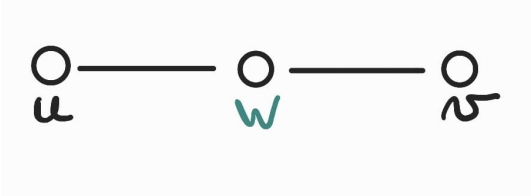
Base $n = 0$

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

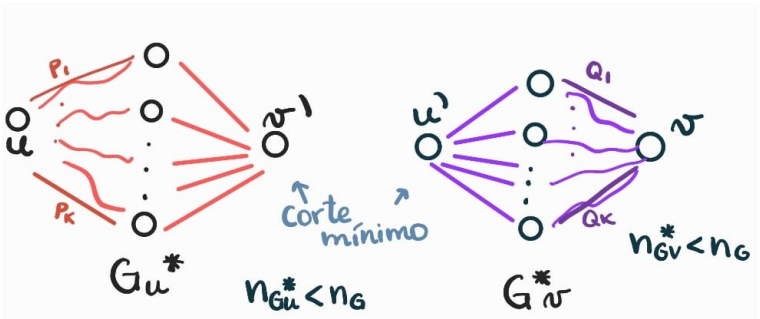
Hipótesis de inducción

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, $n_G - 1 < n_G$ y se cumple el teorema por hipótesis de inducción.

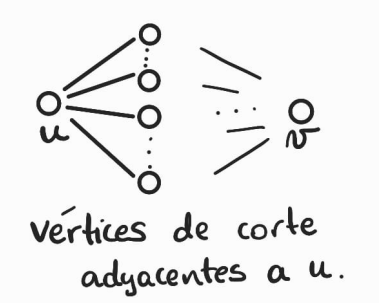


Caso 2: Existe $V' = \{w_1, \ldots, w_k\}$ tal que hay algún vértice no adyacente a v y algún vértice no adyacente a u. Sea $G_u \setminus \{v\}$ el grafo de todos los caminos que inician en u y terminan en w_i sin pasar por otro w, y $G_v \setminus \{u\}$ su análogo. Si existen $\{P_1, \ldots, P_k\}$ caminos de u a w_i y creamos un vértice v', tendremos i caminos entre u y v', lo mismo para v con un u' conectados por caminos $\{Q_1, \ldots, Q_k\}$. 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 $\{P_1Q_1, P_2Q_2, \ldots, P_kQ_k\}$ (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 $C = (u, x, y, \ldots, v)$ 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 tanto, todos los vértices en V'' son vecinos de u. $V'' \neq \emptyset$ con $k \geq 2$, y no son vecinos de v. $V'' \cup \{y\}$ es un u-v-corte mínimo en G, entonces hay vértices de este conjunto vecinos a u y por hipótesis todos lo son; en particular, y es adyacente a u, formando una contradicción ya que, según C, la existencia de x implicaría que habría dos caminos entre u e y, lo cual no es posible ya que C es geodésica.



Conclusión

Hemos 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 para 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, permitiéndonos diseñar redes de tal manera que puedan soportar fallos sin perder conectividad.

Teorema Dinic

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

Este algoritmo tiene 3 partes principales, la creación de un grafo de nivel, búsqueda de bloqueos de flujo, y la construcción de un grafo residual, que se desarrollan en ese orden, y permiten encontrar la capacidad, o flujo, máximo de un grafo.

Para la primera parte de la construcción del grafo de nivel lo que se tiene que hacer es buscar todos los vértices que están a distancia n + 1, con n el nivel del grafo actual. Para ello, hay que posicionarse en el vértice fuente del grafo, y ver cuáles otros vértices puedo llegar, marcarlos como de nivel 1, y eliminar todas las aristas que conecten a dos vértices del mismo nivel. Así se sigue con el resto del grafo, dejando así un grafo con caminos entre el vértice fuente y sumidero de largo mínimo. Esto se logra con un BFS, que a su vez va marcando que aristas pasan de un nivel al siguiente, y de qué nivel es cada vértice.

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

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

Luego, el algoritmo ve los caminos que llegan al vértice sumidero, buscando los caminos directos que contemplen la mayor cantidad de flujo, esto con un DFS, que recorre el grafo de nivel enviando la mayor cantidad de flujo posible por las aristas, y avanzando por caminos hasta llenar todas las aristas con su respectivo flujo máximo, y suma el total de flujo final, cantidad referenciada como bloqueo de flujo del grafo. En este proceso, el algoritmo marca dichas aristas que se han llenado al dejar pasar el flujo, para luego, al momento de buscar el flujo residual, no vuelva a usar las mismas aristas ya que no pueden llevar más de lo que su máximo lo indica, y a su vez marca que cantidad de flujo se dejó pasar por las aristas que no se llenaron.

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

Por último, queda hacer el grafo de flujo residual, que se toma del grafo inicial, pero tiene bloqueadas las aristas ya llenas por el paso anterior, y las que no se llenaron se reduce la capacidad en lo que pasaron la vez anterior. Luego de armado este grafo se busca un camino que pueda llegar de la fuente al sumidero, y en caso de ser posible se le suma su capacidad al bloqueo de flujo obtenido anteriormente, proceso que se repite hasta que

el vértice sumidero no se puede alcanzar desde la fuente, dando así 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 Minimo

El algoritmo de Dinic también nos puede ayudar a encontrar el corte mínimo de un grafo, ya que una vez obtenida la red residual nos podemos dar cuenta de cuáles son los vértices que no son alcanzables desde la fuente, y los que sí lo son, y el corte mínimo termina siendo la suma de las capacidades restantes de las aristas que llegan a esos vértices desde los vértices no alcanzados desde la fuente. Dando así el mismo valor que el flujo máximo.

Relación Conectividad

Finalmente, una de las utilidades del corte mínimo de un grafo es que nos puede ayudar para conocer la conectividad de un grafo aplicando el algoritmo de Dinic. Esto se debe a que el corte mínimo nos indica el flujo mínimo que se debe cortar para desconectar los vértices fuente y sumidero del grafo, de esta forma, si tenemos las capacidades del grafo y su grafo residual generado por el algoritmo al momento de obtener el corte mínimo, podemos saber cuáles son las aristas que tenemos que eliminar antes de que el grafo deje de ser conexo y ya no se pueda llegar de la fuente al sumidero. Y ahora aplicándolo a nuestra problemática de eliminación de vértices tenemos que al eliminar los vértices que contienen a las aristas de este corte mínimo perderíamos la conexidad en nuestro grafo, por lo que sabemos cuántos y cuales vértices nos impiden limitan la conectividad.

Aclaraciones en el proyecto

Durante el desarrollo de pseudocódigo nos dimos cuenta de que el algoritmo de Dinic implementado en la búsqueda de la conectividad nos permite encontrar la conectividad de una manera muy simple para los grafos a los cuales está planteado el algoritmo. Esto es que, si tenemos un grafo al que ponemos como capacidad en cada arista 1, nos damos cuenta de que no es necesario hacer el grafo residual, porque las aristas no tienen capacidad residual pues quedan saturadas, y todos los caminos que salen o llegan al vértice sumidero no pueden repetirse.

Dicho esto, notamos que el corte mínimo, al ser igual que el flujo máximo, el cual va a estar 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 disyuntos posibles entonces el flujo máximo va a ser dicho valor, ya que no puedes generar más caminos que induzcan a una mayor conectividad, y al eliminar los vértices que generan

```
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 []
```

dichas vecindades el grafo quedara disconexo. El segundo caso se da ya que si tenemos alguna arista puente que divide dos secciones del grafo, al momento de pasar por dicha arista con el algoritmo de Dinic, solo vamos a poder pasar una vez, ya que se va ver saturado después de eso, por lo que el flujo máximo sería 1. Y finalmente, el grado mínimo de un grafo puede darnos el valor del flujo máximo, ya que para alguna de las posibles orientaciones que le damis a los vértices vamos a obtener como flujo máximo dicho grado, y al igual que en el primer punto quedaría 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 Completo

Algorithm 2 Calcular la conectividad del grafo usando el algoritmo de Dinic

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

Para cada arista a **en** E:
    Establecer la capacidad de las aristas del grafo a 1
    Esto permite hacer un **Dinic** que entregue el valor exacto de la conectividad
$flujo\_max\_minimo \leftarrow 0$
$combinaciones\_usadas \leftarrow [*]$
for cada $vertice1$ **en** V **do**
  for cada $vertice2$ **en** V **do**
    if $vertice1 \neq vertice2$ **y** $(vertice1, vertice2) \notin combinaciones\_usadas$ **then**
      Asegurarse que se han visitado todos los vértices
      Evitar combinaciones ya probadas
      $vertice1 \leftarrow s$ *(Se establece vertice1 como fuente)*
      $vertice2 \leftarrow t$ *(Se establece vertice2 como sumidero)*
      $G' \leftarrow$ BFS_en_nivel(G) *(Transformamos el grafo en uno en niveles con un BFS, eliminando las aristas que no sirven)*
      $flujo\_max \leftarrow$ encontrar_flujo_maximo_inicial(G')
      Mientras haya caminos desde s a t:
        Marcar los nodos s y t
        Usar DFS para recorrer los caminos
        Eliminar los nodos bloqueados en la última iteración
      $Gr \leftarrow$ transformar_a_grafo_residual(G') *(Transformamos G' en un grafo residual, marcando y buscando aristas bloqueadas)*
      Obtener $caminos\_finales(Gr)$ *(Buscar los caminos entre s y t que no se encontraron en la primera iteración)*
      if $flujo\_max < flujo\_max\_minimo$ **then**
        $flujo\_max\_minimo \leftarrow flujo\_max$ *(Guardar el mínimo valor de todos los flujos máximos)*
      end if
      $combinaciones\_usadas \leftarrow (vertice1, vertice2)$
    end if
  end for
end for
Imprimir "La conectividad del grafo obtenida a partir del algoritmo de Dinic es de $flujo\_max\_minimo$"
=0

Aspectos Técnicos

El análisis y la ejecución de los experimentos fueron realizados en dos plataformas diferentes: Windows 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: _____
  - Núcleos: _____
  - Hilos: _____
  - Modelo: _____
  - RAM: _____
  - Sistema Operativo: Windows _____ (versión _____)
  - Compilador utilizado: MinGW (versión _____)
- **Plataforma Linux Mint:**
  - Arquitectura: _____
  - Núcleos: _____
  - Hilos: _____
  - Modelo: _____
  - RAM: _____
  - Sistema Operativo: Linux Mint _____ (versión _____)
  - Compilador utilizado: GCC (versión _____)

Descripción Datos

Implementación de Codigo

Experimentación y testing

Conclusiones