Universidad de Concepción
Facultad de Ingeniería
Departamento de ingeniería informática

Matemáticas Discretas
Análisis de conectividad en grafos

INTEGRANTES
Chloe Yañez Lavin
Francisca Nuñez Larenas
Emily Gaete Bobadilla

Noviembre 2024

Tabla de Contenidos

1 Problemática

1.1 Descripción

El presente análisis surge a raíz de la necesidad de estudiar la k-conexidad de grafos y su aplicación en problemáticas reales. En este caso, se enfoca específicamente en la infraestructura eléctrica de Chile, la cual ha sufrido los efectos de los recientes eventos climáticos en el país. El estudio de la k-conexidad se presenta como una herramienta clave para evaluar la robustez de dichas redes eléctricas.

Antes de comenzar, es importante definir la k-conexidad. Sea $G = (V, E)$ un grafo, se llama k-conexo, $k \in \mathbb{N}$, si:

1. $|V| > k$ y
2. para todo conjunto de vértices $X \subseteq V$ tal que $X < k$ se cumple que $G - X$ es conexo.

El máximo k tal que G es k-conexo se denomina **conectividad** de G.La conectividad mide cuántos vértices pueden ser eliminados de un grafo antes de que deje de ser conexo.

Un concepto importante que se mencionó fue la **robustez**. Cuando se habla de esta propiedad en grafos, nos referimos a la tolerancia a la eliminación aleatoria de alguno de los nodos de este, en términos de redes, sería la falla aleatoria de los componentes de la red, ya sea esto por sobrecarga de alguno de los mismos, pérdida de energía, etc.

Los objetivos principales de este análisis son el estudio de la k-conexidad y la conectividad y, mostrar por qué el estudio de la conectividad de grafos es relevante y útil para la simulación de la resistencia de la red eléctrica frente a fallos en varios nodos y, de este modo, la búsqueda de soluciones en este ámbito a problemas de redes.

1.2 Análisis teórico

Para proceder a analizar la problemática y la definición presentada, se utilizarán tres diferentes enfoques para abordar la conectividad como procede:

1.2.1 Algoritmo basado en eliminación de vértices:

Para determinar la k-conectividad de un grafo $G = (V, E)$ se puede implementar un algoritmo basado en la eliminación de distintas combinaciones sus vértices. Para esto el algoritmo comienza con combinaciones de 0 vértices eliminados hasta combinaciones de $|V| - 1$, siendo este el máximo de vértices que pueden ser eliminados.

El algoritmo comienza con $k = 0$ y luego va eliminado k vértices por cada ciclo. Luego por cada ciclo el algoritmo genera todas las combinaciones de $G-(V', E')$, donde $|V'| = |V| - k$. A continuación se realiza una búsqueda DFS (Depth-first search) por cada combinación $G' = G - (V', E')$ para determinar la conexidad del grafo generado.

Cuando se finaliza la búsqueda se verifica si se visitaron todos los vértices en el grafo G'. Si los nodos visitados son iguales a los nodos totales del grafo, entonces el grafo es conexo y se continua con la siguiente combinación del ciclo. Al finalizar un ciclo si todas las combinaciones son verificadas como conexas se continua con $k = k + 1$ y se vuelve a eliminar n vértices de G.

El algoritmo se detiene cuando encuentra una combinación de k vértices cuya eliminación hace que el grafo G' sea disconexo. El valor de k representa el mínimo número de vértices necesarios para desconectar el grafo, y se establece como el valor de conectividad del grafo.

Algorithm 1: Connectivity using Vertex Elimination

```
   Input: Graph G = (V, E)
   Output: Connectivity k of the graph
 1 k ← 0;
 2 for 0 to |V| − 1 ∈ G do
 3 |   |V'| = |V| − k;
 4 |   for each combination of G − (V', E') possible do
 5 |   |   G' = G − (V', E');
 6 |   |   reachable_nodes ← DSF_Algorithm(G');
 7 |   |   if reachable_nodes is not equal to V' then
 8 |   |   |   return k;
 9 |   k ← k + 1;
10 return k;
```

Algorithm 2: DFS Algorithm

```
   Input: graph: The graph on which to perform the search.
   start_node: The node to start the search from.
   Output: Mark all nodes reachable from start_node.
 1 Function DFS(graph, start_node):
 2 |   visited ← array of size n with all values set to False;
 3 |   stack ← [];
 4 |   visited[start_node] ← True;
 5 |   stack.push(start_node);
 6 |   while stack ≠ [] do
 7 |   |   current_node ← stack.pop();
 8 |   |   foreach neighbor ∈ graph[current_node] do
 9 |   |   |   if visited[neighbor] == False then
10 |   |   |   |   visited[neighbor] ← True;
11 |   |   |   |   stack.push(neighbor);
```

1.2.2 Teorema de Menger

El Teorema de Menger establece que si, sea $G = (V, E)$ grafo finito y se tienen u y v dos vértices no adyacentes tal que $u, v \in V$, entonces el menor cardinal de un conjunto de vértices que separan a u de v (conjunto de Corte Menor) es igual a la cantidad máxima de caminos internamente nodo-disyuntos de u a v en G.

Una cantidad de caminos internamente nodo-disyuntos son caminos entre un par de vértices u y v que no comparten ningún vértice intermedio, es decir, solo comparten el vértice con el que comenzaron y con el que terminaron.

Recordemos el concepto de la conectividad k de un grafo G, esta indica que necesitamos una cantidad de al menos k para desconectar cualquier par de vértices en G, en paralelo, Menger establece que la menor cantidad de vértices que se deben eliminar para desconectar dos vértices no adyacentes en G es igual al número máximo de caminos internamente nodo-disyuntos entre estos.

Sea $G = (V, E)$ un grafo k-conexo, es decir, que para cualquier vértice $u, v \in V$, se requiere eliminar al menos k vértices para desconectarlos y por ende su conectividad es k. Al ser grafo, cumple el Teorema de Menger.

Supongamos que, por reducción al absurdo, existen un par de vértices u y v en G grafo k-conexo, para los cuales no existen k caminos nodo-disyuntos entre sí.

Utilizando el Teorema de Menger, si existen menos de k caminos nodo-disyuntos entre u y v, entonces existe un conjunto de corte mínimo de menos de k vértices que, al eliminarlos, desconectan u de v. Sin embargo, si recordamos la definición de conectividad, este establecía la necesidad de al menos k vértices para desconectar un grafo, separando a u de v cualquier par de vértices en el grafo, llegando a una contradicción.

Por lo tanto, podemos concluir que la conectividad k de un grafo cualquiera garantiza la existencia de k caminos internamente nodo-disyuntos entre cada par de vértices.

Para concluir, en **Algorithm 3** se puede observar un pseudocódigo para establecer la conectividad k de un grafo G. Notemos también que en la línea 9 de **Algorithm 3**, se hace uso del algoritmo BFS, del cual se muestra su pseudocódigo, ajustado a las necesidades del problema, en **Algorithm 4**.

El pseudocódigo realiza un análisis a cada par de vértices u y v en G, realizándose una copia del grafo G, al cual iremos modificando la capacidad de las aristas, siendo esto necesario para el algoritmo de flujo máximo (Algoritmo de Edmonds-Karp), el cual es utilizado al momento de contar los caminos disjuntos entre u y v a continuación: Para el algoritmo de flujo máximo se utiliza BFS. Al

encontrar caminos en aumento, se actualizan las capacidades de las aristas en el camino encontrado y, a su vez, incrementando las capacidades de las aristas inversas. Luego se incrementa el contador de caminos disjuntos. Finalmente se actualiza la conectividad mínima. Esto se repite hasta llegar a k que representará la conectividad k del grafo.

Algorithm 3: Connectivity using Menger's Theorem

```
   Input: Graph G = (V, E)
   Output: Connectivity k of the graph
 1 k ← ∞;
 2 for each pair of vertices (u, v) ∈ G do
 3     if u = v then
 4         continue;
 5     E' ← E but with the condition that each edge (u, v) ∈ E' appears exactly once
         (capacity of 1);
 6     G' = (V, E');
 7     CounterDisjointPaths ← 0;
 8     while there exist a path from u to v ∈ G' do
 9         IncreasingPath ← BFS(G', u, v);
10         if IncreasingPath = null then
11             break;
12         CurrentPath ← v;
13         while CurrentPath ≠ u do
14             PredessesorNode ← predecessor[CurrentPath];
15             Decrease the capacity of edge (PredecessorNode, CurrentPath) by 1;
16             Decrease the capacity of the reverse edge (CurrentPath, PredecessorNode)
                 by 1;
17             CurrentPath ← PredecessorNode;
18         CountDisjointPaths ← CountDisjointPaths + 1;
19     MinimumCut ← CountDisjointsPaths;
20     k ← min(k, MinimumCut);
21     if k = 1 then
22         return k;
23 return k;
```

Algorithm 4: BFS's pseudocode for Algorithm 3

```
   Input: Residual Graph G', vertices u and v
   Output: Predecessor list representing an augmenting path from u to v, or null if no such
           path exists
 1 Create a queue Q and enqueue u;
 2 Create a predecessor array predecessor[x], initially set to null;
 3 predecessor[u] ← −1;
 4 Create a visited array visited[x], initially set to False;
 5 Create a distance array distance[x], initially set to ∞;
 6 visited[u] ← True;
 7 distance[u] ← 0;
 8 while Q is not empty do
 9     Dequeue u from Q;
10     for each neighbor v of u in R do
11         if visited[v] = False and capacity(u, v) > 0 then
12             visited[v] ← True;
13             distance[v] ← distance[u] + 1;
14             predecessor[v] ← u;
15             Enqueue v into Q;
16 if visited[v] = False then
17     return null;
18 Return predecessor;
```

1.2.3 Algoritmo de Dinic:

El algoritmo de Dinic es utilizado para el cómputo del problema de flujo máximo en redes de flujo, calculando la máxima cantidad de flujo que puede circular desde un nodo de origen (fuente) hasta uno de destino (sumidero), donde cada arista tiene una capacidad de flujo.

- Pasos del algoritmo:
  1. Inicialización
     - El flujo en cada arista parte en 0.
     - La variable de flujo total parte en 0.
     - Construir una red residual, que parte igual al grafo de flujo original
  2. Construcción del Red de Niveles
     - Se etiqueta cada nodo con su distancia mínima desde la fuente de la red usando el algoritmo de Búsqueda por Anchura (BFS)
     - Las aristas con capacidad residual menor o igual a 0 o que conectan nodos con la misma distancia son ignoradas.
     - Si no es posible llegar al destino, el algoritmo finaliza.
  3. Búsqueda de Caminos Aumentables
     - Utilizando el algoritmo de Búsqueda por Profundidad (DFS) en la red de niveles se buscan los caminos aumentables desde el origen al destino.
     - Se envía la capacidad máxima para los caminos encontrados, con respecto a las capacidades de la red residual.
     - El flujo enviado se suma a nuestra variable de flujo total.
     - Se actualiza la red residual reduciendo las capacidades.
  4. Repetir hasta no encontrar caminos aumentables.
  5. Volver al paso 2 para construir una nueva red de niveles
- Pasos para encontrar el corte mínimo:
  1. Las aristas de la red original que conectan a los vértices que están en la red residual con los que no están usando su capacidad máxima, lo que indica que no existe un camino alternativo por donde pueda pasar el flujo (son imprescindibles para la conexidad).
  2. Estos arcos conforman al corte mínimo.

- Conectividad usando Dinic:
    1. Por el teorema de corte mínimo[1], se sabe que el valor de este es igual al del flujo máximo en la red de flujo.
    2. Dado que el conjunto de aristas en el corte mínimo es esencial para la conectividad de la red, podemos construir una red de flujo a partir de un grafo en la cual todas las aristas tienen una capacidad de flujo de 1. Aplicando el algoritmo de Dinic en esta nueva red, obtenemos el valor de flujo máximo, que es igual al valor del corte mínimo. Este valor representa la conectividad del grafo, es decir, el número mínimo de aristas necesarias para desconectarlo.

Algorithm 5: Dinic's Algorithm

```
   Input: Flow Network F
   Output: Maximum Flow k of the Network
 1 foreach edge e in F do
 2     set flow of e to 0 ;
 3 set total flow to 0 ;
 4 create residual network equal to the original flow network ;
 5 while there is a path from source to destination in residual network do
 6     create level network ;
 7     tag every vertex with its distance from the source using BFS ;
 8     ignore edges with residual capacity ≤ 0 or those connecting nodes with the same
         distance ;
 9     if destination is not reachable then
10         stop the algorithm ;
11     while there are increaseable paths from source to destination do
12         perform DFS to find increaseable paths ;
13         set flow to the maximum possible capacity along each path using residual
             capacities in the network ;
14         add the new flow to total flow ;
15         update the residual network by reducing the capacities along the path ;
16 return total flow
```

Algorithm 6: Connectivity using Dinic's Algorithm

```
   Input: Graph G
   Output: Connectivity k of the graph
 1 # Create a flow network F  with edges of capacity 1 from G ;
 2 Create an empty dictionary F ;
 3 foreach vertex u in G do
 4     if u not in F then
 5         F[u] = empty list ;
 6     foreach vertex v in G do
 7         if v not in F then
 8             F[v] = empty list ;
 9         Append (v, 1) to F[u] ;
10         Append (u, 1) to F[v] ;
11 connectivity ← Dinic(F) ;
12 return connectivity
```

References

[1] L. R. Ford and D. R. Fulkerson, "Maximal Flow Through a Network," *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956. doi:10.4153/CJM-1956-045-5

2 Aspectos Técnicos

Las especificaciones de la máquina utilizada para la experimentación son las siguientes:

- **PC:**
  Nombre: ASUS TUF Gaming F15 FX506HC
  OS: WINDOWS 11
- **CPU:**
  Nombre: Intel Core i5-11400H
  Núcleos: 6
  Hilos: 12
  Velocidad base: 2.70 GHZ
  Velocidad incremental: 4.50 GHZ
  Caché L1 Data: 6 x 48 Kbytes
  Caché L1 Inst.: 6 x 32 Kbytes
  Caché L2: 6 x 1.25 Mbytes
  Caché L3: 12 Mbytes
  Tecnología: 10 nm
  Max TDP: 45.0 W
- **MEMORIA:**
  RAM: 12 GBytes
  Reloj: 2660 MHz
  Tipo: DDR4
  Voltaje: 1.20 V
- **GRÁFICA:**
  Nombre: NVIDIA GeForce RTX 3050 Laptop
  GFX Core Clock Base: 1237.0 MHz
  GFX Core Clock Boost: 1500.0 MHz
  Tecnología: 8 nm
  Tamaño memoria: 4 GBytes
  Tipo memoria: GDDR6
  Reloj: 6001.0 MHz
  Ancho de bus: 128 bits
  TDP: 60.0 W

3 Descripción de Datos

3.1 Entrada estándar

La entrada estándar consiste en una dirección de archivo, sin formato específico, el cual es leído al iniciar el programa. El formato del archivo es dividido en dos partes: la primera es un número en la primera línea que indica el orden del grafo, las siguientes líneas contienen el grafo en su formato de lista de adyacencia, es decir, contiene cada uno de los vértices del grafo y, posterior al número del vértice se listan sus vértices adyacentes, separados por una coma. Cabe destacar que los números i de los vértices están definidos tal que sea n la cantidad de vértices $i \in \{1, 2, ..., n\}$.

Ejemplo de entrada:

5	$\leftarrow$ *orden del grafo*
$1:2,3,5$	$\leftarrow$ *lista de adyacencia vértice 1*
$2:1,4$	$\leftarrow$ *lista de adyacencia vértice 2*
$3:1,5$	$\leftarrow$ *lista de adyacencia vértice 3*
$4:2$	$\leftarrow$ *lista de adyacencia vértice 4*
$5:1,3$	$\leftarrow$ *lista de adyacencia vértice 5*

A continuación, se explicará el procesamiento de la entrada estándar y la obtención del grafo mediante este proceso. Cabe destacar que cuando se mencione la memoria, se hace uso de memoria dinámica, puesto que se hizo uso de *malloc* para la asignación de memoria para el grafo y sus nodos:

3.1.1 Estructura para grafo

Para la representación de un grafo cualquiera en C, se implementaron dos structs:

1. **Struct Node:** El cual representa un vértice cualquiera. Este contiene dos variables, la variable *int vertice*, la cual contiene el número de vértice que representa la estructura y la variable *struct Node* next_node*, la cual contiene el vértice adyacente siguiente a este, entendiéndose que los conecta, siendo u nuestro nodo y v un vértice adyacente, un arco de tipo (u, v). Se menciona arco debido a que en el struct se considera la dirección.

2. **Struct Graph:** El cual representa el grafo. Este contiene dos variables, la variable *int num_vertices*, la cual contiene la cantidad de vértices del grafo, es decir su grado y la variable *Node** adjacency_lists*, un puntero doble a *Node*, este es usado para crear una matriz de punteros. Cada índice de esta corresponde a un vértice y apunta a la lista de adyacencia de ese vértice.

3.1.2 Lectura de Archivo e inicialización del Grafo

Al inicializar el programa, se le solicita al usuario que ingrese el nombre de un archivo. Si este no es válido se solicitará constantemente hasta que ingrese un nombre válido de archivo. Obteniendo un archivo válido y, asumiéndose que el archivo contiene el formato correcto de entrada estándar del programa, se toma el número indicado en la primera línea para asumirlo como la cantidad de vértices. Esta cantidad es ingresada como parámetro a la función *initializeGraph()*.

- **initializeGraph():** Función que hace espacio en memoria para el grafo, luego toma el número de vértices ingresado para hacer espacio en memoria para la cantidad de structs de Nodos que se necesitarán. Posteriormente, inicializa las listas de adyacencias para la cantidad de nodos que posee el grafo, listas vacías. Retorna el grafo.

- **readGraph():** Es llamada después de haber inicializado el grafo. Esta función abre el archivo ingresado en modo lectura, extrayendo el contenido de las líneas posteriores a la primera, las cuales recordemos que cada línea es un vértice y sus vértices adyacentes. Se utiliza la función *addEdge()* para añadir la arista que conecta los vértices.

- **addEdge():** Crea las aristas que conectan a los vértices, al tratarse de un grafo, sin dirección, sean u, v los vértices, se tiene que crear tanto la arista (u, v) como la arista (v, u). Para ello se crea un nodo auxiliar con *createNode()* para poder generar la arista en ambos sentidos.

- **createNode():** Es una función bastante sencilla que crea un vértice usando *struct Node*, dándole el número del vértice ingresado como parámetro e inicializando su nodo vecino en *NULL*, luego, retorna el nodo creado.

3.1.3 Liberación de Memoria

Hablar de la liberación de memoria es fundamental si hablamos de memoria dinámica, para ello, se implementó la siguiente función:

- **freeGraph():** Es una función fundamental, como se puede observar, se utilizó mucho *malloc* para la memoria dinámica, por lo tanto la tarea de este método es liberar el espacio creado para las matrices de adyacencias, los nodos y el grafo.

Así mismo, al finalizar el programa, la memoria destinada para el buffer se libera mediante *free(buffer)*.

3.2 Metodología

Para la metodología utilizada a la hora de la generación de documentos de prueba para experimentar la k-conexidad y conectividad de grafos, fue la creación de grafos tanto conexos como disconexos, los cuales cumplen las siguientes características:

- **Grafos Conexos:**
  - **Grafo común con Ciclos:** Es el caso común de grafo, las primeras pruebas se realizaron con este tipo de grafo para verificar el funcionamiento correcto general de la implementación.
  - **Grafo Trivial:** Este grafo es un caso puntual, el cual no se había considerado en un primer momento debido a lo especial que es. Este grafo es 0-conexo y de conectividad 0.
  - **Grafos con Vértices de Corte y/o Puentes:** Grafos relevantes al contar con esta estructura, serán de utilidad en la etapa de experimentación por los grados de sus vértices.
  - **Grafos de orden Alto:** Grafos con alta cantidad de vértices, útiles para probar el límite de los algoritmos implementados para detectar la conexidad, k-conexidad y conectividad.
  - **Grafos de gran Tamaño:** Grafos con altas cantidades de aristas, serán de utilidad en la etapa de experimentación al buscar la relación entre su tamaño y conectividad.
- **Grafos Disconexos:** Para la verificación de conexidad se implementó la función *isConnected()*, la cual internamente posee el algoritmo de búsqueda DFS. Grafos que contengan un sólo vértice aislado fueron vitales para verificar la conexidad misma, ya que es el caso que roza la conexidad ante tener un único vértice aislado. Así mismo, grafos disconexos variados, con grandes cantidades de vértices, para verificar que la función verificadora funcione correctamente.

4 Implementación

A continuación, se describe la implementación del primer enfoque tratado en el análisis de conexidad: mediante un **Algoritmo Basado en Eliminación de Vértices**.

4.1 Descripción del algoritmo

Antes de comenzar, es necesario establecer la existencia de dos tipos de dato que la implementación puede entregar: La conectividad mínima directamente y la k-conexidad de un grafo dado un $k \in \mathbb{N}_0$ señalado. Aclarar que cuando nos referimos a conectividad mínima nos referimos a la cantidad de vértices que son necesarios eliminar para desconectar el grafo, mientras que la k-conexidad nos entregará información acerca de la conexidad del grafo tras eliminar $k-1$ cantidad de vértices. Por lo tanto, el algoritmo tiene esos dos objetivos principales, determinar tanto la **conectividad mínima** como la ***k*-conexidad** de un grafo con un k entregado, es decir, un enfoque de la resistencia general del grafo y para niveles específicos de eliminación de vértices.

La idea del algoritmo es eliminar vértices de forma inteligente, para buscar que el grafo $G = (V, E)$ se vuelva disconexo en la menor cantidad de vértices eliminados. Esto se realizará a través de la búsqueda de combinaciones de vértices, es decir, se probara eliminar diferentes conjuntos de vértices V' tal que $V' \subseteq V$, con $|V| = k$ para luego, verificar si esta eliminación del conjunto afecta a la conectividad del grafo G, es decir, si $G - V'$ sigue siendo conexo.

La conexidad del grafo es verificada mediante un recorrido DFS (Depth First Search). Es decir, comenzamos desde cualquier nodo del grafo que no haya sido eliminado, es decir un nodo de $G - V'$. Si finalizando el recorrido, hay vértices no visitados, esto significaría que $G - V'$ es disconexo. Esto significaría que el conjunto V' es considerado una posible solución para la conectividad mínima o la k-conexidad.

Entonces, si alguna de las combinaciones hace que G sea disconexo, se comparará el valor de k con el valor de una variable auxiliar inicializada en el orden el grafo, es decir $min\_disconnectivity = |V|$. Si k es menor, se actualiza el valor de $min\_disconnectivity$ para establecer que se ha encontrado un conjunto de menor tamaño que desconecta el grafo.

Notar que el proceso descrito es iterativo por cada subconjunto V' debido a la gran cantidad de combinaciones que se pueden llegar a producir dependiendo del orden de un grafo. Finalmente, k es la conectividad de G.

Para la verificación de la k-conexidad para un valor dado de k, el procedimiento no es muy diferente al anterior. De hecho, se realizan combinaciones, pero esta vez de una cantidad de vértices $k-1$. Para verificar la desconexión,

se comprueba directamente si la eliminación de los vértices desconecta el grafo. Si es que esto sucede y el grafo se vuelve disconexo, se indica que el grafo no es k-conexo. En caso que mantenga su conexidad, se dice que el grafo es k-conexo.

Si bien no se ven muchas diferencias en la explicación, en el proceso de codificación, se destacarán aquellas sutilezas en los métodos utilizados que hacen la diferencia en el resultado y, al ejecutar el programa realizado.

4.2 Proceso de codificación

Como se mencionó anteriormente, el grafo quedó almacenado en un struct, en él tenemos los datos necesarios para proseguir. Empezamos el proceso de obtener la conectividad del grafo, para ello, llamamos a la función *connectivity()*:, la cual recibe como parámetro al grafo y retornará finalmente la conectividad k de este.

```
1  int connectivity(Graph* graph) {
2      int min_disconnectivity = graph->num_vertices - 1;
3      int* excluded = (int*)calloc(graph->num_vertices, sizeof(int));
4      for (int k = 1; k <= graph->num_vertices; k++) {
5          int combination[k];
6          combine(graph, excluded, combination, 0, 0, k, &min_disconnectivity);
7          (...)
```

4.2.1 Generación de combinaciones de Vértices a eliminar

Si observamos el código de la función *connectivity()* podemos notar que se inicializa la variable *min_disconnectivity* en el valor $|V| - 1$, esta contendrá la conectividad. Luego se crea un array para almacenar los vértices que serán eliminados *int* excluded*, en un comienzo todos estos empiezan en 0 (no han sido excluidos). Mediante un ciclo for, iteraremos $|V|$ veces, y en estas iteraciones, crearemos un array que contendrá la combinación que se generará mediante la función *combine()*.

```
1  void combine(Graph* graph, int* excluded, int* combination, int start, int idx, int
       k, int* min_disconnectivity) {
2      if (idx == k) {
3          for (int i = 0; i < k; i++) {
4              excluded[combination[i]] = 1;
5          }
6          if (!isConnected(graph, excluded)) {
7              if (k < *min_disconnectivity) {
8                  *min_disconnectivity = k;
9              }
10         }
11         for (int i = 0; i < k; i++) {
12             excluded[combination[i]] = 0;
13         }
14         return;
15     }
16     for (int i = start; i < graph->num_vertices; i++) {
17         combination[idx] = i;
18         combine(graph, excluded, combination, i + 1, idx + 1, k, min_disconnectivity
               );
19     }
20 }
```

La función recibe como parámetros al grafo, el arreglo para almacenar los vértices a eliminar, el array en donde se almacenará la combinación, los dos siguientes enteros son variables que al llamarla por primera vez inician en cero pero, como veremos, esta función es **recursiva**, con lo que el rol de estos enteros es modificar los índices de la combinación, luego el entero k y la variable *min_disconnectivity*.

Si la combinación está completa, se excluirán los nodos al marcarlos en el array de *excluded*, posteriormente se verificará si al quitar esos nodos el grafo es conexo, en caso contrario, se actualizará la variable *min_disconnectivity*. Luego se limpia el arreglo de *excluded* para una siguiente combinación de nodos o cantidad de nodos a eliminar. Notar que en el último ciclo *for*, la función *combine()* se llama a sí misma, generando combinaciones de forma recursiva, variando los nodos para la evaluación de todas las posibilidades.

4.2.2 Verificación de conectividad al eliminar vértices

Recordemos que dentro de *combine()* se hace la verificación de conexidad del grafo tras eliminar los vértices contenidos en *excluded*, esto sucede en la línea 6. En breves palabras, esta función, dependiendo de si se entrega una cantidad de vértices a eliminar calcula la conexidad a un grafo con todos sus vértices (Esto al entregar *NULL* como parámetro) pero en este caso, nos interesa con el arreglo de vértices a eliminar. Para lograr determinar si el grafo es conexo o no, se hace uso del algoritmo DFS el cual hace una búsqueda en profundidad, este ya ha sido mencionado a lo largo del documento, su pseudocódigo puede ser encontrado en secciones anteriores. El algoritmo fue implementado de esta forma:

```
1  void dfs(Graph* graph, int* excluded, int* visited, int start_node) {
2      Stack* stack = createStack(graph->num_vertices);
3      push(stack, start_node);
4      visited[start_node] = 1;
5      while (!isStackEmpty(stack)) {
6          int node_id = pop(stack);
7          Node* neighbor = graph->adjacency_lists[node_id];
8          while (neighbor) {
9              int neighbor_id = neighbor->vertice;
10             if (!visited[neighbor_id] && (excluded == NULL || !excluded[neighbor_id
                   ])) {
11                 visited[neighbor_id] = 1;
12                 push(stack, neighbor_id);
13             }
14             neighbor = neighbor->next_node;
15         }
16     }
17     freeStack(stack);
18 }
```

Se hace uso de una nueva estructura de datos *struct Stack*, la cual representa una pila. Crea la pila mediante *createStack()*, la inicializa con la cantidad $|V|$. Agrega a la pila al primer nodo desde donde realizará la búsqueda, esto con la función *push()*, para luego, marcarlo como visitado en un arreglo auxiliar para almacenar los vértices visitados.

Entra en bucle hasta que la pila esté vacía. Al inicio de cada iteración se obtiene un nodo, mediante *pop()*, para recorrer sus vecinos. Mientras encuentre vecinos y estos no hayan sido visitados ni eliminados, los visitará y añadirá a la pila.

Volviendo a la función *isConnected()*, se revisa la lista de visitados, si esta contiene a todos los vértices del grafo, es conexo, en caso contrario, disconexo.

4.2.3 Cálculo de conectividad Mínima

Ya generadas las combinaciones y verificando la conexidad de estas de forma iterativa, si en alguna iteración *min_connectivity* llega a ser de menor tamaño que $|V|$, entonces se ha encontrado la conectividad. El ciclo *for* se rompe y la función *connectivity()* retorna el valor de la conectividad.

```
    (...)
    }
    for (int k = 1; k <= graph->num_vertices; k++) {
        int combination[k];
        combine(graph, excluded, combination, 0, 0, k, &min_disconnectivity);
        if (min_disconnectivity < graph->num_vertices) {
            break;
        }
    }
    free(excluded);
    return min_disconnectivity;
}
```

4.2.4 Verificación de k-conexidad para un $k \in \mathbb{N}_0$ dado

Se implementó la función *isKConnected*, la cual recibe el valor k ingresado, en caso de que k sea mayor que $|V|$ se dice que no es k-conexo de forma automática. Posterior a esto, como hablamos de k-conexidad, debemos generar combinaciones de largo $k-1$ y debemos eliminar $k-1$ vértices. Para ello, usamos $k\_$ que almacena el valor de $k-1$. Notar también a la variable *is_k_connected*, la cual contendrá 0 o 1, según si es disconexo o conexo respectivamente.

```
int isKConnected(Graph* graph, int k) {
    if (k > graph->num_vertices) {
        return 0;
    }
    int k_ = k-1;
    int* excluded = (int*)calloc(graph->num_vertices, sizeof(int));
    int combination[k_];
    int is_k_connected = 1;
    combineK(graph, excluded, combination, 0, 0, k_, &is_k_connected);
    free(excluded);
    return is_k_connected;
}
```

Llamamos a la función *combineK*(), similar a *combine*(), sin embargo, al ella no calcular la conectividad, no calculamos un *min_disconnectivity*, sino, simplemente hacer combinaciones de largo $k-1$ y probar si el grafo al quitarle esa cantidad de vértices en todas las combinaciones posible sigue siendo conexo. Es decir, marca los vértices a eliminar, verifica conexidad con la combinación actual, limpia el arreglo de los vértices a eliminar, se vuelve a llamar a sí misma.

```
1  void combineK(Graph* graph, int* excluded, int* combination, int start, int idx, int
       k_, int* is_k_connected) {
2      if (*is_k_connected == 0) {
3          return;
4      }
5      if (idx == k_) {
6          for (int i = 0; i < k_; i++) {
7              excluded[combination[i]] = 1;
8          }
9          if (k_ < graph->num_vertices-1) {
10             if (!isConnected(graph, excluded)) {
11                 *is_k_connected = 0;
12             }
13         }
14         for (int i = 0; i < k_; i++) {
15             excluded[combination[i]] = 0;
16         }
17         return;
18     }
19     for (int i = start; i < graph->num_vertices; i++) {
20         combination[idx] = i;
21         combineK(graph, excluded, combination, i + 1, idx + 1, k_, is_k_connected);
22     }
23 }
```

Después del análisis mediante las combinaciones, se retorna *is_k_connected*, como se mencionó anteriormente, 0 si es disconexo y 1 si es conexo. Así, se comprueba la k-conexidad.

4.3 Desafíos

Los desafíos que se presentaron en un comienzo fue la comprensión de la conectividad y k-conexidad, ya que es cuestión de no comprender que la conectividad será la cantidad de vértices mínima que se eliminan para hacer al grafo disconexo y que la k-conexidad tiene como k a ese número que provoca que se vuelva disconexo, es decir, para implementar esta última, había que hablar de un $k-1$.

En segundo lugar, tenemos el algoritmo de generación de combinaciones. Tener que recurrir a una función recursiva fue de las últimas ideas que llegaron al equipo tras plantearse la mejor forma de generar las dichosas combinaciones. Gracias a múltiples casos de prueba y gracias a implementar *printf's* para llevar el seguimiento de las combinaciones, se logró llegar al algoritmo propuesto.

En tercer lugar, tenemos el algoritmo DFS, la implementación de su pila. En un comienzo se implementó sin pila pero esta corría el riesgo de fallar ante grafos de grandes cantidades de datos. Al organizar mejor utilizando la pila, fue más claro poder hacer el seguimiento a los vértices que se iban marcando como visitados.

Y por último, la unificación de los códigos. En un principio se trabajó, por un lado, la lectura de los datos y la creación del grafo con estos datos y, por el otro lado, el proceso de obtención de conectividad. Hacer la traducción de las variables auxiliares usadas a la estructura de datos *Graph* y *Node* fue algo exhaustivo y frustrante.

5 Experimentación

Ya implementado el código y explicada la implementación realizada, es hora de ponerla a prueba. Para ello, se seleccionaron tres parámetros los cuales podrían ser o no relevantes o influyentes para la conectividad de un grafo. Con estos se desarrollaron las tres hipotesis a continuación. En esta etapa, para la obtención de resultados, se utilizarán grafos de orden $n \in \{5, 10, 15, 20\}$.

5.1 Tiempo de Ejecución

Para comenzar, nos hacemos la siguiente pregunta: ¿El tiempo de ejecución de los algoritmos de obtención de conectividad de un grafo G varía según grafo? Y si es así, ¿Gracias a qué parámetros? La primera hipótesis que presentamos es que el tiempo de ejecución de los algoritmos de obtención de conectividad si varia y el motivo de esta variación es el orden del grafo.

Para poder evaluar nuestra hipótesis, primero necesitamos implementar de alguna forma la obtención del tiempo del algoritmo de conectividad. Para ello, se añade la opción de *Obtener promedio en n iteraciones de cálculo de conectividad*. Al seleccionar esta opción, el usuario puede ingresar un $n > 0$ para que el programa realice n iteraciones para ver los diferentes tiempos de ejecución y, además, hacer un cálculo del promedio entre estos tiempos de ejecución.

```
int n = 0;
double promedio = 0;
do {
    printf("Indicar-n mero-de-iteraciones:-");
    scanf("%d",&n);
}
while (n < 0);
printf("Tiempos-de-ejecuci n:-");
    for(int i = 0; i < n; i++){
            clock_t start, end;
            start = clock();
            if (graph->num_vertices != 1) {
                int connectivity_value = connectivity(graph);
            }
            end = clock();
            double time = (double)(end - start) / CLOCKS_PER_SEC;
            promedio += time;
            printf("%f-", time);
        }
promedio /= n;
printf("\nPromedio-de-tiempos:-%f\n", promedio);
break;
```

A continuación, se ponen bajo estudio los casos de prueba, considerando su orden, conectividad y finalmente el tiempo de ejecución. Para ello, se ejecuta el programa definiendo $n = 1000$, esto con el objetivo de obtener un promedio de tiempo de ejecución más preciso. Los datos obtenidos se ven registrados en **Table 1**.

Como se puede apreciar en la tabla, los tiempos de ejecución van en crecimiento a medida que el orden del grafo aumenta. En los grafos de orden 5, los tiempos son similares, tanto en grafos con conectividad 1 que con conectividad 4, esto es debido a que, al ser una cantidad tan pequeña de vértices, aunque

se trate de un K_5, sigue ejecutándolo de forma rápida. Para grafos de orden 10 la situación es similar, pero los tiempos aumentan de lo que eran en orden 5. Al llegar a grafos de orden 15 comenzamos a notar una mayor diferencia en el promedio de tiempos, aparte de que el tiempo en general es mayor a los de orden 10, veamos que Árbol-15 es el que más tiempo le llevó. Notemos que este es el de mayores aristas en el grafo. Y por último, los grafos de orden 20 tienen el mayor tiempo de ejecución, resaltando al grafo Dodecaedro[1] , el cual es el tiempo de ejecución más alto de la tabla.

Gracias a estos datos, podemos concluir que nuestra hipótesis era correcta, que efectivamente el tiempo de ejecución, uno de esos factores es el orden del grafo, mientras más vértices sean, mayor el tiempo. Sin embargo, notemos también que uno de los factores que influyen en el tiempo de ejecución es la cantidad de aristas, puesto que, mientras más aristas, existen más caminos, por ende, dificulta la desconexión del grafo, por lo que puede llevar más tiempo en buscar la combinación para desconectar un grafo de muchas aristas y vértices.

G	$\|V\|$	$\|E\|$	$\delta(G)$	$\Delta(G)$	Conectividad	$\overline{x}$(Tiempo) (seg)
Puente-5	5	8	1	5	1	0.000004
De Corte-5	5	6	2	4	1	0.000003
K_5	5	10	4	4	4	0.000004
Ciclo-5	5	5	2	2	2	0.000005
Puente-10	10	25	4	9	1	0.000277
De Corte-10	10	25	4	9	1	0.000280
K_{10}	10	45	9	9	9	0.000472
Petersen	10	16	3	4	3	0.000277
Puente-15	15	50	6	8	1	0.009383
De Corte-15	15	56	7	14	1	0.012853
K_{15}	15	105	14	14	14	0.024617
Árbol-15	15	14	1	6	1	0.006134
Puente-20	20	91	9	10	1	0.426506
De Corte-20	20	100	9	19	1	0.561192
K_{20}	20	190	19	19	19	1.204770
Dodecaedro	20	30	3	3	3	0.276915

Table 1: Tabla que recopila los datos obtenidos con diferentes tipos de grafos de orden $n \in \{5, 10, 15, 20\}$.
Para visualizar los grafos, redirigirse a la sección de **Anexos**.

[1] Al igual que con el grafo Árbol-15, al probar con 1000 iteraciones el programa termina deteniéndose, por esto en el caso del dodecaedro fueron solo 100 iteraciones. Esto es lo mismo con el grafo K_{20}.

5.2 Tamaño del grafo

¿Es el tamaño de un grafo relevante para la conectividad? ¿Que un grafo posea más aristas hace que este tienda a tener mayor conectividad? La segunda hipótesis que planteamos es que la conectividad de un grafo sí aumenta conforme al valor del tamaño del grafo, sin embargo, dependerá de la distribución de las aristas, del grado mínimo.

Para poner a prueba la veracidad de nuestra hipótesis, necesitamos analizar casos: Grafos con ciclos, estos con diferentes cantidades de ciclos, grafos con puentes y/o vértices de corte y grafos árbol, todos estos grafos deben tener un grado máximo alto, el grado mínimo es la variable que puede ser decisiva a la hora de realizar las pruebas pese al tamaño alto del grafo.

Se ponen bajo estudio los casos de prueba, considerando características de los grafos tales como su orden, tamaño, grado mínimo y máximo del grafo y su conectividad. Los datos obtenidos se ven registrados en **Table 1**.

Como podemos observar en la tabla, la conectividad no es directamente proporcional al tamaño del grafo. Podemos ver que el grafo De corte-20, el cual posee 100 aristas, basta con quitarle un vértice para desconectarlo, a pesar de su gran cantidad de aristas. Esto es debido a la existencia de un vértice de corte, podemos ver un patrón en los grafos árbol, los que tienen puente y los que tienen únicamente vértice de corte, sin importar la cantidad de aristas, vértices o los grados que posean, su conectividad es de uno. Ahora, si únicamente nos fijamos en los grafos K_n, notemos que aquí si es directamente proporcional la conectividad con el tamaño del grafo, esto es debido a la cantidad enorme de aristas que empiezan a tener a medida que n es mayor, ya que empiezan a existir demasiados caminos que, aunque se borre un vértice, no significa mucho ya que aún quedarían muchos caminos por los cuales los vértices pueden conectarse.

Podemos concluir entonces que nuestra hipótesis estaba medianamente correcta, únicamente que el grado mínimo no tiene relevancia en el resultado de la conectividad. Podemos decir que la conectividad es directamente proporcional a la cantidad de vértices y aristas (densidad) que tiene un grafo si y solo si, este no posee vértices de corte.

5.3 Uso de Memoria

¿Cómo se ve afectada la memoria ante el uso de los algoritmos? ¿De qué depende el uso de memoria? ¿Cómo cambia la memoria en la implementación? La tercera hipótesis que planteamos es que la cantidad de vértices influye en la cantidad de memoria que demora el proceso del programa, aumentándola.

Antes de comenzar con la etapa de experimentación, necesitamos instalar la herramienta *valgrind*, la cual permite detectar errores de memoria, leaks, y en general analizar la memoria, de ella necesitamos la herramienta *massif*, esta nos ayudará a ver la distribución del uso de la memoria en el tiempo de ejecución. La instalación se hace mediante la siguiente línea:

```
sudo apt-get install valgrind
```

Ya instalada, utilizamos la siguiente línea de código para ejecutar el código y a su vez, usar *massif*:

```
valgrind --tool=massif ./nombredelprograma
```

Cada vez que se ejecute el programa con *massif*, se generará un archivo con nombre *massif.out.*<*PID*>, donde <*PID*> es el ID del proceso asociado. Para poder visualizarlo, se ejecuta la siguiente línea de código, claramente reemplazando <*PID*> por el ID que se encontrará.

```
ms_print massif.out.<PID>
```

Después de esta explicación, podemos comenzar a referirnos a la experimentación en sí. Para ello, cada archivo se ejecutó y directamente se calculó la conectividad, se cierra el programa y usamos *massif* para obtener los siguientes resultados mostrados en **Table 2**:

G	time(i)	max total(B)	Conectividad
Puente-5	171,644	7,256 bytes	1
De Corte-5	170,755	7,160 bytes	1
K_5	173,738	7,352 bytes	4
Ciclo-5	170,183	7,160 bytes	2
Puente-10	185,767	8,104 bytes	1
De Corte-10	187,367	8,104 bytes	1
K_{10}	203,512	9,064 bytes	9
Petersen	183,488	7,864 bytes	3
Puente-15	209,396	9,352 bytes	1
De Corte-15	214,442	9,640 bytes	1
K_{15}	258,319	11,992 bytes	14
Árbol-15	181,705	7,624 bytes	1
Puente-20	241,351	11,328 bytes	1
De Corte-20	253,977	11,784 bytes	1
K_{20}	340,868	16,104 bytes	19
Dodecaedro	195,790	8,424 bytes	3

Table 2: Tabla que recopila el uso de memoria de diferentes tipos de grafos de orden $n \in \{5, 10, 15, 20\}$.
Para visualizar los grafos, redirigirse a la sección de **Anexos**.

Antes de continuar con el análisis de los datos en la tabla, se explicarán los parámetros utilizados en la tabla:

- **time(i):** Representa el tiempo que ha transcurrido desde que se inicia el programa hasta el momento que se toma la medición de memoria. Esta está medida en milisegundos.
- **total(B):** Indica la memoria utilizada en el instante, esta contempla el *useful-heap(B), extra-heap(B)* y el *stack*.

Además, los datos recopilados en la tabla contemplan el tiempo en que la memoria fue más utilizada, el uso máximo de esta. Ya explicado, analicemos los datos obtenidos. Como es de esperar y, sabiendo de nuestra hipótesis 1, grafos más grandes y complejos necesitan de más tiempo. Como era de esperar, la memoria reacciona de la misma manera, esta tiende a aumentar con el tamaño del grafo y el orden de este, siendo K_{20} el que ocupó más memoria en un punto con 16,104 bytes.

En conclusión, al hipótesis presentada estaba en lo cierto, hay una relación estrecha entre el tipo de grafo, su tamaño, orden y los recursos necesarios para el cálculo de su conectividad.

6 Conclusión

Después de este análisis que contempló diversos enfoques, una implementación directa y tras haber completado las simulaciones experimentales en diversos grafos y, habiendo puesto a prueba las hipótesis propuestas, podemos recopilar las conclusiones obtenidas en el proceso anterior:

La conectividad de un grafo no necesariamente va de la mano con el tamaño de un grafo, esto es debido a que, si el grafo posee un vértice de corte, pese a que tenga muchas aristas, su conectividad será 1. Cuando no hay vértice de corte, este resultado cambia, dependiendo ya de los caminos que posea el grafo, si vértices y aristas aumentan a la par, entonces la conectividad será directamente proporcional. Si no se cumple esto último, dependerá meramente de los caminos y la proporción de vértices y aristas.

Luego, hemos podido apreciar que mientras la cantidad de vértices aumenta, el tiempo de ejecución y los recursos referentes a la memoria lo hacen también, debido a que los algoritmos implementados para buscar la conectividad son algoritmos de búsqueda visitando vértice a vértice, invirtiendo mucho tiempo y utilizando memoria al momento de crear las combinaciones, por lo que tiene sentido que el tiempo y uso de memoria aumenten si el orden del grafo también lo hace. Además, si el grafo tiene muchos vértices y muchas aristas, el tiempo de ejecución y el uso de memoria del algoritmo de conectividad aumenta aún más por la dificultad de desconectar el grafo ante las grandes cantidades de aristas.

La etapa de experimentación se presenta como un argumento sólido de la relevancia que tiene el desarrollar una base para la simulación práctica de escenarios de vulnerabilidad en infraestructuras eléctricas, el cómo, gracias a los resultados obtenidos, logramos recopilar datos interesantes y relevantes a la hora de la planificación y protección de redes, ya que estos y el concepto de la conectividad permiten identificar puntos críticos y ayudan a plantear soluciones efectivas para minimizar el impacto de posibles fallos y disminuir la probabilidad de estos mismos.

Es importante considerar que el estudio hecho fue realizado con la implementación de uno de los enfoques presentados en un comienzo, el algoritmo mediante eliminación de vértices. Con el Teorema de Menger y el Algoritmo de Dinic se abren puertas al descubrimiento y generación de nuevas hipótesis centradas en corolarios que pueden nacer del Teorema, problemas de flujo mayor con Dinic, la capacidad de mantener la conectividad bajo diferentes escenarios, etc. Las posibilidades de análisis aumentan, demostrando aún más la importancia del estudio de la conectividad cuando se refiere a redes y, así mismo, aumentan las potencias diversas aplicaciones que se le puede dar a la conectividad en redes de diferente naturaleza.

Finalmente, a lo largo del desarrollo de este análisis hemos podido apreciar y percatarnos de que el estudio de la conectividad juega un papel crucial a la hora de explorar aplicaciones en sistemas de redes distribuidas de forma inteligente en la vida real. El análisis de datos obtenidos gracias a la conectividad pueden ser una gran ayuda a la hora de mejorar sistemas de interconexión, como la problemática que se presentó al comienzo de todo, maximizando la estabilidad de una red ante eventos imprevistos. En conclusión, la conectividad de grafos no es únicamente una propiedad teórica que se queda encerrada en el ámbito de los grafos, al contrario, es una herramienta esencial y útil para el diseño de redes con mayor robustez, más resilientes y más seguras.

7 Anexos

7.1 Grafos de orden 5

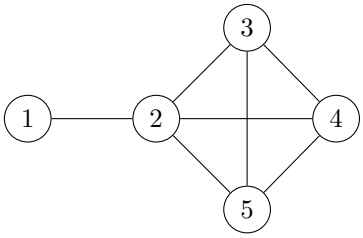

Figure 1: Bridge graph

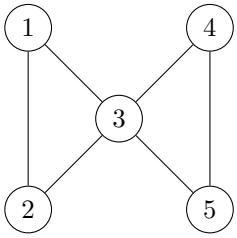

Figure 2: Cut vertex graph

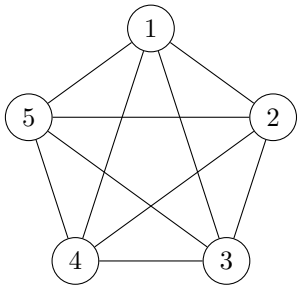

Figure 3: K_5 graph

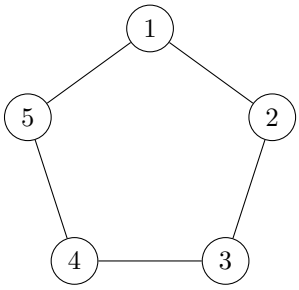

Figure 4: Cycle graph

7.2 Grafos de orden 10

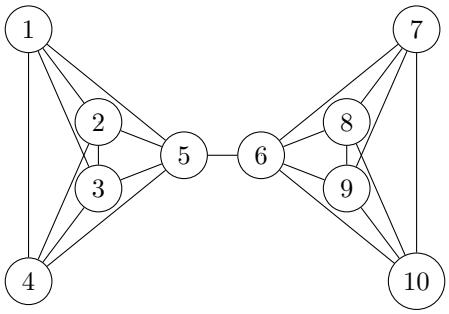

Figure 5: Bridge graph

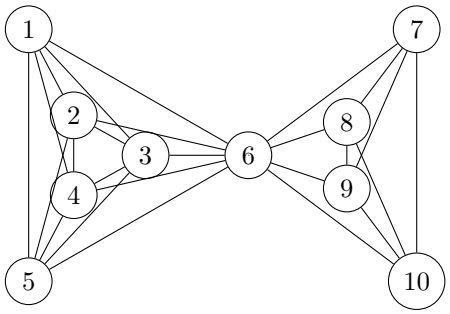

Figure 6: Cut vertex graph

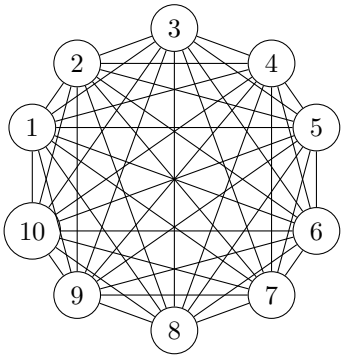

Figure 7: K_{10} graph

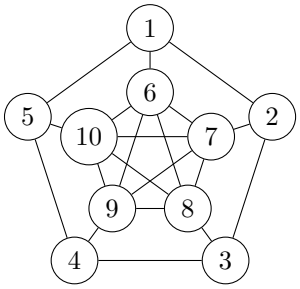

Figure 8: Petersen's graph

7.3 Grafos de orden 15

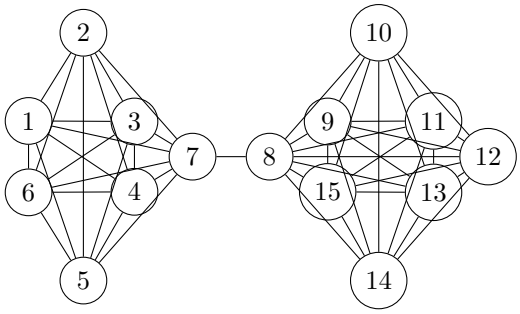

Figure 9: Bridge graph

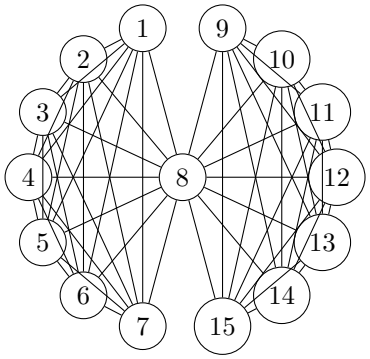

Figure 10: Cut vertex graph

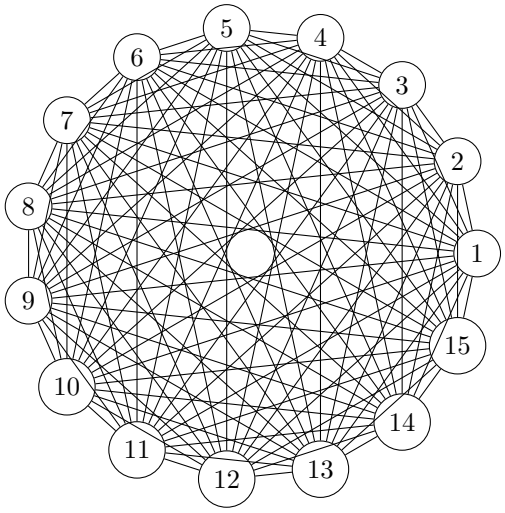

Figure 11: K_{15} graph

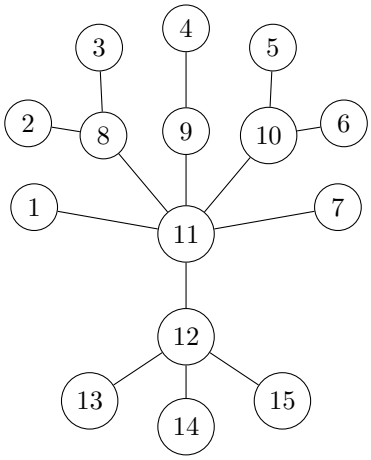

Figure 12: Tree graph

7.4 Grafos de orden 20

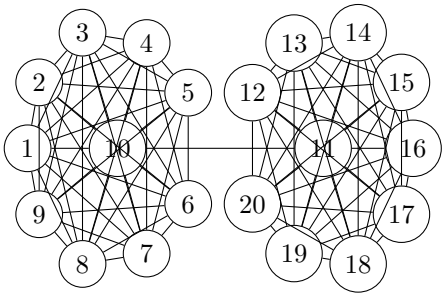

Figure 13: Bridge graph

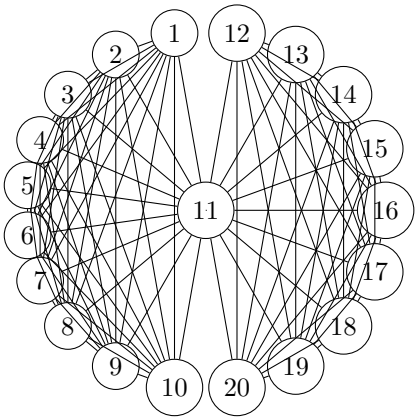

Figure 14: Cut vertex graph

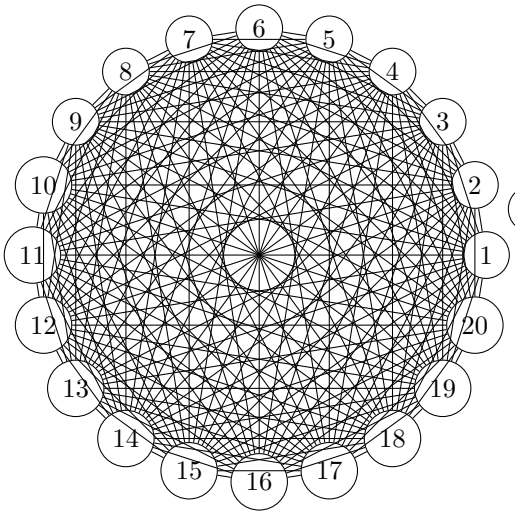

Figure 15: K_{20} graph

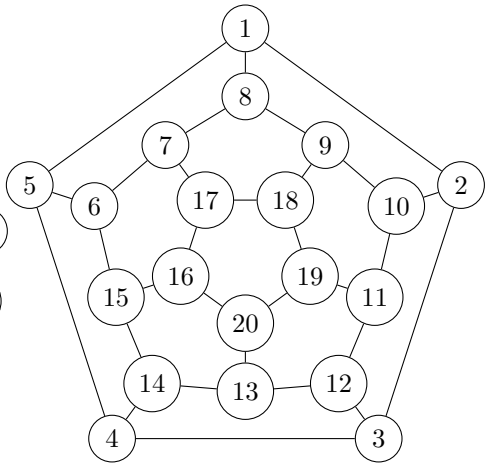

Figure 16: Dodecahedron graph