# Análisis de problemas en grafos.

Rie Kaneko Bojacá
Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia

Felipe Jiménez Borrero
Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia

Oscar David Virguez Castro
Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia

***Resumen. En el siguiente documento se analizarán tres diferentes problemas encontrados al manejar grafos en programación, para los cuales se identificarán sus componentes y definiciones, y se expondrán algunas posibles soluciones. Se tratarán problemas sobre: distancia y caminos más cortos entre nodos de un grafo, cliques en un grafo de Hamming, y árboles de expansión mínima.***

***Palabras clave— grafo; clique; distancia de hamming; árbol de expansión mínima; algoritmo.***

## I. INTRODUCCIÓN

Al trabajar con múltiples datos al momento de programar, se debe entender que en bastantes casos dichos datos van a estar relacionados de cierta manera. Las relaciones entre datos pueden ser bastante sencillas, de modo que los datos en conjunto se comporten como una lista; o bastante complejas, de forma que las relaciones entre los datos se expresen teóricamente como un grafo.

Al trabajar con un grafo de cualquier tipo, surgen problemas cuyas soluciones muchas veces pueden llegar a ser heurísticas, en vez de algorítmicas, o soluciones bastante complejas de por sí. Para dichos casos se pueden llegar a aproximaciones por medio de algoritmos, sin embargo, suelen ser problemas NP-completo, lo cual implica una muy alta complejidad.

A continuación, se presentarán tres problemas distintos de manejo de grafos, los cuales se analizarán para llegar a propuestas de soluciones a los mismos.

## II. CLIQUES EN GRAFOS DE HAMMING

La distancia de Hamming *dist(u,v)* entre dos vectores binarios
*v=(v1, v2, v3, ..., vn)* y w=*(w1, w2, w3, ..., wn)* es el número de índices k tal *vk sea diferente de wk*. Una pregunta fundamental en la teoría de la codificación es determinar el número:

$$A(n,d) = \max|\{S \subset \{0,1\}n \mid dist(u,v) \geq d \text{ for all distinct } u,v \in S\}|$$

El máximo número de vectores binarios de longitud *n* que uno puede encontrar tal que dos vectores diferentes tienen una distancia de Hamming mayor o igual a *d*.

Encuentre un algoritmo "eficiente" para calcular el clique máximo en el grafo de Hamming (calcular el clique máximo es un problema NP-difícil).

En esta problemática se busca encontrar el clique más grande dentro del grafo, dicho subgrafo se construye dependiendo de la distancia de Hamming mínima que se desea tener entre los dos vértices. Para construir este subgrafo, los vértices son conectados cuando la distancia de Hamming es mayor o igual al parámetro ingresado ($d$).

Un grafo es un conjunto de vértices (nodos) unidos por aristas (arcos), que relacionan los nodos, donde permiten representar relaciones binarias entre los elementos de un conjunto. El grafo se puede describir de la siguiente forma:

$$G = (V, E)$$

Donde $G$ es un grafo, $V$ son los conjuntos de vértices y $E$ es un conjunto de aristas.

Existen dos tipos de grafos: el grafo dirigido y el no dirigido. El grafo dirigido tiene un sentido definido en las aristas. Esto hace que importe la dirección entre un nodo y otro de manera que $\{a,b\} \neq \{b,a\}$. En el grafo no dirigido no importa las direcciones de las aristas entre los nodos de manera que $\{a,b\} = \{b,a\}$. Para esta problemática se usa el grafo no dirigido ya que no nos interesa la dirección de cada nodo sino la cantidad de nodos que están conectados.

La distancia de Hamming es la cantidad de pasos necesarios para encontrar la diferencia entre los dos datos. Por ejemplo, si se tiene una palabra “Casa” y otra palabra “Mapa” entonces la distancia de Hamming sería de 2 ya que su diferencia es de dos letras. En este problema se maneja la misma lógica, pero con números binarios, un ejemplo para este caso sería “100” y “000” en el cual su distancia de Hamming es de 1.

El código de Hamming es un código para la corrección de errores que se puede representar como grafos no dirigidos. En redes de comunicación se usa este código para verificar si los datos que se transmiten de un computador a otro llegan correctamente, y el caso de no lo se corrigen. Para ello se usa la siguiente fórmula:

$$2^n \geq n + m + 1$$

Donde $n$ es el número de bits de Hamming y $m$ es el número de bits del dato. Este es uno de los usos que se le da a la distancia de Hamming en la vida real.

La construcción del grafo de Hamming se hace de la siguiente manera:

1. Elegir la dimensión: Esto significa que se escoge el $n$ donde $n$ es la cantidad de dígitos de los números binarios.
2. Sacar todas las posibles combinaciones: Se debe sacar todas las posibles combinaciones de 1's y 0's dentro de las dimensiones elegidas, es decir el total de combinaciones es ($2^n$).
3. Elegir la distancia mínima de Hamming: Se decide que distancia de Hamming mínima para hacer las conexiones se desea tener.
4. Conectar los nodos: Conectar los números binarios que tengan dicha distancia de Hamming. En la Figura1 se puede ver un ejemplo de 3 dimensiones con distancia de Hamming igual a 1.

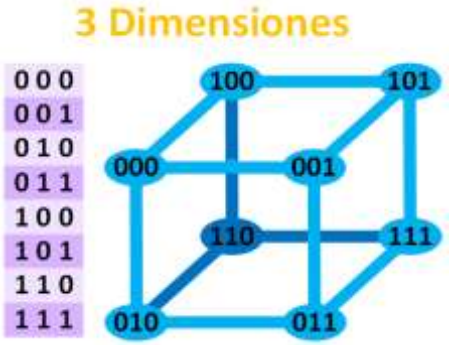


**Figura 1, tomada de [4]**

En la teoría de grafos, el clique es un subconjunto de vértices donde están conectados todos con todos creando un grafo completo.
Cuando se habla del clique máximo, hace referencia al clique que tiene mayor número de vértices, por lo tanto, en el problema que se trabaja, se busca encontrar el clique que tenga más nodos dependiendo de la distancia de Hamming ya que es de ella que se crean las aristas.

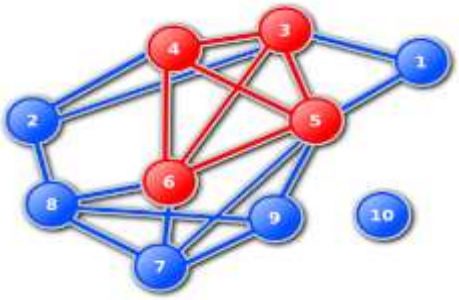


**Figura 2, tomada de [5]**

El cálculo del clique máximo es un problema NP-difícil. Para esto es posible construir un algoritmo que trabaje en tiempo polinómico para cualquier problema de NP, este es el objetivo del proyecto, encontrar un algoritmo que solucione el problema planteado.

A continuación, se plantearán algunos ejemplos y posibilidades de solución.

Ejemplo 1. Sea $A(n, d)$ donde $n$ es la cantidad de dígitos de los números binarios y $d$ la distancia de Hamming mínima que se desea, encontrar el número de clique máximo cuando $A(3,2)$.

Diseño del grafo

i. Encontrar las dimensiones

Como $n = 3$, entonces la es de 3 dimensiones.

ii. Sacar todas las posibles combinaciones entre 0 y 1

Las combinaciones posibles son de $2^3 = 8$.

| | |
|---|---|
| 000 | 100 |
| 001 | 101 |
| 010 | 110 |
| 011 | 111 |

iii. Encontrar la distancia de Hamming mínima

Como $d = 2$, entonces su distancia mínima es de 2.

iv. Conectar nodos

Conectar los datos (punto ii), entre los que tengan distancia de haming mayor o igual a 2.

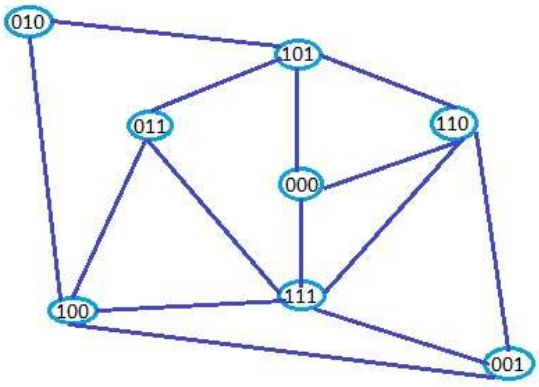


**Figura 3**

Ejemplo 2. Sea $A(n, d)$ donde $n$ es la cantidad de dígitos de los números binarios y $d$ la distancia de Hamming mínima que se desea, encontrar el número de clique máximo cuando $A(4,4)$.

Diseño del grafo

i. Encontrar las dimensiones

Como $n = 4$, entonces la es de 4 dimensiones.

ii. Sacar todas las posibles combinaciones entre 0 y 1

Las combinaciones posibles son de $2^4 = 16$.

| | |
|---|---|
| 0000 | 1000 |
| 0001 | 1001 |
| 0010 | 1010 |
| 0011 | 1011 |
| 0100 | 1100 |
| 0101 | 1101 |
| 0110 | 1110 |
| 0111 | 1111 |

iii. Encontrar la distancia de Hamming mínima

Como $d = 4$, entonces su distancia mínima es de 4.

iv. Conectar nodos

Conectar los datos (punto ii), entre los que tengan distancia de haming mayor o igual a 4.

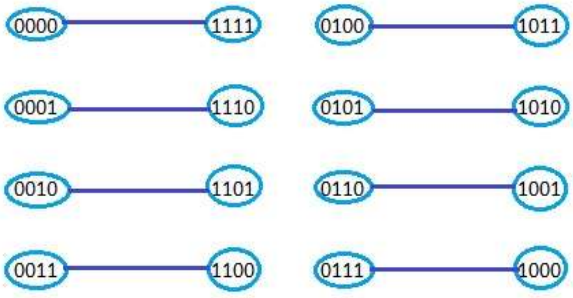


**Figura 4**

Solución. Encontrar el clique máximo

Para encontrar los cliques máximos se puede usar el Algoritmo de Bron-Kerbosch que consiste en pararse inicialmente en cualquier nodo y buscar un nodo adyacente, este proceso se realiza iterativamente siempre buscando nodos que tengan conexión con el clique formado he intentado mezclar cliques para encontrar otros más grandes.

## III. PROBLEMA DE DISTRIBUCIÓN DE TIENDAS

El problema consiste en que se planea construir en una ciudad una cadena de tiendas, a partir de unas ubicaciones estratégicas previamente identificadas en diferentes barrios. Se conoce también con las ubicaciones, la demanda de productos en cada uno de los barrios. Se quieren construir *k* tiendas, con lo cual se formulan las siguientes preguntas:

- ¿Dónde se deben localizar las tiendas para lograr minimizar la distancia promedio entre los clientes?
- Asumiendo que se conoce el costo de construcción de cada una de las tiendas, y se cuenta con la opción de variar el número de tiendas a construir, ¿En qué ubicaciones se deben construir las tiendas para lograr minimizar la distancia promedio entre los clientes y el costo total de producción?

Para dar solución a esta problemática desde una perspectiva algorítmica, debemos adaptarla a una estructura de datos. Para esto definiremos los conceptos necesarios.

Árbol de expansión: Es un árbol que recorre y conecta todos los nodos del grafo desde un nodo inicial teniendo en cuenta el costo o distancia de las aristas de tal manera que no haya ciclos dentro del mismo, este tipo de árboles generalmente son de tipo mínimos o máximos (suma de los costos o distancias de las aristas).

En este problema se hace indispensable el uso de grafos, para su representación más específicamente un grafo no dirigido, en donde los nodos o vértices son los puntos donde se encuentra la demanda y las aristas son los costos de desplazamiento. Es un problema muy relacionado con la vida cotidiana, un ejemplo de este es cuando una empresa quiere construir una o más plantas de producción en una determinada ubicación, puede ser ciudad o barrio, dicha localización no puede ser escogida de manera aleatoria, ya que dependiendo de las necesidades de la empresa, tiene que tener en cuenta factores como la cercanía de la o las plantas con los clientes o en dado caso, los costos de su construcción haciendo indispensable reducir gastos sin afectar la calidad de las plantas, es aquí donde surge la necesidad de encontrar los puntos más eficientes dentro del grafo.

A continuación, se plantearán algunas posibles soluciones a este problema.

a. Árbol de expansión:

Una de las posibles soluciones para este problema es haciendo el uso de un árbol de expansión, en donde e analiza todo el grafo cada vez que se desee insertar una nueva planta, dicho árbol se crearía mediante el algoritmo de Prim, al cual deberá hacérsele unas determinadas adaptaciones para el caso de que se desee construir una cantidad variable de plantas minimizando su

costo de construcción. Debido a que el algoritmo trabaja sobre las aristas, al haber costos de construcción debe también analizarse los valores de cada nodo. [1]

b. Algoritmo ADD

Es el algoritmo que más se ajusta a las necesidades del problema, ya que este si analiza el costo fijo (Costo de construcción), la demanda asociada a cada vértice y el costo variable (distancia entre vértices), dicho algoritmo fue propuesto por John Wiley en el año 1945.

En el caso de que se desea abrir una cantidad variable de tiendas su proceso es:

1. Se busca el vértice que minimiza los costos fijos y variables de todo el grafo (Árbol de expansión mínima) y se realiza la primera construcción en dicho punto.
2. Se realiza el proceso iterativo buscando más puntos en donde se pueda atender la demanda teniendo en cuenta los costos de las aristas.
3. Se continua el ciclo hasta que deje de existir un costo mínimo.

ADD trabaja creando una matriz de costos fijos y variables:

- Los costos fijos en este caso son dados con anterioridad para cada nodo
- Los costos variables están dados por la siguiente expresión:

$$distancia\ entre\ nodos \times demanda = Costo\ variable$$

Una vez creada la matriz con los costos variables para cada par de nodos, se realiza la sumatoria de costes totales para cada nodo y se escoge el de menor valor, este sería el punto inicial de la construcción.

Para cada nueva construcción que se desea realizar es necesario recrear la matriz de costos debido a que, al existir plantas anteriores los costos variables van disminuyendo.

En el caso de que se desea abrir una cantidad K de tiendas su proceso es muy parecido al anterior, con la diferencia de que el proceso iterativo se realiza k veces. [1] [3]

## IV. ÁRBOL DE EXPANSIÓN MÍNIMA

Según este problema, se cuenta con un grafo, el cual podríamos representar teóricamente en una red telefónica, con *m* aristas y *n* vértices, en donde cada arista puede ser azul o roja. En los parámetros del problema hay también una variable numérica *k*, se debe encontrar a partir del grafo un árbol de expansión mínima que cuente con exactamente *k* aristas azules y *n-k-1* aristas rojas. De el algoritmo que solucione este problema se debe evaluar su tiempo de ejecución y demostrar su correctitud.

Para entender bien los conceptos de este problema, se usarán las definiciones de grafo y de árbol de expansión mínima previamente definidas en este texto.

El algoritmo que dé solución a este problema debe contar con las siguientes entradas:

- Datos equivalentes a los vértices, cuyo contenido no es relevante dentro del problema.
- Matriz de relación, de tamaño *n x n* (considerando que el grafo tiene *n* vértices) en donde se indique qué vértices se relacionan entre si y de qué color es su relación (azul o roja).
- Variable k que indique al algoritmo, después del respectivo análisis, cuántas aristas azules y cuántas rojas tendrá que tener el árbol resultado.

El grafo de entrada, representado teóricamente tendrá que lucir como el siguiente ejemplo:

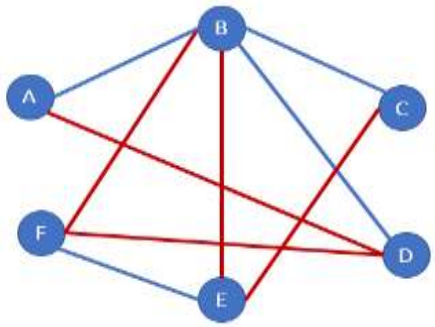


**Figura 5**

El algoritmo debe tener como única salida un conjunto de datos equivalentes a los datos de los vértices, junto con una matriz de relación en la cual se especifica qué vértices se relacionan entre sí y con qué color, de manera que formen un árbol de expansión mínima con las características requeridas por quién define la variable *k*.

Para el ejemplo de grafo de entrada mostrado anteriormente, una posible salida podría ser la siguiente:

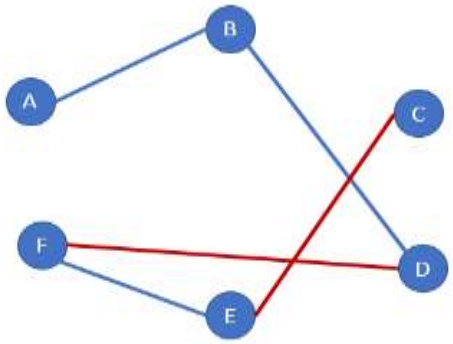


**Figura 6**

Para dar solución a la creación de árboles de expansión mínima a partir de un grafo existen diferentes algoritmos actualmente. Para este problema en particular, se debe tener en cuenta que las soluciones que se implementen, aunque puedan basarse en algún algoritmo ya existente, deben evaluar como condiciones adicionales para la construcción del árbol, los límites de aristas de cada tipo dependiendo del valor de la variable *k*.

Un algoritmo sobre el cual se puede establecer la base de la solución a este problema es el algoritmo de Kruskal, el cual es un algoritmo voraz. Este algoritmo se basa bajo el principio de agregar aristas al árbol siempre que ellas no formen un ciclo. Además, el algoritmo ordena las aristas de menor a mayor para trabajar también con los costos de las mismas. Un pseudocódigo del algoritmo sería el siguiente [9]:

```
1  método Kruskal(Grafo):
2      inicializamos MST como vacío
3      inicializamos estructura de unión
4      ordenamos las aristas del grafo
5      para cada arista e que une los vértices u y v
6          si u y v no están en la misma componente
7              agregamos la arista e al MST
8              hacemos la unión de las componentes de u y v
```

Para este caso en particular, en el que fuera del simple método del árbol de expansión, requerimos también verificar los colores de las aristas, se haría necesario modificar algunas partes de este proceso.

Primero, se tendrían que descartar los elementos del algoritmo que no son relevantes al problema en particular. Para este caso, el árbol no está trabajando con costos, así que el ordenamiento de las aristas del grafo se puede descartar sin afectar negativamente nuestro resultado.

Adicionalmente tenemos que añadir las condiciones que la inserción de la variable *k* implica. Debemos tener en cuenta el añadir como parámetro a la función la variable k, con la cual, dentro de la función, declaramos una variable *x = k*, la cual determinará el número de aristas azules; y una variable *y = k -n-1*, la cual determinará el número de aristas rojas. Al momento de entrar a la condición interna, se deben añadir otras instrucciones, al agregar una arista al árbol, y antes de hacer la unión de los componentes de u, se debe restar en 1 el valor de *x* si la arista agregada es azul, o restar en 1 el valor de *y* si la arista agregada es roja. Por último, para que entre e la condición interna se debe evaluar también, que, si la arista a agregar es roja, el valor de *y* sea mayor a 0, y que, si la arista a agregar es azul, el valor de *x* sea mayor a 0.

De esta manera se podrá garantizar que el árbol de expansión mínima que se obtiene, cumpla con las características dadas por la variable k en los parámetros de la función.

## V. IMPLEMENTACIÓN

- **Problema de Cliques en grafos de Hamming:**

Como se dijo anteriormente, el problema se enfoca en encontrar el clique máximo a través de un grafo construido por la distancia de Hamming, el cual depende de los números binarios (n) y la distancia mínima desea (d). El "n" y el "d" son las entradas que se reciben en el archivo de texto.

Para la implementación de este algoritmo, en la construcción del grafo se utilizó la matriz de adyacencia, para lo cual se usó la librería *numpy* con el fin de poder reservar el tamaño que es dependiente de los números binarios y de sus relaciones con aristas, el grafo se construye dependiendo de la distancia mínima de Hamming que se obtiene por el archivo de texto. Para la implementación de esto, se tiene las siguientes funciones:

**construct_graph:** Esta función crea el grafo, recibe el parámetro "n" como números binarios y "d" como la distancia mínima deseada. Para crear el grafo primero se mandan las dos variables a la función *graph_relition* para obtener las posiciones de conexiones del grafo. Cuando se obtiene todas las posiciones para conectar las aristas del grafo, los valores de las aristas se ponen en 1. Al final se retorna "graph", el cuál es el grafo resultante, es una matriz de adyacencia y el "count" son los números de aristas que tiene ese grafo.

**graph_reltion:** Primero se manda la variable "n" a la función *true_table* para encontrar todos los vértices.

Después se mira todas las conexiones entre los vértices con dos ciclos y esos vértices se mandan a la función *hamming* para ver las diferencias binarias entre esas dos. Después de obtener la distancia de Hamming, se mira si esa distancia que se obtuvo es mayor o igual a la distancia "d" y en el caso de que coincida, se guarda la posición en la lista "positions" donde al final es retornada.

**hamming:** Es la función que encuentra y retorna la distancia de Hamming de los dos vértices "binary_1" y "binary_2" que se envían por parámetro.

**true_table:** Tiene como parámetro los números de bits (b), en el cual se encuentran todas las posibles combinaciones de 1's y 0's dentro de esa dimensión, es decir todas las combinaciones de $n^2$. Las combinaciones posibles se van a guardando en una lista llamada "vértices" la cual es retornada al final.

Para la implementación del clique máximo, se utilizó el algoritmo Bron-Kerbosch que es un algoritmo que trabaja en tiempo polinómico para cualquier problema de NP, fue implementado con algunas modificaciones para adaptarlo a esta problemática. Este algoritmo consiste en pararse inicialmente en cualquier nodo y buscar un nodo adyacente, este proceso se realiza iterativamente, siempre buscando nodos que tengan conexión con el clique formado he intentado mezclar cliques para encontrar otros más grandes. Las funciones utilizadas son los siguientes:

**maxim_clique:** Se manda como parámetro la matriz de adyacencia o más bien el grafo que se construyó con la función *contrusc_graph* y el "n" que es el número de las dimensiones. El grafo se manda a la función *bron_kerbosch* para encontrar los cliques máximos donde se va guardando en la variable "data", esta última es enviada como también parámetro. Ya obtenida la variable "data" que es la información de los cliques máximos, se crea "clique_list" para retornar finalmente esta variable con la información organizada.

**bron_kerbosch:** Es un algoritmo recursivo backtracking, qe recibe como parámetros, "p" vértices, "r" algunos de los vértices de "p", y "x" ninguno de los vértices de "p". Donde P se inicializa con todos los vértices que tiene el grafo, "r" y "x" vacío. En cada llamada recursiva el algoritmo revisa los vértices de "p" en donde "r" se añade a los vértices(v), "p" y "x" deben ser los vecinos de N(v) para hacer el llamado recursivo. Después se elimina el vértice(v) de la lista "p" y se añade a la lista "x" para excluirlo.

**N:** Es una función que hará los bucles al vértice del grafo y los retorna. Para esto se necesita de los parámetros grafo y los respectivos vértices.

Al final del código se tienen funciones adicionales que se utilizan para dibujar el grafo que se construyó, la lectura y escritura del archivo del texto.

**draw_graph:** Recibe la matriz como parámetro y dentro de la función se utiliza la librería *network* para cambiar a tipo graph, después se dibuja el grafo con la librería *pyplot*.

**open_text:** Como el nombre lo dice, es una función reada para obtener la información de un archivo del texto. Para que pueda obtener la información correctamente, es necesario que en la primera fila del archivo sea la cantidad de nodos "n" y en la segunda fila la distancia mínima de Hamming "d". En el parámetro se recibe el nombre del archivo que se desea leer.

**save_text:** En el parámetro se envía el nombre del archivo, los vértices, el grafo como una matriz de adyacencia y la lista de cliques máximos para guardar toda la información en un archivo de texto. La información se guarda siendo primero todos los vértices, después el grafo en forma de matriz, y por último todos los cliques máximos que se tienen en el grafo.

- **Problema de distribución de tiendas:**

La primera parte del enunciado solicita crear *n* cantidad de tiendas. Este algoritmo se basa en dos partes importantes, la primera es del uso de un algoritmo de "Distancia más corta" puede ser Dijkstra o FloydWarshall, en este caso, se vio mucho más cómodo hacerlo por Warshall, la segunda es el cálculo de los nodos más factibles a crear la nueva tienda, se usa la librería numpy para el manejo de matrices, en resumen el código primero calcula las menores distancias entre cada par de nodos, una vez sean obtenidas realiza la sumatoria horizontal por cada nodo y en el menor costo, será la primera localización de la tienda, el proceso es iterativo y se repite la cantidad de veces que se desee construir una nueva tienda, una vez se asigna un nodo, este no se tiene en cuenta para una nueva asignación, con la diferencia que los valores de la matriz varían por las nuevas tiendas construidas para atender la demanda.

El proceso para realizar la ubicación de la mayor cantidad de tiendas posibles, con el mínimo costo se hace de la misma forma, sólo que en este caso, el proceso iterativo se detiene cuando la última inserción de una tienda, tiene un costo mayor a su antecesor.

Si el problema solicita calcular la mínima distancia promedio, se realiza el proceso descrito anteriormente, pero una vez se acaban las asignaciones, se busca la distancia mínima en cada nodo no asignado y se promedia con los demás nodos que no fueron asignados.

Descripción de funciones:

**FloydWarshall(W):** Algoritmo para hallar la distancia más corta entre cada par de nodos, este algoritmo se implementó exactamente de la misma manera que la suministrada por el profesor en los notebooks y usada por nosotros en los talleres.

**minVal(mat, add, col):** Función que recibe como parámetros la matriz de costos (mat), una lista de nodos con tiendas (add), numero de la columna actual evaluada (col). La función busca y retorna el costo mínimo en la matriz de costos entre un nodo con los demás.
**names(add, names):** Función que recibe como parámetros un vector de nodos con tiendas (add) y un vector con los nombres de los nodos (names). La función retorna una lista con los nombres de los nodos a crear nuevas tiendas.
**minVal(mat, add, col):** Función que recibe como parámetros, la matriz final de costos (mat) y un arreglo con las posiciones de los nodos asignados (add). La función retorna la distancia promedio de los nodos no asignados.
**names(add, names):** Función que recibe como parámetros un vector de nodos con tiendas (add) y un vector con los nombres de los nodos (names). La función retorna una lista con los nombres de los nodos a crear nuevas tiendas.
**averageDistance(mat, add):** Función que recibe como parámetros, la matriz final de costos (mat) y un arreglo con las posiciones de los nodos asignados (add). La función retorna la distancia promedio de los nodos no asignados.
**addNPlants(mat, dem, nPlants):** Función que recibe como parámetros la matriz de costos (mat), una lista con la demanda de cada nodo (dem) y un número de tiendas a construir (nPlants). La función una vez obtiene los costos de distancia y demanda en general busca los valores mínimos y asigna las tiendas a los nodos.
**accumCost(add, cc):** Función que recibe como parámetros una lista de tiendas asignadas (add) y una lista de costos de construcción (cc). La función retorna el acumulado de costos de construcción para cada nodo producto de la creación de tiendas anteriores.
**addAllPlants(mat, dem, cc, nPlants):** Función que recibe como parámetros la matriz de costos (mat), una lista con la demanda de cada nodo (dem) y una lista con los costos de construcción de cada tienda (cc). La función una vez obtiene los costos de distancia y demanda en general busca los valores mínimos teniendo en cuenta los costos de construcción y asigna las tiendas a los nodos.
**fillData(lines, dem, option):** Función que recibe como parámetros una lista de líneas escaneadas en el archivo de texto (lines), una lista con los valores de la demanda (dem) y una opción para el caso que se incluyan o no costos de construcción (option), esta función se encarga de tokenizar cada línea y organizar la información de la manera adecuada para posteriormente llamar a la función de ADD. Retorna la lista de nodos a construir tiendas y los nombres de todos los nodos.
**readFile(textFile):** Lectura del archivo, función que recibe el nombre del archivo como parámetro y retorna una lista con las líneas del archivo.

- **Problema de árbol de expansion minima con colores:**

Para la implementación de este algoritmo se toma como base algún algoritmo ya existente con el que se puedan detectar árboles de expansión dentro de un grafo. Se decide usar específicamente el algoritmo de Kruskal, el cual encuentra dentro de un grafo ponderado, un árbol de expansión mínima.

El algoritmo requiere ciertas modificaciones, por ejemplo, con respecto a los pesos, cuyo espacio en la etiqueta de la arista se duplica para incluir una etiqueta adicional que especifique si la arista es azul o roja. Además, se debe modificar la condición que permita establecer si una arista entra o no dentro del árbol de expansión para posteriormente unir sus vértices y agregarla; considerando también que: el color sea azul y queden aún aristas azules por incluir (verificado con un contador), ó el color de la arista sea rojo. Luego de agregar la arista se disminuye el número de aristas azules por ubicar en el caso en que esta sea azul.

Para asegurar la correctitud de este algoritmo se tienen 3 invariantes:

*A. En cada iteración del algoritmo, en la cual se agregue una arista al árbol de expansión mínima, el árbol resultante no debe contener ciclos. Esta invariante es heredada del algoritmo de Kruskal.*

*B. Se debe tener antes de retornar el árbol de expansión completo, igual cantidad de aristas azules a las solicitadas, lo cual verificará que también existan tantas aristas rojas como se requiera, en caso de que también se cumpla la tercera invariante.*

*C. Se debe verificar antes de retornar el árbol de expansión, que la cantidad de aristas existentes sea igual a la cantidad de vértices del grafo inicial -1.*

Debido a que este algoritmo parte del algoritmo de Kruskal, su complejidad cambia solo en los aspectos en los cuales el algoritmo presente diferencias. Las diferencias se darán en la comparación adicional del color de la arista al momento de decidir si esta se poda o no, lo cual no afectaría considerablemente su complejidad debido a que igualmente se podarán cuantas aristas sean necesarias para que el grafo resultante sea un árbol de expansión mínima. Por estas razones, para un grafo de $n$ vértices y $m$ aristas, la complejidad será de $O(m*log(n))$.

Se utilizaron para la implementación de este algoritmo dos algoritmos en particular, uno basado en Kruskal, el cual evalúa el color de la rama antes podarla para garantizar la correctitud, y otro que lee el grafo y las especificaciones a partir de un archivo y las adapta al formato de entrada del algoritmo.

Para una segunda implementación de este algoritmo se decide también tomar como base un algoritmo ya existente para hallar árboles de expansión mínima, en este caso en particular, el algoritmo de Prim.
El algoritmo requiere ciertas modificaciones, por ejemplo, con respecto a los pesos, cuyo espacio en la etiqueta de la arista se duplica para incluir una etiqueta adicional que especifique si la arista es azul o roja. Además, se debe modificar la condición que permita establecer si una arista que cuente con un nodo no visitado, entrará o no dentro del árbol de expansión mínima; considerando también que: el color sea azul y queden aún aristas azules por incluir (verificado con un contador), ó el color de la arista sea rojo. Luego de agregar la arista se disminuye el número de aristas azules por ubicar en el caso en que esta sea azul.

Para asegurar la correctitud de este algoritmo se tienen 3 invariantes similares a las requeridas por el algoritmo de Kruskal:

*D. En cada iteración del algoritmo, en la cual se agregue una arista al árbol de expansión mínima, el árbol resultante no debe contener ciclos. Esta invariante es heredada del algoritmo de Prim.*

*E. Se debe tener antes de retornar el árbol de expansión completo, igual cantidad de aristas azules a las solicitadas, lo cual verificará que también existan tantas aristas rojas como se requiera, en caso de que también se cumpla la tercera invariante.*

*F. Se debe verificar antes de retornar el árbol de expansión, que la cantidad de aristas existentes sea igual a la cantidad de vértices del grafo inicial -1.*

Debido a que este algoritmo parte del algoritmo de Prim, su complejidad cambia solo en los aspectos en los cuales el algoritmo presente diferencias. Las diferencias se darán en la comparación adicional del color de la arista al momento de decidir si esta se incluye o no dentro del árbol, lo cual no afectaría considerablemente su complejidad, debido a que igualmente se incluirán dentro del árbol la misma cantidad de aristas. Por estas razones, para un grafo de n vértices, la complejidad será de O(n2).

## VI. RESULTADOS Y ANÁLISIS

- **Problema de Cliques en grafos de Hamming:**

La complejidad de la función *bron_kerbosch* es de $2^v$ donde v son los números de vértices. La complejidad de la distancia de Hamming es de $n^2$. Por lo tanto, la complejidad en el peor de los casos es de $2^v$. En la siguiente grafica se puede ver el resultado del tiempo y la cantidad de procesos realizados en la ejecución del algoritmo.

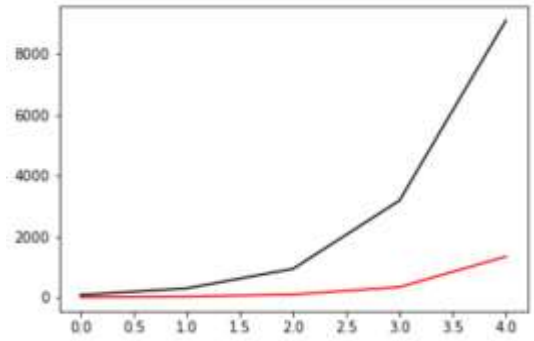


**Figura 7. Comparación del tiempo y procesos realizados en la ejecución de algoritmo del cliques máximos y distancia de hamming.**

En la figura 7, la línea del color negro es la cantidad de procesos y la línea rojo es el resultado del tiempo. Para el análisis del tiempo se realizaron 5 iteraciones del algoritmo debido a que cuando se aproxima al peor de los casos es muy demorado en la ejecución.

- **Problema de distribución de tiendas:**

El algoritmo utilizado para solucionar los problemas de las tiendas fue el algoritmo ADD, el cual trabaja de una forma parecida a un algoritmo Greedy tomando la mejor decisión según las condiciones en la ejecución, tiene una complejidad de $O(n^3)$, ya que se basa en una matriz de distancias entre cada par de nodos en el grafo del cual también depende su costo en memoria, más las listas y variables auxiliares.

Los tiempos de ejecución de cada entrada pueden encontrarse en el archivo 'output_LP.txt'. Es un poco complejo graficar el tiempo de ejecución con matrices aleatorias, debido a que se pueden generar valores muy grandes lo que obligue a siempre instalar una sola tienda, por tal motivo no se realizarán pruebas gráficas de la ejecución del algoritmo, sin embargo, como ya se mencionó, al final de cada ejecución en el archivo de texto, se muestra su tiempo de ejecución.

- **Problema de árbol de expansión mínima con colores:**

Cada uno de los algoritmos present resultados favorable para el problema planteado, pues se puede notar en las implementaciones, que, aunque puedan obtener árboles diferentes (en entradas iguales), siempre se tendrá el mismo peso, garantizando la correctitud del árbol de expansión mínima y de la cantidad de aristas azules o rojas requeridas.

Se realiza en cada uno de los algoritmos planteados su tiempo de ejecución junto con los pasos que toma para su solución utilizando 80 grafos aleatorios de 1 a 80 vértices con una cantidad de aristas de aproximadamente el doble de sus respectivos vertices para cada uno de los grafos.

Para la primera implementación, basada en Kruskal, se obtiene un tiempo de ejecución relativamente corto para cada uno de los casos, sin embargo, podemos ver un claro aumento en la cantidad de operaciones a realizar.

La segunda implementación, basada en el algoritmo de Prim, se comporta de manera muy similar a la primera, en la cual los tiempos son relativamente cortos y la cantidad de pasos aumenta considerablemente.

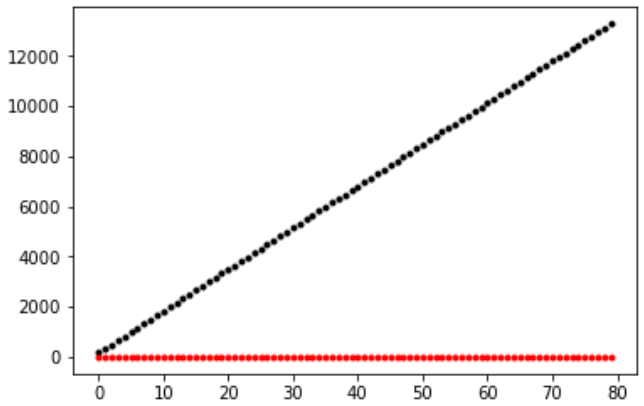


**Figura 8. Implementación basada en algoritmo de Kruskal. En rojo se ve el tiempo y en negro la cantidad de pasos.**

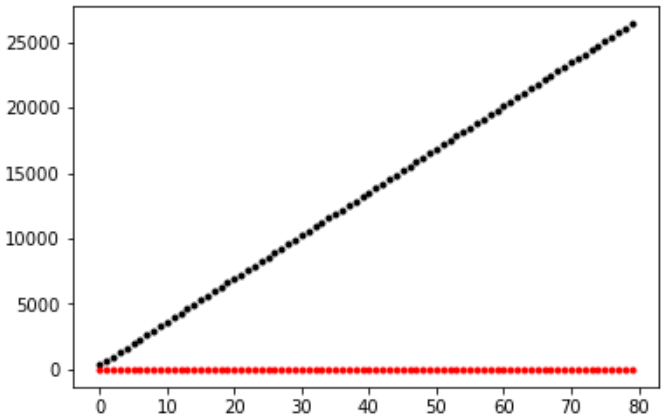


**Figura 9. Implementación basada en algoritmo de Prim. En rojo se ve el tiempo y en negro la cantidad de pasos.**

Para comparar ambas implementaciones , se ubicaron sus resultados en grafos aleatorios de cantidad de vértices de 1 hasta 60; en una misma gráfica. En la gráfica se muestra con color rojo el tiempo y los pasos por la implementación basada en Prim, y en color negro el tiempo y los pasos por la implementación basada en Kruskal.

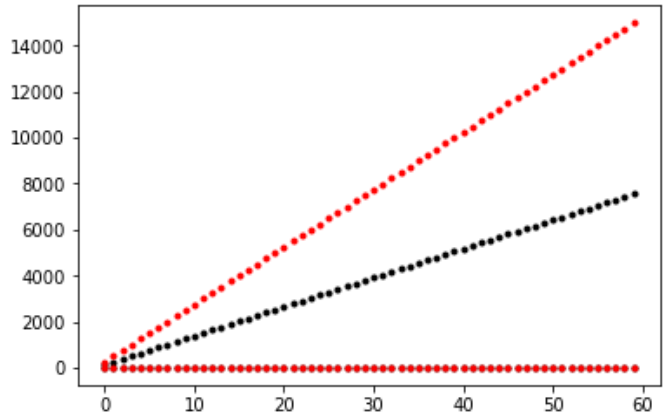


**Figura 10. Comparación de implementaciones. En negro se ve la implementación basada en Kruskal, y en rojo se ve la implementación basada en Prim**

A partir de esta gáfica podemos concluir que aunque los tiempos de ejecución no difieran mucho, por cuestiones de que probablemente muchos de los casos aleatorios no puedan obtener un árbol de expansión mínima con las especificaciones requeridas; siempre veremos una cantidad considerablemente mayor de instrucciones en el algoritmo basado en Prim, por lo cual la primera implementación, basada en el algoritmo de Kruskal, será la más eficiente.

## VII. REFERENCIAS


[1] M. Soto, «YouTube,» 7 Noviembre 2014. [En línea]. Available: https://www.youtube.com/watch?v=IB2rGgMWnM0. [Último acceso: 15 Febrero 2017].

[2] «YouTube,» 28 Diciembre 2008. [En línea]. Available: https://www.youtube.com/watch?v=Kh60Juq-hwM. [Último acceso: 2017 Febrero 15].

[3] «Scielo,» Abril 2004. [En línea]. Available: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382004000100008. [Último acceso: 2017 Febrero 15].

[4] R. Invarato, «Jarroba.com,» 9 Noviembre 2016. [En línea]. Available: https://jarroba.com/hamming/. [Último acceso: 20 Febrero 2017].

[5] R. González, «Blogger,» 5 Julio 2011. [En línea]. Available: http://esteban-gzz.blogspot.com.co/2011/07/problema-de-clique.html. [Último acceso: 20 Febrero 2017].

[6] L. Alcón, «Universidad Nacional de La Plata,» Marzo 20013. [En línea]. Available: http://www.mate.unlp.edu.ar/tesis/tesis_liliana.pdf. [Último acceso: 20 Febrero 2017].

[7] Wikipedia, «Wikipedia,» 11 Febrero 2017. [En línea]. Available: https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch_algorithm. [Último acceso: 21 Febrero 2017].

[8] Wikipedia, «Wikipedia,» 17 Mayo 2016. [En línea]. Available: https://es.wikipedia.org/wiki/Problema_de_la_clique. [Último acceso: 21 Febrero 2017].

[9] J. Arias «Algorithms and more» 19 Abril 2012. [En línea]. Available: https://jariasf.wordpress.com/2012/04/19/arbol-de-expansion-minima-algoritmo-de-kruskal/ [Último acceso: 20 Febrero 2017].