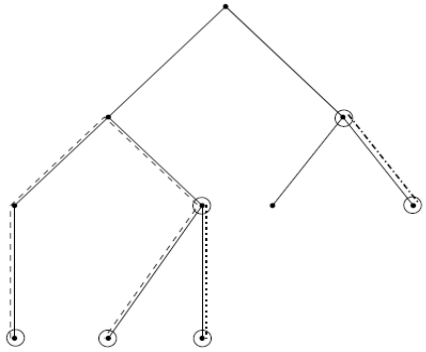
1.- Sea T un árbol con raíz r y S un conjunto con un número par de vértices de T. Diseña un algoritmo de tiempo polinomial que encuentre |S|/2 caminos simples y disjuntos en aristas que emparejen pares de vértices en S. La figura siguiente muestra un ejemplo en el que S consta de los seis vértices encerrados en círculos y tres caminos (denotados con tres tipos distintos de líneas punteadas) que emparejan esos vértices.
Tip: Gana intuición haciendo instancias como la de la figura y observando lo que ocurre cuando recorres toda la frontera del árbol casi tocándola, pero sin hacerlo realmente; luego piensa en como expresar la idea de semejante recorrido con un algoritmo recursivo.

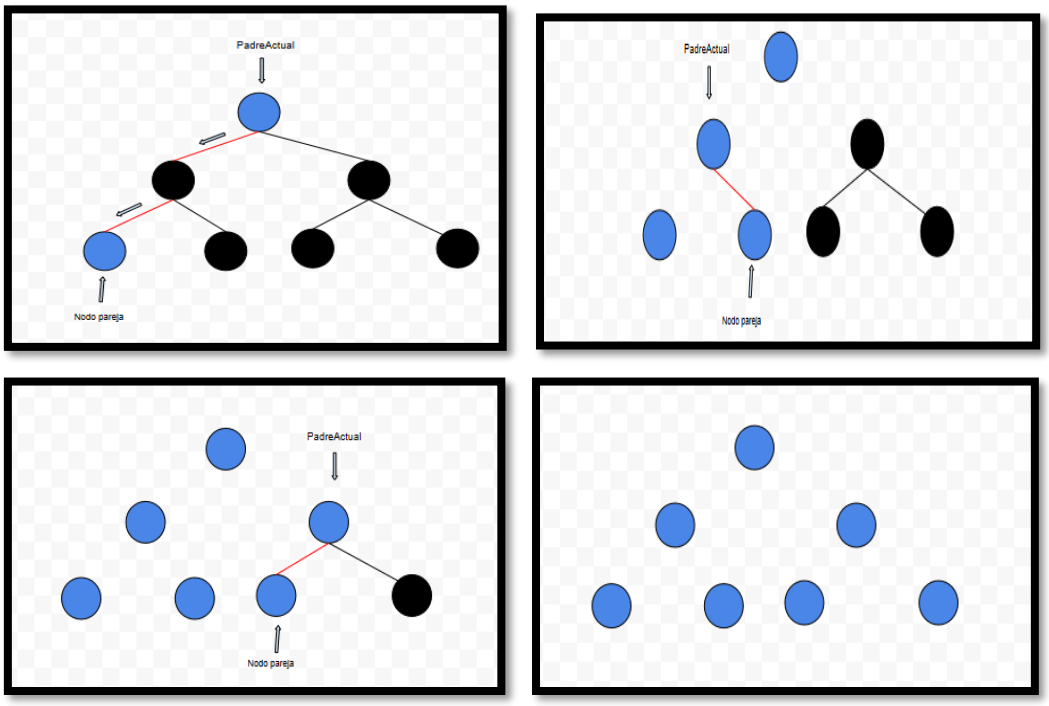


```
Inicializa
    PadreActual = 0 //Variable global que contiene el padre actual
    Bloq[n]         //Arreglo global, se crea un arreglo de tamaño n de
                    //valores boleanos
    Visitados[u]    //Arreglo global, se crea un arreglo que sirve para
                    //indicar los nodos ya visitados
    Caminos[n]      //Arreglo global, se crea un arreglo de tamaño n donde
                    //cada uno de sus elementos es una pila, es decir es un
                    //arreglo de pilas
    for i = 1 to n
        Bloq[i]      = false //inicializa a false cada ubicación
        Visitados[i] = false //inicializa a false cada ubicación
    endfor

Conjunto(Adj)
    Inicializa(Adj.length)
    if Adj.length > 0 then
        for i = 1 to Adj.length
            if Bloq[i] == false && Adj[i].length > 0 //se verifique que el vertice tenga relaciones y este libre
                emparejado  = false //Variable global que indica que no se ha emparejado el vertice i
                Bloq[i]     = false //se bloquea el nodo para que nadie lo considere
                PadreActual = i //se establece el padre actual
                Emparejar_Vertice(i,Adj) //se empareja i con algún nodo
Emparejar_Vertice(u,Adj)
    Visitados[u] = true //se indica que esta visitado el nodo
    for i = 1 to Adj[u].length
        if emparejado == false && Visitados[Adj[u][i]] == false && Indep[Adj[u][i]] == false then
            Emparejar_Vertice(Adj[u][i],Adj) //se llama recursivamente al método
        endif
        if emparejado == true then
            Eliminar_Relacion(u,Adj[u][i],Adj) //como es una relación bidirecional
            Eliminar_Relacion(Adj[u][i],u,Adj) //se eliminan ambas relaciones
            Camino[PadreActual].meter(u) //agregar a la pila el vertice parte del camino
            break                        //rompe el ciclo y ya no sigue iterando en los demas hijos
        endif
    endfor
    if emparejado == false then // la primer llamada recursiva que termine el ciclo for y llegue aquí
                                //entonces quiere decir que es un nodo candidato para ser pareja
        emparejado = true       //se indica que ya se encontro pareja
        Camino[PadreActual].meter(u) //agregar a la pila el vertice parte del camino
    endif
    Visitados[u] = false //se restablece el false

Elimina_Relacion(v,u,Adj)
    for i = 1 to Adj[v].length
        if Adj[v][i] == u then
            Adj[v].remover_elemento(i)
        endif
        if Adj[v].length == 0 then //si ya no tiene relaciones con nadie, entonces se marca como bloqueado
            Bloq[v] = true     //con el fin que no sea tomado en cuenta
        endif
        if Bloq[v] == true then //si el vertices ya esta bloqueado, entonces se eliminan todas las relaciones
            Eliminar_Relacion(Adj[v][i],v,Adj) //con sus demas hijos para que ya no sea considerado
            Adj[v].remover_elemento(i) //Elimina un elemento en la posición i
        endif
    endfor
```

Idea principal:
Para resolver el problema se utilizo una modificación de la búsqueda en profundidad, la idea es elegir un nodo *u* y bajar hasta llegar a un nodo "hoja" *v,* después *u* y *v* son parte de S y se obtiene el camino que une dichos nodos, tales nodos se marcan como "bloqueados" y se eliminan las aristas que son parte del camino que unen a *u* y *v*, el algoritmo trata entonces de ir descomponiendo el grafo, es decir, ir retirando todos los vértices que son emparejados junto con sus aristas que son parte del camino, de esta manera al final de la ejecución solo quedan los vértices sin ninguna arista.



Faces de búsqueda y descomposición del grafo

Corrección del algoritmo:
A continuación, se demostrará que el algoritmo es correcto y devuelve al conjunto S con un número par de vértices y |S|/2 caminos diferentes que emparejan a los pares de vértices.
Primero que nada, las gráficas que recibe el algoritmo son no-dirigidas y están representadas en una lista de adyacencia y cada elemento en la lista, que inicia en 1, representa un vértice, es decir el vértice 1, 2, …, n.
El algoritmo inicia en la línea 15 con la función llamada "Conjunto", en la línea 16 se inicializan las variables que utiliza el algoritmo, todas esas variables con globales, entonces analizando la función "Inicializa" (línea 1) tenemos que:

PadreActual (línea 2): Contiene el "padre actual" que es el nodo raíz actual el cual se intenta emparejar con otro nodo.

Bloq (línea 3): Este arreglo sirve para "bloquear" a los nodos que ya son parte del conjunto o que están solos y no es posible emparejarlos

Vistados (línea 5): Este arreglo ayuda durante la búsqueda de los nodos, indicando que nodos ya fueron visitados.

Caminos (línea 7): Es un arreglo en el que cada posición representa un número de un nodo y el contenido es una pila que contiene el camino que empareja un par de vértices donde el primer y

último elementos en sacar son los vértices a emparejar y el resto conforman el camino, el primer elemento corresponde al nodo con el número de la ubicación del arreglo y el último es el vértice con el que hace pareja.

En las líneas 10-13 se establece a cada elemento de los arreglos “Visitados” y “Bloq” su valor a false. Continuando en la función “Conjunto” en la línea 17 se verifica que la lista de adyacencia tenga más de un vértice, ya que se necesita una gráfica con mínimo dos vértices para que puedan ser emparejados, de otra manera no es posible.
En el ciclo de la línea 18 se itera tantas veces como la cantidad de vértices existentes en la gráfica.
En la línea 19 se verifica que siga disponible y que tenga relaciones con otros vértices.
En la línea 20 se establece la variable de emparejado a false indicando que espera ser emparejado, en la línea 21 se bloquea el vértice, en la línea 22 se establece el padre actual, este sirve para saber a qué ubicación del arreglo “Caminos” se agregará el camino encontrado y en la línea 23 se llama a la función “Emparejar_Vertice” parametrizado con el vértice a emparejar y la lista de adyacencia.
En la línea 30 de la función “Emparejar_Vertice” se indica que el vértice *u* ha sido visitado, después en la línea 31 se recorre su lista de relaciones con sus demás hijos. En la línea 32 se verifica que la variable emparejado aun no sea verdadera y que el i-esimo vértice en la lista de relaciones de *u* no este bloqueado y no este visitado, entonces si la condición if se evalúa como verdadera entonces se exploran los hijos del i-esimo vértice (línea 33).
En la línea 35 se verifica que la variable “emparejado” sea verdadera, si esto es cierto entonces en las líneas 36-37 se eliminan las relaciones con entre vértices (ya que habíamos considerado que la gráfica es no-dirigida), después se agregar a la pila del “padre actual” el vértice *u* como parte del camino y finalmente se obliga a salir del ciclo ya que no tiene caso seguir buscando en los demás hijos de *u* (línea 39) porque el vértice ya ha sido emparejado.

Cuando una rama de la grafica sea explorada en profundidad, entonces las líneas 42-46 serán ejecutadas. Vamos a suponer que existe un vértice *w*, para el cual no se evalúa la línea 32 como verdadera, esto es porque ya no tiene más vértices que visitar (Fig. 1) o forma parte de una relación con su vértice a emparejar (Fig. 2)

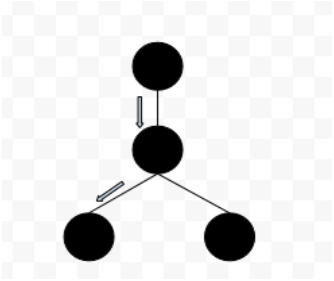


Fig.1 Recorrido en gráfica donde se evidencian las hojas

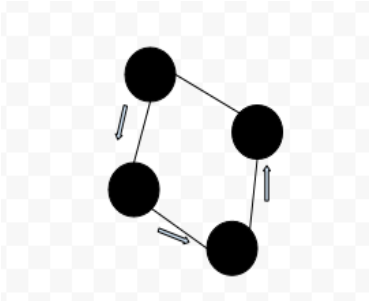


Fig. 2 Recorrido en grafica donde el nodo a emparejar es hijo del nodo que será su pareja.

Entonces si esta condición no se cumple para ningún vértice conectado a *w*, entonces no se volverá a llamar la función recursivamente, por lo tanto, el ciclo finalizará y podrá por primera vez ser establecida la variable "emparejado" a true (línea 44) y posteriormente se agregará *u* como parte del camino en la pila del vértice "PadreActual".
Después de que las llamadas recursivas regresen, en la línea 36-37 se elimina las relaciones para descomponer el grafo y no pueda ser considerada esa arista nuevamente y también es agregar *u* como a la pila del vértice "PadreActual".
La función "Elimina_Relación" es la encargada de eliminar una relación entre vértices, recibe como parámetros dos vértices y la lista de adyacencia, el ciclo for de la línea 51 recorre la lista de vértices incidentes a *v* y en la línea 51 se verifica que el elemento i-esimo de la lista de *v* sea igual a *u* y si lo es entonces en la línea 52 se elimina ese elemento de la lista en la posición i, si la lista de v después de la eliminación queda como vacía entonces es un vértice aislado y queda como bloqueado (línea 55) y si es el caso que es un vértice bloqueado entonces se procede a eliminar en cada iteración del ciclo for la relación con sus hijos (líneas 57-60), la línea 58 invoca recursivamente la función "Elimina_Relación" para eliminar la relación contraria y después en la línea 59 se elimina el vértice en la posición *i* de la lista de *v.*
Finalmente, en la línea 27 de la función Conjunto es devuelvo el arreglo con todos los pares de vértices, donde existió un emparejamiento la pila de ese elemento en "Caminos" debe de ser mayor que 0 en longitud, todas las demás ubicaciones pueden llegar a tener pilas vacías.

Entonces como puede observarse para una grafica no-dirigida G es posible encontrar un conjunto S con un número par de vértices y $|S|/2$ caminos disjuntos.
Para el caso base donde el grafo tiene un solo vértice, en la línea 19 la condición if es evaluada como false (porque si existe un solo vértice entonces su lista de adyacencia no puede tener elemento alguno con quien relacionarse) y entonces se retorna la variable "Caminos" sin ningún tipo de información.
Para el caso donde el grafo tiene más de un vértice entonces se recorre la lista de adyacencia (línea 8) y la línea 19 se evalúa como verdadera porque como es una gráfica no-dirigida entonces existe una conexión mutua y no importa cual de los dos vértices se elija cualquier tiene una conexión al otro, después se repite todo el proceso anteriormente descrito encontrando así el conjunto S y los caminos correspondientes que unen los pares de vértices en el arreglo "Caminos".

Análisis de complejidad:
El algoritmo inicia en la línea 15 y en la línea 16 se invoca la función "Inicializa" en la cual las líneas 2,3,6 y 7 toman tiempo constante y el ciclo for de la línea 10 itera n+1 veces por lo tanto las líneas 11 y 12 se ejecutan n veces, donde n es el número de vértices del grafo, entonces la función "Inicializa" toma tiempo O(v) lo que significa que es lineal en el número de vértices del grafo. Continuando en la línea 17 esta se ejecuta solo una vez y el ciclo for de la línea 18 iterar n+1 veces, entonces el ciclo corre en tiempo O(v), después en la línea 23 se iteran por las aristas del grafo (línea 31) y en la línea 36 y 37 se ejecutan en el mismo tiempo ya que iteran por las aristas de los vértices actuales por lo tanto la complejidad que tiene "Emparejar_Vertices" es $O(E^2)$ por lo tanto la complejidad del algoritmo es $O(E^2)$, cuadrático en el número de las aristas.

2. - Tenemos un conjunto de n usuarios de una red que quieren conectarse a m puntos de acceso. Cada usuario y punto de acceso se identifica con una posición (x, y) en el plano. De acuerdo a las propiedades de los dispositivos de los usuarios y capacidades de los servidores (distancia, potencia del dispositivo, etc.), un usuario dado puede conectar a ciertos puntos de acceso y a otros no; esa información la tenemos disponible. También, para cada punto de acceso tenemos una capacidad máxima de usuarios que se pueden conectar a él.
Diseña un algoritmo de tiempo polinomial que, dadas las restricciones, decida si es factible conectar los usuarios a los puntos de acceso (cada usuario se conecta exactamente a un punto de acceso), y de serlo devuelva como se conectarían. Tip: Transforma el problema a uno de flujo máximo.

```
Obtener_Conexion(Adj,c)
G = Crear_Matriz(Adj,c,Adj.length)
<Gr,f> = Obtener_Relacion(G,Adj.length+1,Adj.lenght+2) //Obtiene la gráfica residual y el flujo maximo en forma de tupla
if f == usuarios then
    return <true,Gr>
else
    return <false,Gr>


Crear_Matriz(Adj,c,n)
    Matriz[n+2][n+2] //se crea un matriz con incremento en dos
                     //ya que se agregarán dos nuevos vertices s y t
    for i = 1 to n+2
        for j = 1 to n+2
            Matriz[i][j] = 0
        endfor
    endfor
    for i = 1 to n
        for j = 1 to Adj[i].length
            Matriz[i][Adj[i][j]] = 1
        endfor
    endfor
    for i = 1 to n
        if c[i] == 0 then
            Matriz[n+1][i] = 1
            usuarios += usuarios + 1 //variable global que cuanta los usuarios
        else
            Matriz[n+2][i] = c[i]
        endif
    endfor
return Matriz

Obtener_Relacion(G,s,t)
    Gr[t][t]          //se crea un gráfica residual de tamaño t x t
    Gr = Inicializa_Residual(Gr,G,t) //Se crea la gráfica residual Gr
    padres[t]         //inicializa el arreglo de padres con ceros
    flujo_maximo  = 0 //se establece el flujo maximo a cero
    while Bfs(Gr,s,t,padres) == true do //mientras existe un camino aumentante entonces se ejecutará el while
        camino = INFINITO //Se establace la variable "camino" con un valor infinito
        v = t
        while v != s do
            u      = padres[v]
            camino = min(camino,Gr[u][v])
            v      = u
        endwhile
        v = t
        while v != s do
            u = padres[v]
            Gr[u][v] -= camino
            Gr[v][u] += camino
            v          = u
        endwhile
        flujo_maximo = flujo_maximo + camino
    endwhile
return <Gr,flujo_maximo>


Inicializa_Residual(Gr,G,n)
    for i = 1 to n
        for j = 1 to n
            Gr[i][j] = G[i][j]
        endfor
    endfor

Bfs(Gr,s,t,padres,n)
    visitados[n] //inicializado con ceros
    cola.encolar(s)
    visitados[s] = true
    padres[s]    = -1
    while cola.es_vacia() == false do
        u = cola.saca()
        for i = 1 to n
            if visitados[i] == false && Gr[u][i] > 0 then
                cola.encolar(i)
                padre[i]     = u
                visitados[i] = true
            endif
        endfor
    endwhile
    if visitados[t] == true then
        return true
    endif
return false
```

Idea principal:
La descripción del problema dice que ya se tiene información sobre que usuarios se pueden conectar a que puntos de acceso por lo tanto se puede deducir que se tiene una gráfica como la Figura 3, en donde se puede observar que cada usuario puede o no puede estar conectado a un punto de acceso, como el problema pide que únicamente un usuario se conecte a un punto de acceso entonces podemos modificar este grafo de manera que parezco algo como un máximo emparejamiento, entonces agregando dos nuevos vértices s y t el gráfico puede lucir como la figura 4.

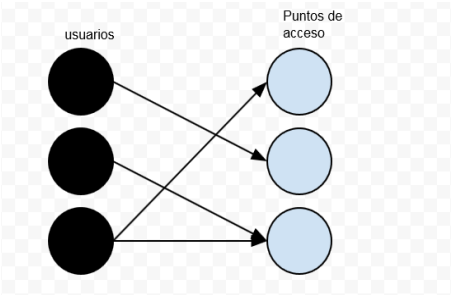


Fig. 3 Puntos de acceso conectados con sus respectivos usuarios

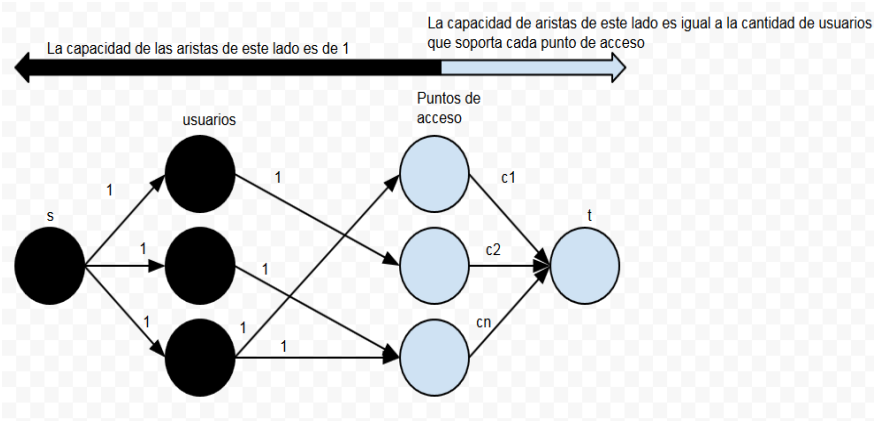


Fig. 4 Gráfica en donde se agregan dos nuevos nodos s y t, la gráfica es dirigida y los pesos de las aristas que salen de s y de los usuarios es de 1 y los pesos de las aristas que salen de los puntos de acceso a t es de ci, es decir la capacidad de recepción de cada punto de acceso.

Como se puede observar en la figura 4, se agregaron dos nuevos vértices y se agregaron pesos a las aristas, el problema dice que solo un usuario puede conectarse a un punto de acceso entonces a las aristas que salen de los vértices usuario se les asigna un peso de 1 y a los aristas que salen de s de igual manera se les asigna un peso de 1, por otro lado a las aristas que salen de los vértices de los puntos de acceso se les asigna un peso que es en realidad la capacidad de usuarios que puede soportar cada punto de acceso, entonces con este grafico modela podemos correar el algoritmo de Ford-Fulkerson y obtener el flujo máximo para la gráfica, finalmente la gráfica residual que regresa el algoritmo se puede recorrer para encontrar los caminos que van desde s a t y tienen peso cero, entonces serán las conexiones validas entre usuarios y puntos de acceso.

Corrección del algoritmo:
El algoritmo inicia en la línea 1 con la función “Obtener_conjunto” recibiendo por parámetro una lista de adyacencia y un arreglo con las capacidades de los vértices, después en la línea 2 se invoca a la función “Crear_Matriz” la cual tiene como objetivo crear una matriz de adyacencia del grafico dado por parámetro y agregar los dos nuevos vértices s y t así como también asigna los respectivos pesos a las aristas. En la línea 11 se crea un matriz de tamaño n+2, donde n es la cantidad de vértices que existen en el grafico y el 2 es por los dos nuevos vértices que se incorporaran al grafo, en la ubicación n+1 se encontrar el nuevo vértice s y en la ubicación n+2 el

nuevo vértice t, las líneas 13-17 inicializan cada valor de la matriz a cero, en las líneas 18-22 se establece para cada usuario que tiene relación con un punto de acceso entonces se pone 1 como peso entre esa arista, en las línea 23-29 se hace el otro proceso adicional que es de la fuente a todos los usuario se pone peso de 1 (línea 25) aquí en el arreglo c únicamente tendrán valores distintos de cero las ubicaciones que corresponden a los puntos de acceso, es decir esta información se da por hecho que así se pasa a el algoritmo y corresponde a la capacidad que soportan como vértices, entonces como a un usuario no se pueden conectar se le asigna un valor de 0 y a un punto de acceso un valor distinto de 1 y se asigna el peso a la arista que va del punto de acceso a t(línea 28), finalmente en la línea 31 se retorna la nueva gráfica que genera esta función.
Continuando en la línea 3 se invoca a la función "Obtener_Relación" a la cual se le envía la gráfica, y la ubicación de s y t. En la línea 34 se crea una nueva matriz que tendrá la función de la gráfica residual, se crea de tamaño t porque es el tamaño de la matriz de adyacencia para la gráfica nueva generada, en la línea 35 se invoca a la función "Inicializa_Residual" (línea 58) para se crea una copia de la gráfica (línea 61). En la línea 36 se inicializa un arreglo de tamaño t del tipo numérico que servirá para almacenar los padres de los vértices, en la línea 37 se crea una variable que será el flujo máximo de la gráfica calculado, en la línea 38 se entra en un while que va a iterar mientras exista un camino de s a t con aristas de peso mayor que cero, la búsqueda de este camino se hace con ayude de la función "Bfs" (línea 65), esta función recibe la gráfica residual, los vértices s y t, un arreglo llamado "padres" que será la ruta que encontrara de s a t y el tamaño de la gráfica, después de crea un arreglo para marcar los vértices visitados, en una cola se agregar a s como elemento, se marca a s como visitado y se asigna -1 como padre de s (líneas 66-69), en la línea 70 se iterar mientras que cola no sea vacía y en la línea 71 se obtiene un elemento de la cola para que en las líneas 72-78 se busque si existe una arista con peso mayor que cero que valla a un vértice aun no visitado, las líneas 70-79 muestran claramente el algoritmo de búsqueda en anchura finalmente en la línea 80 se verifica si fue alcanzado el vértice t, si fue así entonces se retorna verdadero o en caso contrario false. Continuando en la línea 39 de la función "Obtener_Relacion" se crea una variable "camino" inicializada con INFINITO, después en la línea 40 se iguala la variable v a t en las líneas 41-45 se recorre el camino (el arreglo padre que nos llenó la función "Bfs") que va desde t a s obteniendo el valor mínimo en las aristas, este valor mínimo será la cantidad de flujo que aumentara nuestro flujo máximo, después en la línea 46-52 se hace un recorrido similar de t a s pero ahora se restan la capacidad de las aristas que van de s a t y se suman a las aristas inversas que van de t a s, con esto logramos actualizar nuestra gráfica residual y finalmente se suma el valor "camino" al "flujo máximo" actual acumulado, finalmente cuando ya no existen "caminos aumentantes" desde s a t entonces el ciclo while finaliza y se retorna la tupla de información en la línea 55 que consiste de la gráfica residual y el flujo máximo. La línea 3 de la función "Obtener_Conexión" recibe esta información y verifica que el flujo máximo obtenido sea igual a la cantidad de usuarios que contiene el grafo con ayuda de la variable global usuarios (que fue actualizada en la línea 26) si el flujo es igual entonces quiere decir que es posible realizar una conexión completa y se regresa la tupla con un valor verdadero y la gráfica residual, en caso contrario se regresa una tupla con un valor falso. Como se comentaba anteriormente ya solo basta buscar los caminos en la grafica residual de s a t con aristas de peso cero para obtener las conexiones correctas.

Análisis de complejidad:
El algoritmo tiene complejidad cuadrática es fácil ver esto porque se está manejando una matriz de adyacencia que se crea nueva y una copia de esta, en las líneas 13-17 y 58-63 se evidencia lo anteriormente dicho, el algoritmo de BFS tiene complejidad O(V+E), es decir lineal en el número de vértices y aristas, por lo tanto, la complejidad del algoritmo es $O(V^2)$

3.- Dada una cadena S = c1, c2, …,cn, determina el mínimo número de caracteres que debes insertar a S para volverla un palíndromo. Una cadena es palíndromo si al invertirla queda igual. Por ejemplo, si S es raro, el número mínimo es 1. Tip: Puedes construir la solución de programación dinámica siguiendo los pasos vistos en clase, pero una solución potencialmente más corta es reducir ese problema a una instancia de un algoritmo de P.D. que hemos visto a profundidad en clase. Si te vas por el segundo camino, asegúrate de argumentar formalmente el porque es correcto.

Idea principal:
Corrección del algoritmo:
Análisis de complejidad:

4.- Considera el siguiente algoritmo ineficiente que decide si existe un camino entre dos vértices s y t de una gráfica dirigida G. Demuestra que el algoritmo es correcto. Además, analiza su complejidad y compárala con la de los algoritmos que vimos en clase, y explica por que efectivamente este algoritmo es ineficiente.

```
ALGORITHM Alcanzable(G, s, t)
   RETURN A(G,s,t,|V(G)|)
END Alcanzable

FUNCTION A(G,s,t,d)
   IF d=1 THEN
      IF s == t OR existe una arista dirigida (s,t) en G THEN
         RETURN TRUE
      ELSE
          RETURN FALSE
   ELSE
      FOR cada vértice v en G DO
         IF A(G,s,v,d/2) AND A(G,v,t,d/2) THEN
            RETURN TRUE
         ENDIF
      ENDFOR
      return FALSE
END A
```