Diseño y análisis de algoritmos

Práctica 2

Grupo 11

Mario Valdemaro García Roque
Alberto Cabello Álvarez

Cuestiones sobre los ejercicios

1. Discutir la aportación al coste teórico del algoritmo de Kruskal tanto de la gestión de la cola de prioridad como la del conjunto disjunto. Intentar llegar a la determinación individual de cada aportación.

Vamos a estudiar, en primer lugar, el aporte de cada parte del algorimo de Kruskal al coste total. A grandes rasgos, este se divide en dos grandes partes, la gestión y creación de la cola de prioridad y la gestión del conjunto disjunto.

Gestión de la cola de prioridad:

- El algoritmo de Kruskal usa una cola prioridad para mantener almacenados los ejes que se encuentran en el grafo y usa esta cola tanto para insertar como para extraer. El tipo de estructura mas eficiente para implementar este tipo de cola es un heap. Los heaps, como sabemos, tienen un coste medio de inserción y de extracción de O(logN). Por tanto, viendo que la operación de inserción se hace una vez por eje y la de extracción se realiza hasta que la cola este vacía, que es lo mismo que decir que se realiza una vez por eje, podemos concluir que el coste de insertar los ejes en el grafo es de **O(E*logV)** y el coste de extraerlos es también **O(E*logV).**

Gestión del conjunto disjunto:

- iniDS: esta operación crea el conjunto disjunto, puesta en detalle consiste en insertar todos los nodos del grafo en un conjunto disjunto, representado como una lista con valores a -1. El coste de esta operación es O(1) y se repite por cada nodo por tanto podemos concluir que su coste es **O(V)**.
- findDS: esta operación busca en el conjunto disjunto un vértice en concreto y nos devuelve el representante de su índice en el conjunto. El coste total de la operación es de O(log*N) que se puede concluir que es O(1) y al estar esta operación dentro de un bucle que se ejecuta E veces el coste total de esta operación en el algoritmo es aproximadamente 2*E ya que se realiza 2 veces, podemos concluir que su coste es **O(E)**.
- unionDS: esta operación une 2 caminos de un conjunto disjunto. El coste de esta operación es O(1) ya que solo tiene que asignar el árbol menor como subárbol del mayor. Dentro del algoritmo se encuentra en un bucle que se repite E veces, sin embargo solo se ejecuta esta operación V veces por tanto el coste de esta operación en el algoritmo es **O(V)**.

Por tanto podemos concluir que el coste del algoritmo depende de la operación mas costosa que como podemos ver se encuentra en la gestión de la cola de prioridad. Siendo el coste de Kruskal **O(E*logV).**

2. De acuerdo a lo anterior, ¿cual es el coste teórico del algoritmo de Kruskal dependiendo de que se aplique o no compresión de caminos?

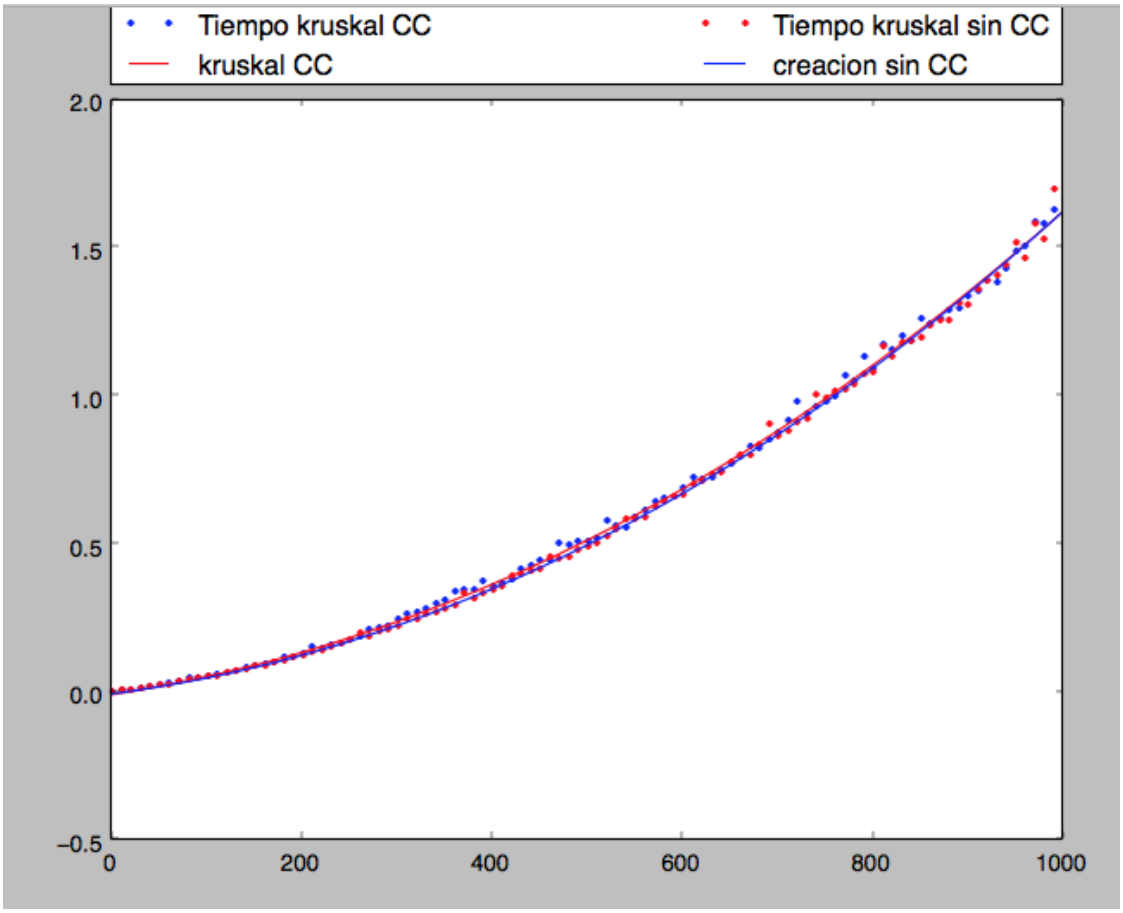
Como hemos concluido anteriormente, la parte más costosa de kruskal es la creación de la cola de prioridad con las ramas del grafo ordenadas. La operación de compresión de caminos se realiza en los finds, en la parte del algoritmo de construcción del arbol abarcador.

Según esto, la compresión de caminos tiene un efecto relativamente pequeño en el coste total del algoritmo. Es más, la compresión en sí tiene un coste extra al recorrer del vértice correspondiente a la raíz cambiando los valores del conjunto disjunto, que se espera recuperar al hacer los siguientes finds, donde se ahorrará tiempo al localizar más rápidamente los valores.

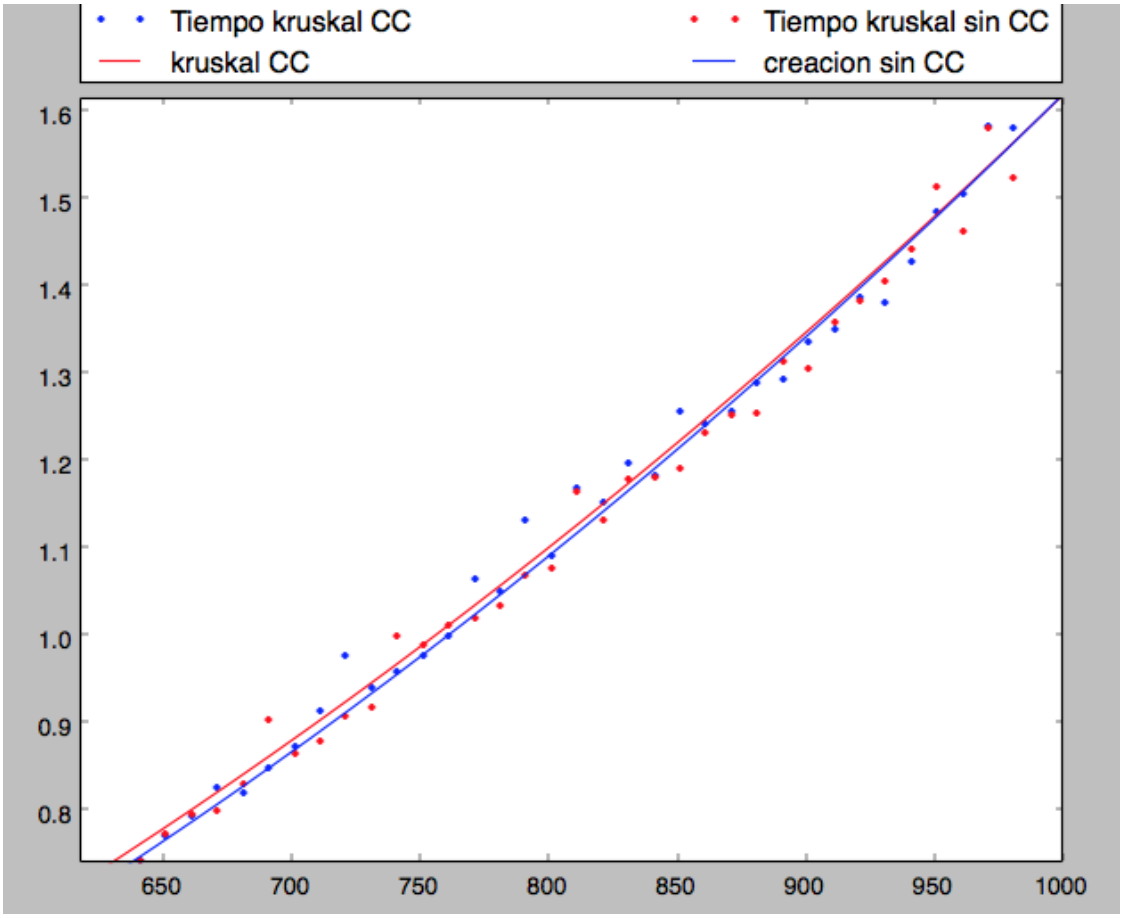
En definitiva, se trata de un tradeoff en el algoritmo, cambiando coste al realizar la compresión por un pequeño ahorro al encontrar de nuevo los nodos comprimidos.

Según los datos obtenidos, este ahorro no es demasiado significativo, es más, en ocasiones puede resultar directamente no rentable, un coste adicional que no llega a recuperarse porque no se hacen suficientes finds en el futuro como para compensar el coste extra.

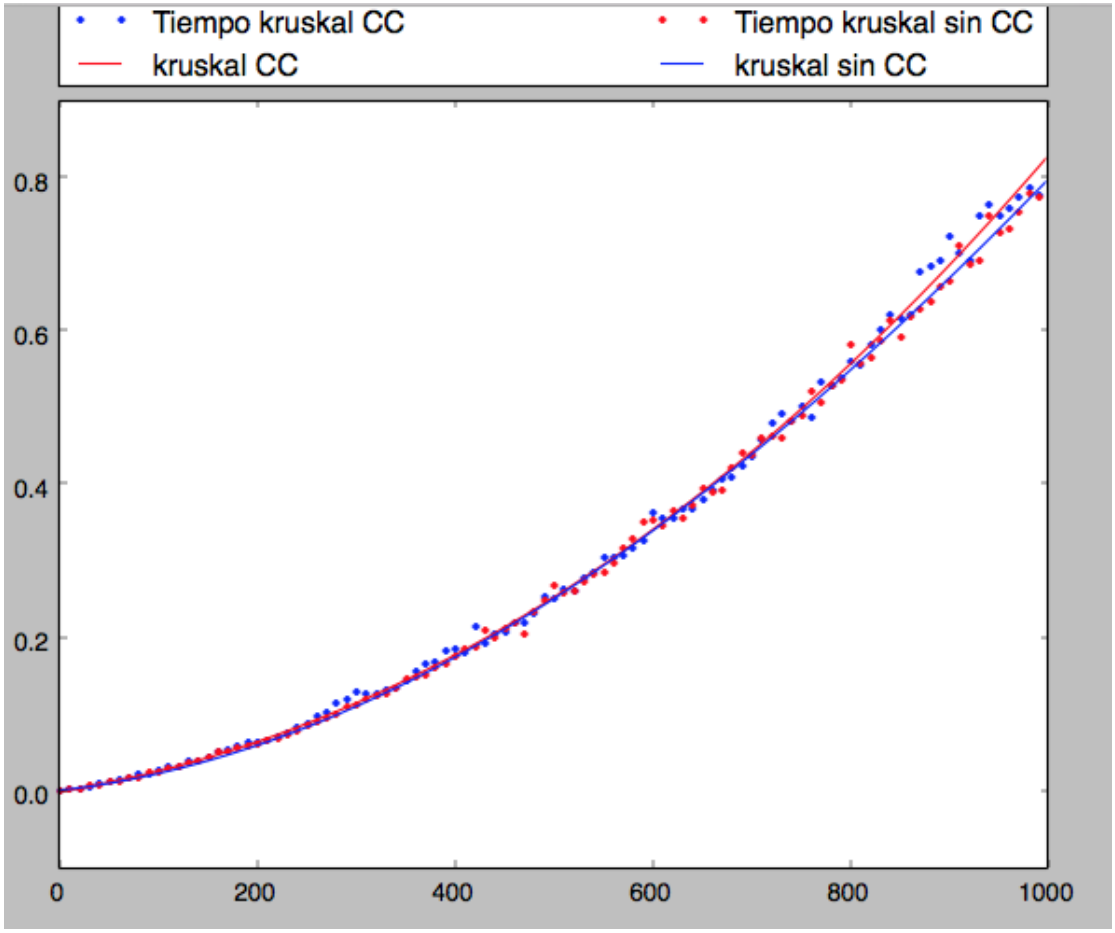
3. Contrastar la discusión anterior con las gráficas a elaborar mediante las funciones desarrolladas en la practica.



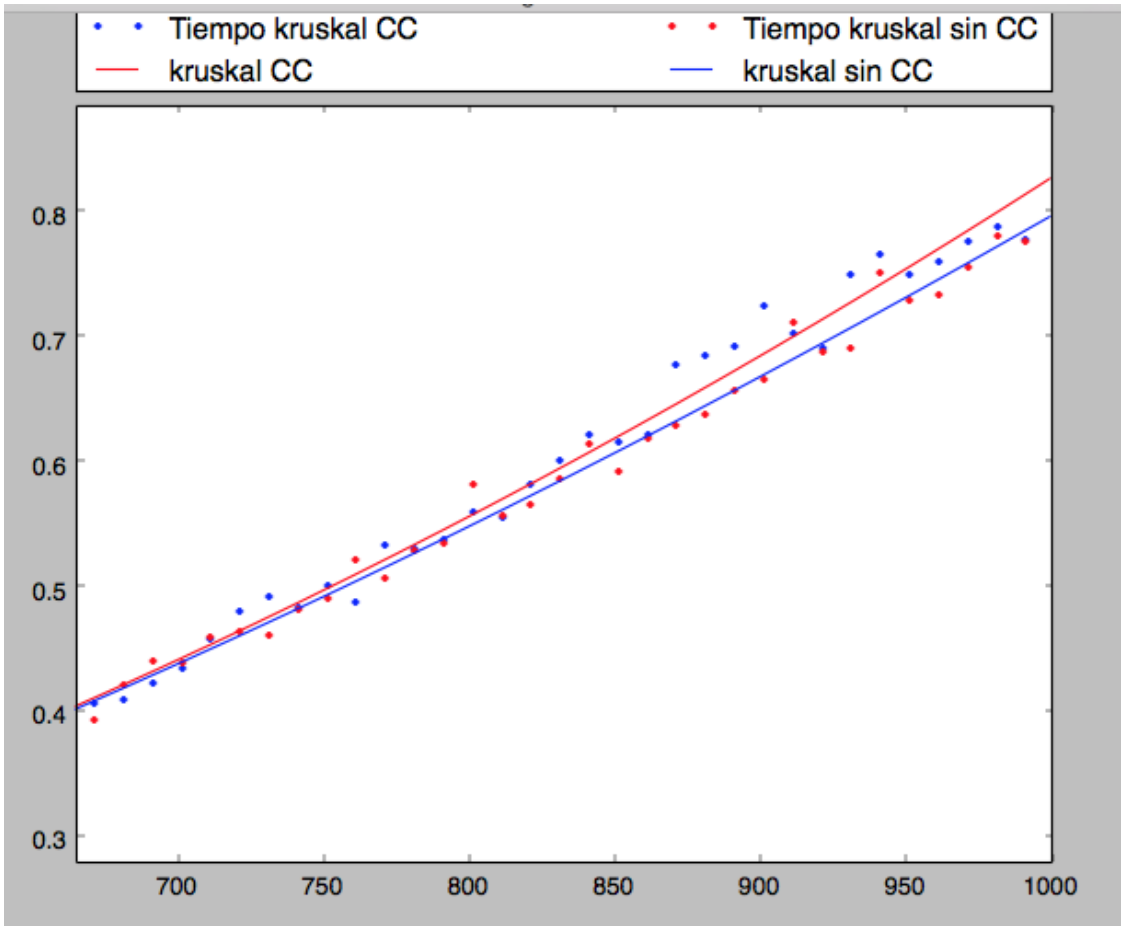
Comparativa tiempo kruskal con y sin compresión de caminos (sparse factor 1)



Zoom Comparativa tiempo kruskal con y sin compresión de caminos (sf 1)



Comparativa tiempo kruskal con y sin compresión de caminos (sparse factor 0.5)

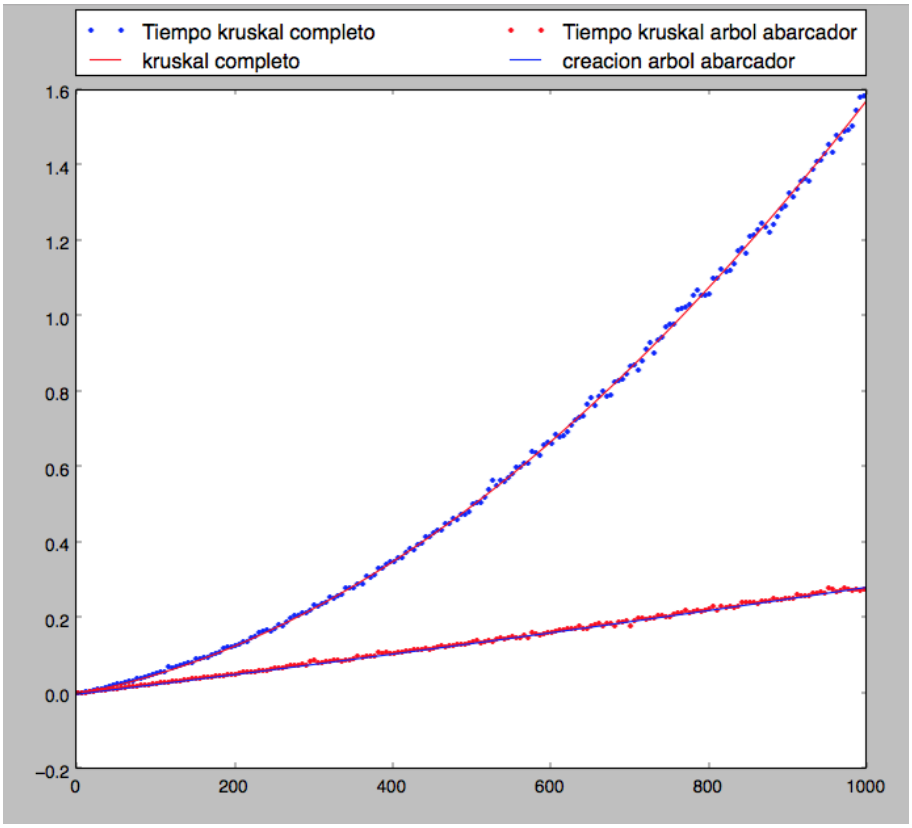


Zoom comparativa tiempo kruskal con y sin compresión de caminos (sf 0.5)

Como se puede observar en las gráficas superiores no hay demasiada diferencia entre usar compresión de caminos o no, es algo mejor no usar compresión de caminos pero podemos decir que el coste es básicamente el mismo. Esto se debe a que ,como hemos explicado, la operación mas costosa de todo el algoritmo es la gestión de la cola de prioridad y esta limita la eficiencia del algoritmo.

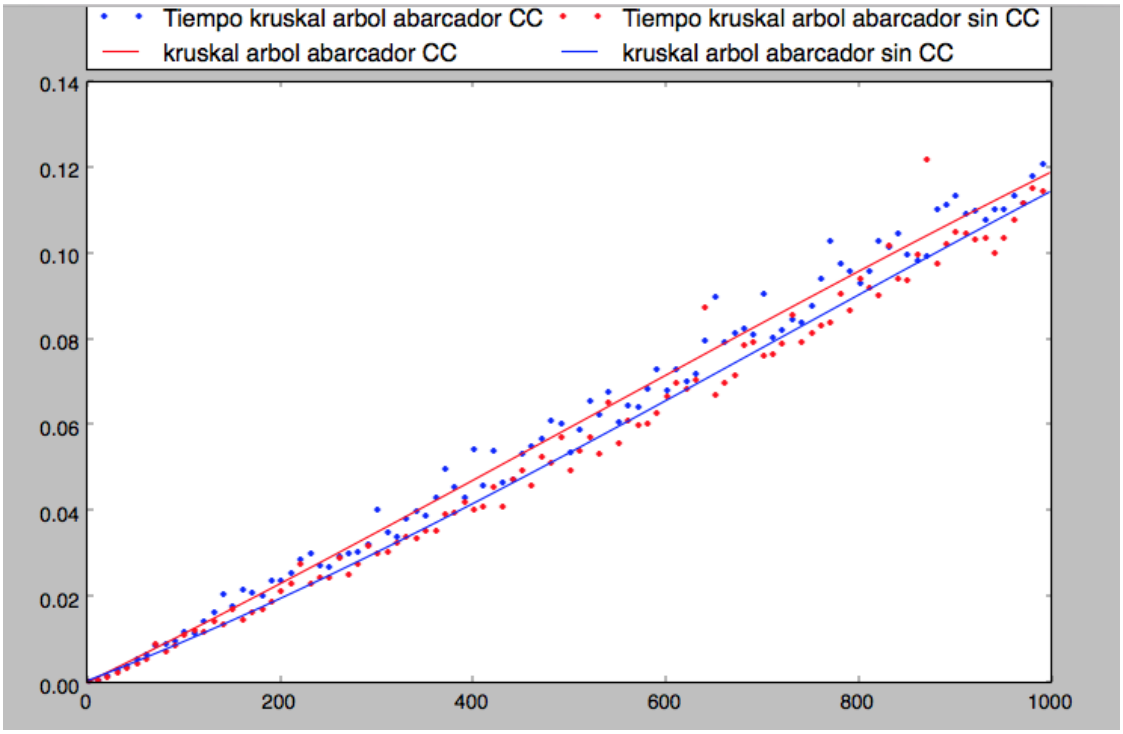
Por otro lado, la diferencia de los costes en función del sparse factor es la esperada. Según el grafo sea más denso, el efecto de la compresión de caminos es más efectiva, siendo la diferencia entre sin CC y con CC más pequeña

A continuación comprobaremos el coste de creación del arbol abarcador con respecto al total del algoritmo, y seguidamente, realizaremos los cálculos anteriores para ver como afecta la compresión de caminos a la parte de creación del arbol abarcador. Esto nos permitirá apreciar mejor esta diferencia ya que las operaciones de find se realizan únicamente en esta parte del algoritmo.



Tiempo kruskal completo vs creacion arbol abarcador

Nótese que la diferencia de costes entre el total y la creación del arbol abarcador es claramente apreciable. Puede ser interesante hacer un estudio más en cuanto a este diferencia en el que observemos cuanto tarda únicamente la parte del algoritmo correspondiente a crear el árbol abarcador con y sin compresión de caminos.



Comparativa tiempo creacion AA con y sin compresión de caminos (sf 0.5)

Como observamos no usar compresión de caminos vuelve a ser mas eficiente. Sin embargo aunque la diferencia entre usar o no CC esta más marcada en este caso que en el caso anterior, volvemos as concluir que su uso no mejora demasiado el tiempo de ejecución.

4. Opcional La información en las tablas de previos, descubrimiento y finalización es de utilidad para la clasificación de las ramas de árboles dirigidos como de árbol, ascendentes, descendentes y de cruce. Discutir esto intentando llegar a una tal clasificación.

Teniendo en cuenta el orden en el que se exploran los nodos en un grafo al realizar búsqueda por profundidad, esto es, teniendo en cuenta el orden en el que los valores de las tablas de previos, descubrimiento y finalización se obtienen, se puede llegar a una clasificación lógica de las ramas según estos parámetros.

Se clasifica cada rama según **u**, el nodo origen; **v**, el nodo destino y las tres tablas de la manera siguiente:
1: Si el vértice **u** es el previo del vértice **v**, entonces nos encontramos en el caso de una rama de **árbol**.
1.5: Para que sea cualquier otro tipo de rama, no tiene que cumplirse la condición anterior, es decir, el previo de **v** no es **u**.

2: Si **u** se ha descubierto antes que **v** y todavía no se ha finalizado la exploración de **u**, entonces es una rama **descendente**. Esto quiere decir que ya se conoce otro camino hasta **v** desde otro nodo distinto de **u** y el camino **u-v** es uno descendente.

3: Si **v** se ha descubierto antes que **u** y **v** aún está siendo explorado, quiere decir que el camino **u-v** es un camino **ascendente**, ya que **v** es superior a **u** en el grafo y la arista parte de **u** hacia un punto de rango superior.

4: Para cualquier otro caso, es un camino de **cruce**, ya que no cumple las características de cualquier de los otros enlaces.

En la práctica, hemos empleado la siguiente función para clasificar las aristas durante la búsqueda en profundidad. En particular nos sirve para distinguir los ascending edges, para comprobar si un grafo es acíclico, ya que sabemos que un grafo es DAG si y solo si no tiene ramas ascendentes.

```
def tRama(u, v, d, f, p):
    """
    Funcion que comprueba el tipo de rama que es u-v. Se ejecuta
durante busqueda en profundidad
    con los valores de d, f, p correspondientes
    :param u: vertice 1
    :param v: vertice 2
    :param d: tabla descubrimiento
    :param f: tabla finalizacion
    :param p: tabla previos
    :return: 1: TREE 2: DESCENDENTE 3:ASCENDENTE 4:CICLO
    """
    if p[v] == u:
        return 1  # TREE
    elif p[v] != u:
        if d[u] < d[v] and f[u] == np.inf:
            return 2  # DESCENDENTE
        elif d[v] < d[u] and f[v] == np.inf:
            return 3  # ASCENDENTE
    return 4  # CICLO/CRUCE
```

Anexo: lista de las funciones utilizadas y docstring

```
def randMatrUndPosWGraph(nNodes, sparseFactor, maxWeight=50.):
    """
    Genera una matriz de adyacencia de un grafo no dirigido
ponderado.
    :param nNodes: Numero de nodos del grafo
    :param sparseFactor: Proporcion de ramas. (0 - 1)
    :param maxWeight: Maximo peso de una arista, por defecto, 50.
    :return: Matriz de adyacencia
    """

def checkUndirectedM(mG):
    """
    Comprueba si el grafo dG es no dirigido. Version con indexing
    :param mG: Grafo como matriz
    :return: True si es un grafo no dirigido, False, si es dirigido.
    """

def checkUndirectedD(dG):
    """
    Comprueba si el grafo dG es no dirigido.
    :param dG: Grafo como diccionario
    :return: True si es un grafo no dirigido, False, si es dirigido.
    """

def initCD(N):
    """
    Inicializa un conjunto disjunto a modo de lista de tamano n con
valores -1
    :param N: Tamano del conjunto disjunto
    :return: Lista con N -1's
    """

def union(rep1, rep2, pS):
    """
    Realiza la union por rangos entre rep1 y rep2 en el conjunto
disjunto pS
    :param rep1: representante 1
    :param rep2: representante 2
    :param pS: conjunto disjunto
    :return: representante de la union segun union por rangos
    """

def find(ind, pS, flagCC):
    """
    Realiza la operacion find para el conjunto disjunto pS
    :param ind: indece que buscar en el CD
    :param pS: Conjunto Disjunto
    :param flagCC: Valor booleano que indica si hacer compresion de
caminos
    :return: Representante del indice ind en pS
    """
```

```
def insertPQMatriz(mG, Q):
    """
    Realiza la insercion de las ramas en la cola de prioridad de
Kruskal
    :param mG: grafo como matriz de adyacencia
    :param Q: Cola de prioridad a la que insertar las ramas
    :return: cola de prioridad Q
    """

def insertPQ(dG, Q):
    """
    Realiza la insercion de las ramas en la cola de prioridad de
Kruskal
    :param dG: grafo como diccionario
    :param Q: Cola de prioridad a la que insertar las ramas
    :return: cola de prioridad Q
    """

def kruskal(dG, flagCC=True):
    """
    Aplica el algoritmo de Kruskal sobre el grafo dG
    :param dG: grafo como diccionario
    :param flagCC: Realizar o no compresion de caminos
    :return: Arbol abarcador minimo resultante del algoritmo
    """

def kruskal2(dG, flagCC=True):
    """
    Aplica el algoritmo de Kruskal sobre el grafo dG devolviendo
ademas el tiempo usado para crear el arbol abarcador
    :param dG: grafo como diccionario
    :param flagCC: Realizar o no compresion de caminos
    :return: Arbol abarcador minimo resultante del algoritmo, tiempo
usado en su construccion
    """

def timeKruskal(nGraphs, nNodesIni, nNodesFin, step, sparseFactor,
flagCC=True):
    """
    Mide los tiempos de ejecucion para el algoritmo de kruskal
    :param nGraphs: numero de grafos de cada tamano a sobre los que
buscar
    :param nNodesIni: tamano minimo del grafo
    :param nNodesFin: tamano maximo del grafo
    :param step: incremento que sumar de tamano de grafo al siguiente
    :param sparseFactor: Proporcion de ramas en los grafos. Valor
decimal de 0 a 1
    :param flagCC: Realizar o no compresion de caminos. True por
defecto
    :return: Lista con los tiempos tardados para cada tamano de grafo
    """
```

```
def timeKruskal02(nGraphs, nNodesIni, nNodesFin, step, sparseFactor,
flagCC=True):
    """
    Mide los tiempos de ejecucion para crear el arbol abarcador del
algoritmo de kruskal
    :param nGraphs: numero de grafos de cada tamano a sobre los que
buscar
    :param nNodesIni: tamano minimo del grafo
    :param nNodesFin: tamano maximo del grafo
    :param step: incremento que sumar de tamano de grafo al siguiente
    :param sparseFactor: Proporcion de ramas en los grafos. Valor
decimal de 0 a 1
    :param flagCC: Realizar o no compresion de caminos. True por
defecto
    :return: Lista con los tiempos tardados para cada tamano de grafo
    """

def fitPlotKruskal(nGraphs, nNodesIni, nNodesFin, step, sparseFactor,
flagCC=True):
    """
    Compara los tiempos de kruskal y realiza las graficas
    :param nGraphs: numero de grafos de cada tamano a sobre los que
buscar
    :param nNodesIni: tamano minimo del grafo
    :param nNodesFin: tamano maximo del grafo
    :param step: incremento que sumar de tamano de grafo al siguiente
    :param sparseFactor: Proporcion de ramas en los grafos. Valor
decimal de 0 a 1
    :param flagCC: Realizar o no compresion de caminos. True por
defecto
    :return: Lista con los tiempos tardados para cada tamano de grafo
    """


# BP, OT y DM's
def incAdy(dG):
    """
    Calcula las tablas de incidencia y adyacencia para los nodos de
dG
    :param dG: Grafo como diccionario
    :return: ([tabla de incidencia],[tabla de adyacencia])
    """

def drBP(dG, u=0):
    """
    Driver que ejecuta busqueda en profundidad
    :param dG: Grafo como diccionario
    :param u: Nodo por el que comenzar la busqueda. 0 por defecto.
    :return: ([descubrimiento][finalizacion][previos])
    """

def BP(u, dG, d, f, p, n):
    """
    Algoritmo de busqueda en profundidad llamado desde el driver
    :param u: Nodo actual
    :param dG: Grafo como diccionario
    :param d: tabla de descubrimientos
```

```
    :param f: tabla de finalizaciones
    :param p: tabla de previos
    :param n: contador con el paso actual para calcular d y f
    :return: n con el paso actual
    """

def tRama(u, v, d, f, p):
    """
    Funcion que comprueba el tipo de rama que es u-v. Se ejecuta
durante busqueda en profundidad
    con los valores de d, f, p correspondientes
    :param u: vertice 1
    :param v: vertice 2
    :param d: tabla descubrimiento
    :param f: tabla finalizacion
    :param p: tabla previos
    :return: 1: TREE 2: DESCENDENTE 3:ASCENDENTE 4:CICLO
    """

def BPasc(u, dG, d, f, p, a, n):
    """
    Algoritmo de busqueda en profundidad con deteccion de ascending
edges llamado desde el driver
    :param u: Nodo actual
    :param dG: Grafo como diccionario
    :param d: tabla de descubrimientos
    :param f: tabla de finalizaciones
    :param p: tabla de previos
    :param a: tabla de ascending edges
    :param n: contador con el paso actual para calcular d y f
    :return: n con el paso actual
    """

def drBPasc(dG, u=0):
    """
    Driver para comenzar BP con deteccion de ascending edges.
    :param dG: Grafo como diccionario
    :param u: Nodo por el que comenzar la busqueda. 0 por defecto.
    :return: ([descubrimiento][finalizacion][previos][ascending
edges])
    """

def DAG(dG):
    """
    Funcion que comprueba si un grafo tiene ciclos, un grafo es
aciclico si no tiene ascending edges
    :param dG:grafo
    :return:True si es DAG(No tiene ciclos), False si no es DAG, no
tiene ciclos.
    """

def OT(dG):
    """
    Funcion que devuelve la ordenacion por orden topologico de dG
    :param dG: grafo
    :return: Lista con el orden topologico del grafo o lista vacia si
no es DAG
    """
```

```
def distMinSingleSourceDAG(dG):
    """
    Compruaba que dG es un DAG con una unica fuente y computa las
distancias minimas a los demas nodos
    :param dG: Grafo
    :return: Tupla con lista con las distancias minimas y tabla de
previos. Si tiene mas de una fuente o no es DAG, devuelve ([np.inf] *
n, [None] * n)
    """
```