Problemas sobre caminos en un grafo:

Hay un gran número de problemas sobre caminos en grafos dirigidos y en grafos no dirigidos. Algunos plantean determinar si dado un par de vértices, o todos los pares de vértices, hay o no camino entre ellos.Otros trabajan sobre grafos con aristas ponderadas, con vértices ponderados o con ponderación en ambos elementos (una ponderacion es un valor asociado a una arista o a un vértice), o a ambos. En esta sección trataremos algunos de los problemas más comunes sobre caminos en grafos con aristas ponderadas.

Problema de los caminos más cortos con un solo origen.

Dado un grafo dirigido **G = (V, A)**, en el cual cada arco tiene asociado un costo no negativo, y donde un vértice se considera como origen, el problema de los "caminos más cortos con un solo origen" consiste en determinar el costo del camino más corto desde el vértice considerado origen a todos los otros vértices de **V**.

Este es uno de los problemas más comunes que se plantean para los grafos dirigidos con aristas ponderadas (es decir con peso en las aristas).

El peso de las aristas debe no negativo. El grafo puede no ser conexo, en este caso la solución indicara que no hay camino que permita alcanzar algunos de los vértices.
El la 'longitud' o 'costo' de un camino es la sumatoria de los pesos de las aristas que lo conforman.
El modelo de grafo dirigido con aristas ponderadas no negativas puede, por ejemplo, corresponder a un mapa de vuelos en el cual cada vértice represente una ciudad, y cada arista (**v**,**w**) una ruta aérea de la ciudad **v**a **w**.
El peso de la arista (v,w) puede ser el tiempo que se requiere para volar de **v** a **w** (y para justificar que eventualmente el peso de (v,w) sea diferente del de (w,v) se puede considerar que eso se debe a la influencia de los vientos. En este caso el problema trataría de encontrar los tiempos mas cortos para, partiendo de una ciudad en particular, alcanzar cada una de las otras.
Otra posibilidad es considerar que los pesos corresponden a las tarifas de esos tramos, y que en ocasiones, el precio del tramo (v,w) es diferente del (w,v). Para esta situación, el problema plantea la búsqueda de los costos mas bajos a pagar para partiendo de una ciudad en particular, llegar a cada una de las otras.

Algoritmo de Dijkstra para el calculo de los caminos mínimos

Este algoritmo se caracteriza por usar una estrategia *greedy*, voraz (o ávida).

En esta estrategia se trata de optimizar (alcanzar el máximo o el mínimo) de una función objetivo, la cual depende de ciertas variables, cada una de las cuales tiene un determinado dominio y está sujeta a restricciones.

En cada etapa se toma una decisión que no tiene vuelta atrás, y que involucra elegir el elemento más promisorio (es decir, el que parece ofrecer más posibilidades de mejorar la función objetivo) de un conjunto y se analizan ciertas restricciones

asociadas a cada variable de la función objetivo para ver si se verifican mejoras en ella.

La solución puede expresarse como una sucesión de decisiones, y en cada etapa de la estrategia se elegirá la mejor opción de las disponibles (óptimo local), analizándose luego si esta elección verifica las restricciones, constituyendo una solución factible.

La estrategia greedy, muy interesante en sí, no sirve para cualquier problema: se debe tratar de un problema de optimización de una función; pero aún en este caso, no asegura que se llegue al optimo global (podría llevarnos a un óptimo local) ni tampoco, en algunos casos, llegar a una solución factible: el problema en cuestión debe verificar ciertas condiciones para que quede garantizado que la estrategia greedy funcione. En el caso del problema de los caminos mínimos con un mismo origen en un grafo ponderado con ponderaciones no negativas, está probado que la estrategia funciona.

Se tiene, entonces, un grafo dirigido **G = (V, A)**
El conjunto de vértices es **V = {1, 2, …, n}**
Se considerara que el vértice de partida es 1.

Cuando la arista correspondiente no exista, consideraremos que el valor de ese elemento del arreglo será un valor mucho mayor que cualquier costo real, es decir 'infinito'.D es el array en el cual se van almacenando la sucesivas mejoras de tiempo obtenidas por el algoritmo y donde finalmente queden almacenados los tiempos mas bajos.

El algoritmo de Dijkstra se puede expresar de modo informal así:

V es el conjunto de vértices del grafo.
S es el conjunto de vértices visitados.
D es el vector de tiempos mínimos, que se actualiza en cada etapa.
En el siguiente algoritmo se considera que el vértice de partida es el 1, y los vértices están numerados de 1 hasta N

```
 {
  Inicializar S = { 1 }
  Para cada  i desde 2 hasta N hacer
                  Asignar M[1, i] a D[i]
  Para cada i entre 1 y n - 1 hacer
        {
        Elegir un vértice w en V – S tal que D[w] sea el mínimo de todos los D[i].
        Agregar w a S.
        Para cada vértice v adyacente a w en V – S hacer:
        D[v]=mín ( D[v], D[w] + Coste de [w,v])
        }
}
```

Ejercicios:

1. Considerar una implementación en la cual los pesos de las aristas estén almacenados en una tabla de pesos, y la información sobre los vértices esté almacenada en un array que se recorra 'de punta a punta' para realizar cada elección del vértice más conveniente. Escribir esa versión del algoritmo..

¿Cuál sería el coste temporal de la implementación del algoritmo? Justificar la respuesta.

2. Considerar una implementación del grafo con listas de adyacencia que almacene en cada nodo adyacente, además de la etiqueta del mismo, el peso de la arista correspondiente. Considerar también que los vértices están ordenados según el coste de alcanzarlos, en un heap.
   Pensar entonces la implementación de la solución de esta forma:

   - Cargar en un heap de mínimo, la información sobre los nodos no visitados y el costo de alcanzar cada uno de ellos desde el vértice origen. Los nodos inalcanzables desde el origen tendrán un valor ∞ asociado.
   - Mientras el heap tenga elementos,
     - remover la raíz y restaurar el heap
     - para cada nodo adyacente al removido (revisar la lista de adyacencia de dicho nodo) determinar si hay mejora. Si la hay, y el nodo no fue visitado, actualizar los valores del heap, asegurando que quede restaurado. Para poder realizar esto, se deberá considerar que el heap en cuestión debe tener además una operación que permita la actualización de prioridad.

a) Escribir la implementación correspondiente del algoritmo
b) ¿Cuál es el coste de armado del heap?
c) ¿Cuántas actualizaciones se harían sobre el heap?
d) ¿Cuál es el coste total de esta implementación? (Escribir el pseudocódigo de la misma y analizar el coste)
e) ¿Cuándo conviene la primera implementación y cuándo esta otra?

Problema de los caminos más cortos entre todos los pares de vértices.

Se trata de determinar los caminos mínimos que unen cada vértice del grafo con todos los otros.

Si bien lo más común es que se aplique a grafos que no tengan aristas negativas, la restricción necesaria es que no existan ciclos con costos negativos.

Este problema no verifica las condiciones que aseguran que un algoritmo “greedy” funciones, por lo cual no puede aplicarse esa estrategia. Sin embargo, verifica las condiciones que permiten aplicar un algoritmo de los llamados de “programación dinámica”

La programación dinámica es una estrategia de resolución de problemas que suele aplicarse a problemas de optimización que verifican ciertas condiciones.Llega a la solución a través de una secuencia decisoria que debe verificar la condición de que “en la secuencia de decisiones óptima toda subsecuencia ha de ser también óptima” (lo cual, dicho sea de paso, se verificaba también en la estrategia “greeddy”).

En estos problemas se realiza una división de problemas en otros problemas menores, pero, a diferencia de lo que ocurría en “Divide y Vencerás”, estos problemas no son independientes entre si, sino que tienen subproblemas en común, hay un ‘solapamiento’ de problemas.

La técnica consiste en resolver y almacenar las soluciones de las zonas solapadas para no volver a realizar los mismos cálculos.

En general en esta estrategia se llega a la solución realizando comparaciones y actualizaciones en datos que han sido tabulados (las soluciones de los subproblemas).

Algoritmo de Floyd

Este algoritmo se basa en una técnica de programación dinámica que almacena en cada iteración el mejor camino entre el que pasa por el nodo intermedio k y el que va directamente del nodo i al nodo j.

La programación dinámica, como se dijo, se basa en dividir el problema en subproblemas, los cuales deben tener ‘zonas comunes’ que constituyen a su vez nuevos subproblemas con las mismas características. El método procede resolviendo estos subproblemas comunes en primer lugar, almacenando los resultados y utilizándolos en la resolución de los problemas mayores. La estrategia de PD es aplicable a este problema por cumplirse el principio de optimalidad para las soluciones factibles.

Para determinar cuál es el mejor camino considera mínimo (D[v] , D[u] + M(u,v)).

Inicialmente se carga una matriz de NxN con los pesos correspondientes a las aristas, asignándose 0 a M[i][i] para todo i .

El método lleva a cabo N iteraciones del proceso en cada una de las cuales se evalúa la eventual mejora de los tiempos por la consideración del vértice i (que variara entre 1 y N a lo largo del proceso) como intermedio.

Este es el algoritmo de Floyd expresado de manera informal: (N es el número de vértices)

```
Floyd
{
Inicializar A con ceros en la diagonal principal y los pesos indicados por el grafo para
cada uno de los restantes elementos. Si una arista no existe, se coloca peso infinito.
Para k desde 1 hasta N
   Para i desde 1 hasta N
      Para j desde 1hasta N
        A[i][j] = mínimo (A[i][j] , A[i][j] + A[k][j])
}
```

Ejercicios

a) Relacionar este algoritmo con el de Warshall para la cerradura transitiva

Algunos problemas clásicos de caminos sobre Grafos no dirigidos.

Definición 1: un **árbol libre** es un grafo no dirigido conexo sin ciclos.

Se verifica que en todo árbol libre con N vértices (N>1), el árbol contiene N-1 aristas.
Si se agrega una arista a un árbol libre, aparece un ciclo.

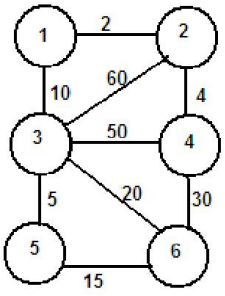
Definición 2: dado **G = (V, A)** grafo no dirigido conexo, un **árbol abarcador** para G, es un árbol libre que conecta todos los vértices de V.

Para un mismo grafo se pueden obtener diferentes árboles abarcadores.

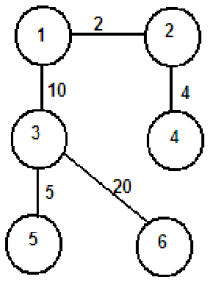
Definición 3: dado un grafo G=(V,A) no dirigido con aristas ponderadas(es decir, que cada arista (v, w) de A tiene un costo asociado, C(v, w)), se llama **árbol abarcador de coste mínimo** al árbol abarcador para G que verifica que la sumatoria de los pesos de las aristas es mínima.

El árbol abarcador de costo mínimo, puede no ser único.

Ejemplo: Para este grafo



El árbol abarcador de coste mínimo es;



Cuyo coste es 2+10+4+5+20=41

Algoritmos que permiten obtener el arbol de expansion de coste minimo para un grafo dado

Existen varios algoritmos que permiten obtener el árbol de expansión de coste mínimo. Se detallaran dos de ellos: el algoritmo de Prim y el de Kruskal.

En ambos casos se aplica la estrategia 'greedy' o voraz para alcanzar el óptimo.

Algoritmo de Prim.

Consideremos un grafo G(V,A) tal que V = { 1, 2, 3, …. , n}.

El algoritmo parte de un conjunto **U** al que se le asigna inicialmente uno de los nodos del grafo; hemos elegido para el pseudocódigo, el 1.

U = {1}.

En dicho conjunto crece un árbol abarcador de costo mínimo, arista por arista.

En cada paso, localiza la arista de menor costo **(u, v)** (con **u** en **U** y **v** en **V - U**) que conecta a **U** y **V - U**.

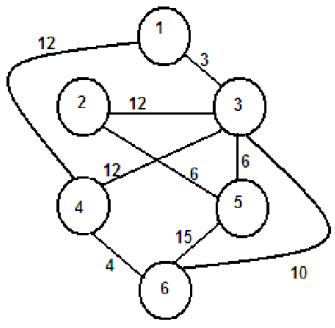
Una vez localizada la arista agrega el vértice **v** al conjunto **U**, creciendo de esta forma el árbol, porque la arista en cuestión es de costo mínimo

Esto se repite hasta que **U = V** (condición de parada).

Llamando G al grafo y A al conjunto de aristas del árbol abarcador de coste mínimo, es

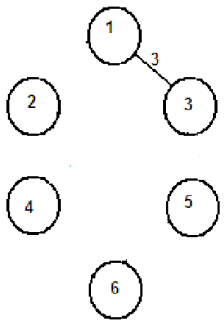
```
{
  A se inicializa en vacío
  U = {1}                          //se almacena vértice inicial
  Mientras U distinto de  V hacer:
   {
      Determinar la arista de costo mínimo, (u,v) tal que u ϵ U y v ϵ (V-U)
      Agregar a A la arista (u, v)
      Agregar a U el vértice v
   }
}
```

Gráficamente:consideremos partir del siguiente grafo:



Aplicando el algoritmo de Prim es:

Paso 1: se selecciona la menor arista que conecte el vértice 1 con alguno de los otros

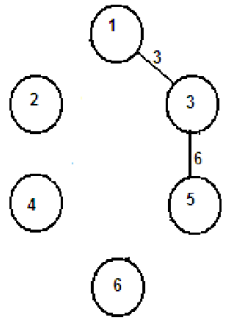


Las aristas disponibles para la conexión tienen pesos 3 y 12.

L a de menor peso vale 3.

U={1, 3}

Paso 2: se elige la arista de menor costo entre las que ligan alguno de los vértices del conjunto {1,3} con otro de los vértices de G

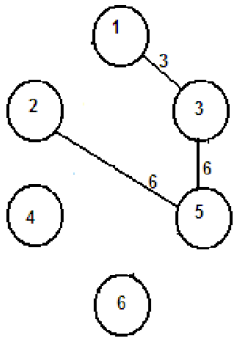


Las aristas disponibles para la conexión tienen pesos 12 (hay tres) y 6.

L a de menor peso vale 6.

U={1,3,5}

Paso 3: se selecciona una arista de peso mínimo que enlace un vértice de {1,3,5} con alguno de los otros nodos.

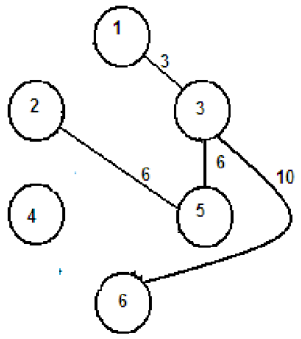


Las aristas disponibles para la conexión tienen pesos 12 (hay tres), 6 y 15

L a de menor peso vale 6.

U={1, 2, 3, 5}

Paso 4: se selecciona una arista de peso mínimo que enlace un vértice de {1, 2, 3, 5} con alguno de los otros nodos.

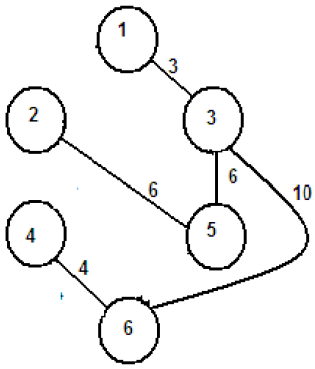


Las aristas disponibles para la conexión tienen pesos 12 (hay dos), 10 y 15

L a de menor peso vale 10.

U={1,2, 3, 5, 6}

Paso 5: Paso 4: se selecciona una arista de peso mínimo que enlace un vértice de {1, 2, 3, 5, 6} con alguno de los otros nodos.



Las aristas disponibles para la conexión tienen pesos 12 (hay dos), y 4

L a de menor peso vale 4.

U={1,2, 3, 4, 5, 6}

Se ha obtenido el árbol de expansión de coste mínimo, con un costo de 3+6+6+4+10=29

Como puede observarse, el tiempo de ejecución del algoritmo depende de la búsqueda del arco de peso mínimo.

Ejercicio:

a) Considerar que los arcos se ordenan en una cola con prioridades. Escribir la implementación correspondiente a Prim y evaluar su coste.

Algoritmo de Kruskal.

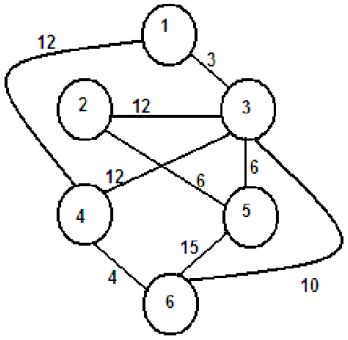
Dado un grafo conexo **G = (V, A)**, con **V = {1, 2, … , n}** y una función de costo **C** definida en las aristas de **A**, el algoritmo de Kruskal propone empezar con un grafo que contenga a todos los vértices de **V** pero no a sus aristas (es decir se comienza con un grafo con N subgrafos conexos, cada uno de los cuales corresponde a un vértice),llamémoslo T

La construcción del árbol abarcador de coste mínimo se hace de este modo: se itera sobre el conjunto de las aristas de G, ordenadas por peso.

En cada iteración se halla una arista de costo mínimo que conecte a dos componentes conexas diferentes y se le añade dicha arista a T.

De este modo se forma una nueva componente conexa a partir de otras dos. Cuando todos los vértices pertenezcan a una misma componente, el algoritmo finaliza, ya que esta componente única será un árbol abarcador de costo mínimo para el grafo G dado.

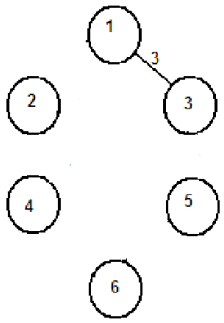
Para este grafo:



La lista de aristas en orden creciente de pesos es:

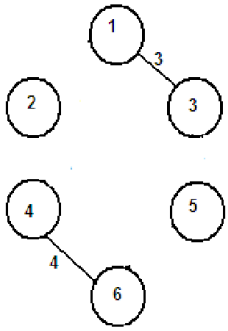
Costo	Aristas
3	(1, 3)
4	(4, 6)
6	(2, 5) , (3,5)
10	(3, 6)
12	(1, 4), (2, 3), (3, 4)
15	(5, 6)

Ahora, veamos cuáles son las componentes conexas que se forman en cada paso del algoritmo:



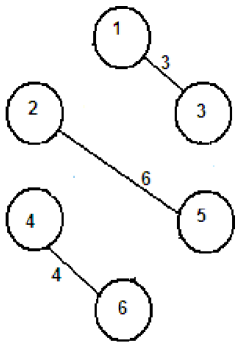
Se elige la arista de peso mas bajo, que vale 3

Ahora hay 5 componentes conexas



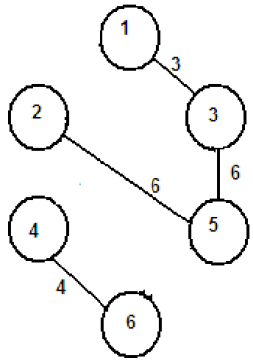
La arista de peso siguiente vale 4 y no forma ciclo. Se agrega.

Ahora hay 4 componentes conexas



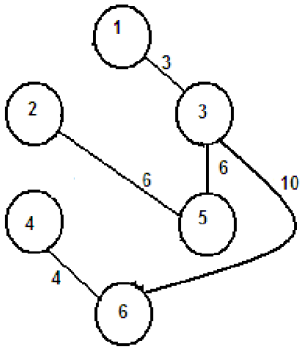
Las aristas de peso siguiente valen 6.Ninguna de ellas forma ciclo. Se elige una de ellas y se agrega.

Ahora hay 3 componentes conexas



La siguiente arista también vale 6 y no forma ciclo. Se agrega.

Ahora hay 2 componentes conexas



Las siguiente arista vale 10 y no forma ciclo. Se agrega.

Ahora hay 1 sola componente conexa.

Como se observa, es un algoritmo voraz que en cada iteración añade un arco de peso mínimo. Parte de un *bosque* que inicialmente contiene un árbol por cada vértice. Los arcos se preprocesan (como habitualmente en los algoritmos greedy), ordenándolos de forma creciente por su peso. Se continúa mientras el bosque inicial no sea un árbol y existan arcos para añadir. Si el arco elegido provoca un bucle, se desecha y se continua con el siguiente.

Para determinar si un arco forma bucle se suele trabajar con conjuntos para representar los diferentes arboles. Cada conjunto tiene una identificación. Si una arista tiene como origen y extremo vértices del mismo conjunto, forma bucle y debe ser desechada. Si los identificadores de los conjuntos son distintos, la arista se agrega y se debe proceder a unir los conjuntos, puesto que ahora ambos vértices estarán en el mismo.

Estos algoritmos se conocen generalmente como Union (el que permite unir dos conjuntos) y Find (el que determina a qué conjunto pertenece un vértice). Obviamente el análisis de los mismos depende de las estructuras que se usen para implementar los conjuntos.

FIGURE 8.12 Concatenating two circular linked lists.

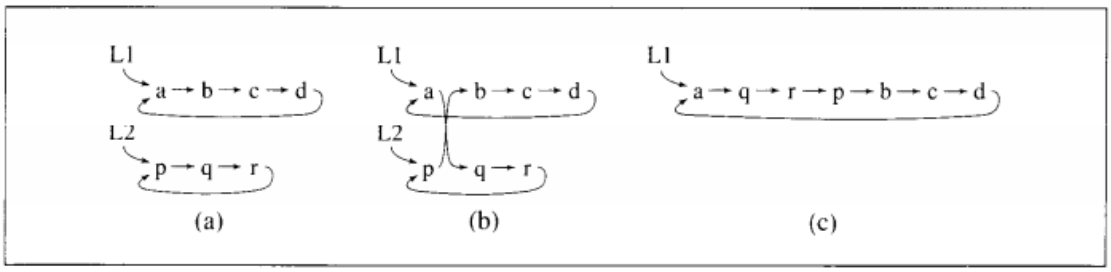
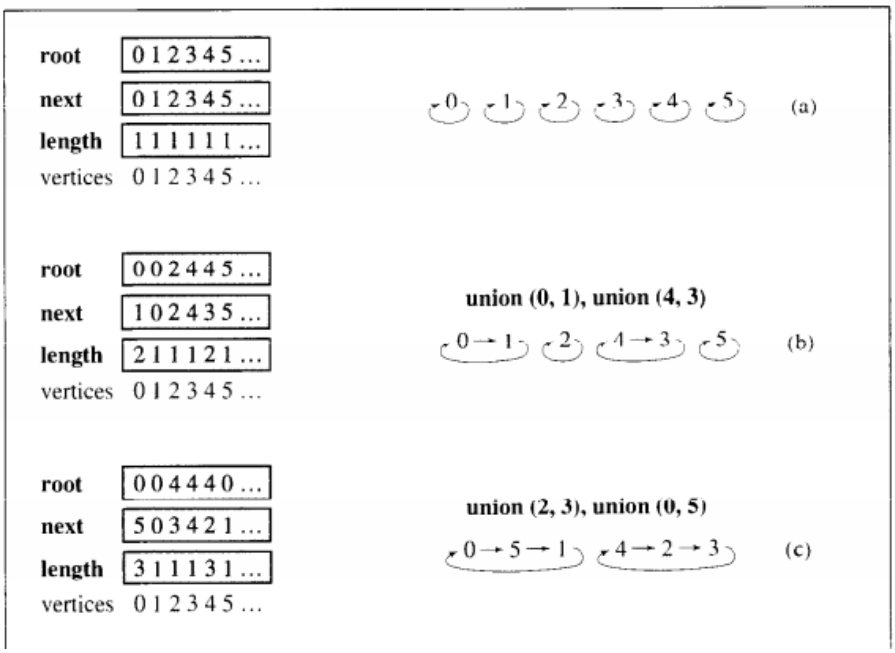


FIGURE 8.13 An example of application of `union()` to merge lists.



An example of the application of `union()` to merge lists is shown in Figure 8.13. After initialization, there are $|V|$ unary sets or one-node linked lists, as in Figure 8.13a. After executing `union()` several times, smaller linked lists are merged into larger ones, and each time, the new situation is reflected in the three arrays, as shown in Figures 8.13b–c.

The complexity of `union()` depends on the number of vertices that have to be updated when merging two lists, specifically, on the number of vertices on the shorter list, since this number determines how many times the `for` loop in `union()` iterates. Since this number can be between 1 and $|V|/2$, the complexity of `union()` is given by $O(|V|)$.

Posible implementación de Kruskal considerando las funciones Union y Find

```
Kruskal
g, MST : Grafo (n);   //n es la cantidad de vértices del grafo
Q : Queue (n);     // cola de prioridades que almacena las aristas en orden creciente de peso
uf : Tipo_UnionFind;
u, v : Tipo_Vertice;
e : Tipo_Arista;

Inicializar MST

Para cada i en Vertices hacer     //Vertices es la lista de vertices del grafo
{
  v := Vertices(i);
 Agregar_vértice v al grafo MST // Los vértices inicalmente están   desconectados
                                  // Las aristas se incorporarán cuando avanza el algoritmo

Make_Set (UF, v);                  // Inicializar estructura Union Find para v
}
```

```
Inicializar Cola Q con los arcos ordenados por peso;

Mientras ( condición)
      {
        e= Desacolar()              // Sacamos un arco con el menor peso
        si  Find (origen_ d_ e(e)) != Find (extremo_de ( e ))
            {
           Agregar_arista e al MST;
           Union (origen_de (e), extremo_de (e));
           }
      }
end kruskal;
```

Ejercicio

b) Completar el algoritmo, analizarlo y determinar el coste temporal del mismo.