ALGORITMOS VORACES

Algoritmo Prim usando montículos.

Iván López de Munain Quintana, 72893594Q

Introducción y problemática

Los algoritmos voraces se caracterizan por aplicar un enfoque miope, es decir, toman decisiones basándose en la información que tienen disponible de modo inmediato, sin tener en cuenta sus posibles efectos en un futuro. Este tipo de algoritmo se suele usar para resolver problemas de optimización como por ejemplo la búsqueda del camino mínimo (algoritmo de Dijkstra) o encontrar un árbol de recubrimiento mínimo en un grafo conexo (algoritmo de Prim o Kruskal).

El funcionamiento de los algoritmos voraces es bastante sencillo. Se parte de en un grupo de candidatos de los cuales se van formando dos conjuntos: los seleccionados y los rechazados. Desde aquí se busca formar una solución a partir de un subconjunto de candidatos utilizando una función de selección, ésta indica cuál de los candidatos restantes (que no han sido rechazados ni seleccionados) es el más prometedor. Disponemos de otra función (factible) que nos sirve para saber si un elemento seleccionado participa o imposibilita una función objetivo (valor de la solución a optimizar).

La mayor característica de estos algoritmos es que avanzan paso a paso, independientemente del futuro, una vez que se ha seleccionado o rechazado un candidato no se puede excluir de la solución o volver a considerarlo.

La problemática de encontrar un árbol de recubrimiento mínimo abarca cuestiones reales como construir un tendido telefónico que conecte diversas ciudades o construir centrales eléctricas que abastezcan a un conjunto de poblaciones de tal forma que se minimice el coste.

Algoritmo de Prim

Sea G = {N, A} un grafo conexo y no dirigido donde N es el conjunto de nodos y A el conjunto de aristas a las cuales se asocian una longitud no negativa, el problema consiste en hallar un subconjunto T de las aristas, tal que usando solo estas, todos los nodos deben quedar conectados sin la existencia de ciclos. Además la suma de las longitudes debe ser lo más pequeña posible.

Para conseguir este objetivo, el algoritmo de Prim escoge un nodo de forma arbitraria al cual vamos a denominar nodo raíz. A partir de este nodo el árbol de recubrimiento mínimo crece de forma natural añadiendo una nueva rama al árbol ya construido. El algoritmo se detendrá una vez se hayan alcanzado todos los nodos.

Sea B un conjunto de nodos y T un conjunto de aristas, B inicialmente está compuesto solo por el nodo raíz y T se encuentra vacío. En cada etapa, el algoritmo busca encontrar la arista de menor longitud *{x,z}* tal que *x* pertenezca a B y z pertenezca a N\B (nodos del conjunto N menos nodos del conjunto B). Seguido a esto añadimos el nodo *z* a B y la arista *{x,z}* a T. De esta forma nos aseguramos en todo momento que las aristas de T forman un árbol de recubrimiento mínimo para los nodos de B.

Pseudocódigo

```
función Prim(G = ⟨N, A⟩: grafo; longitud A→ℝ⁺): conjunto de aristas.
    {Iniciación}
    T ← ∅
    B ← {un miembro arbitrario de N}
    mientras B ≠ N hacer
        buscar e = {u, v} de longitud mínima tal que
            u ∈ B y v ∈ N \ B
        T ← T ∪ {e}
        B ← B ∪ {v}
    devolver T
```

Para obtener una implementación sencilla en una computadora supongamos que nuestro grafo se define mediante una matriz simétrica en la que se representan las distancias entre los nodos L[i,j], siendo dicha distancia igual a infinito en caso de no existir la arista correspondiente. Para todo nodo *i* que no haya sido aún incluido en B, *masProximo[i]* nos dará el nodo de B que está más próximo a *i,* y *distmin[i]* proporcionará la distancia desde *i* hasta el nodo más próximo. Si *i* ya se encontrara en B entonces *distmin[i]=-1*.

```
función Prim(L[1..n, 1..n]): conjunto de aristas
    {Inicialización: sólo el nodo 1 se encuentra en B}
    T ← ∅ {contendrá las aristas del árbol de recubrimiento mínimo}
    para i ← 2 hasta n hacer
        más próximo[i] ← 1
        distmin[i] ← L[i, 1]
    {bucle voraz}
    repetir n – 1 veces
        mín ← ∞
        para j ← 2 hasta n hacer
            si 0 ≤ distmin[j] < mín entonces mín ← distmin[j]
                                             k ← j
        T ← T ∪ {{más próximo[k], k}}
        distmin[k] ← –1 {se añade k a B}
        para j ← 2 hasta n hacer
            si L[j, k] < distmin[j] entonces distmin[k] ← L[j, k]
                                             más próximo[j] ← k
        devolver T
```

Según este esquema el bucle principal del algoritmo se ejecutaría n-1 veces puesto que hay n nodos. Por cada pasada, el bucle anidado tendría un coste de O(n). Esto significaría que el algoritmo Prim sin modificaciones necesitaría un tiempo de $O(n^2)$.

Este coste es bastante elevado comparado con el algoritmo de Kruskal que es logarítmico. Es por eso que vamos a introducir una estructura de ordenación, los montículos (heap), para mejorar dicho coste. Para este caso se va a realizar un estudio del coste más detallado, pero antes de nada vamos a recordar qué es un montículo.

Montículos

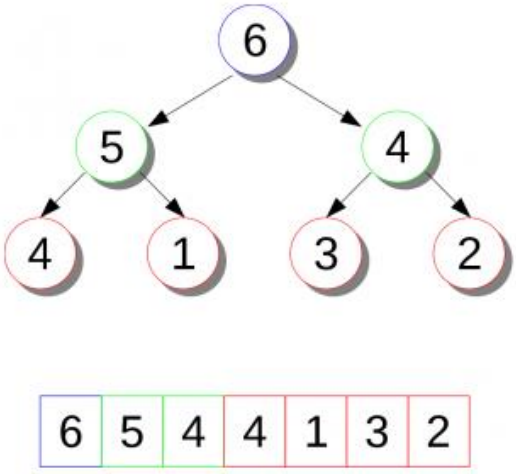
Un montículo es un árbol binario esencialmente completo que tiene la propiedad de que ningún padre tiene un hijo mayor (montículo de máximos) o menor (montículo de mínimos) a él. En nuestro caso nos interesa que sea un montículo de mínimos puesto que buscamos las aristas de menor longitud. Se trata de una estructura de datos que pese a ser representada mediante un árbol se implementa sobre un vector. Al ser binario, un nodo solo puede tener dos hijos a los que se accederá mediante la fórmula (siendo i el índice del padre):

*Hijo_izq=2*i*

*Hijo_dch=2*i+2*

El acceso al padre se realizará con una división entera (i//2).

El uso de montículos nos permite ordenar los elementos de un vector de forma sencilla y rápida, en nuestro caso ordenar las distancias de las aristas del grafo. En suma, nos facilita la obtención del elemento mínimo.



Ejemplo de montículo de máximos.

Pseudocódigo del algoritmo Prim usando montículos

Siendo G={N,A} un grafo de N nodos y un conjunto de aristas A, la matriz *w[i,j]* representa las distancias entre el nodo i y el nodo j. El vector *mstConstruido[v]=u* representa la arista escogida para realizar el MST (Minimum Spanning Tree) que une el nodo *u* con el *v* y el vector *key[i]* es el atributo asociado al nodo *i* que sirve para ordenar el montículo. Por último *filtrarArriba(v,key[v])* reorganiza el árbol binario a partir del nodo *v* habiendo actualizado su valor clave con *key[v]*.

```
Función PrimMonticulos (Grafo,w,nodoRaiz)
        {Inicializaciones}
1       For each i in Grafo.vertices
2               key[i]=∞
3               mstConstruido[i]=-1
4       key[nodoRaiz]=0
5       Montículo=Grafo.vertices
6       while Montículo≠vacio
7               u=extracción-minimo(montículo)
8               for each v in Grafo.adyacentes[u]
9                       if estaEnMonticulo(v) and w[u,v]<key[v]
10                              mstConstruido[v]=u
11                              key[v]=w[u,v]
12                              filtrarArriba(v,key[v])
```

Análisis de coste:

Las líneas 1-5 se utilizan para inicializar las distintas estructuras necesarias en el algoritmo. En esta parte el coste necesario es O(N) puesto que el primer *for* se repite tantas veces como nodos haya para inicializar tanto los valores de *key[i]* como de *mstConstruido[i].* Estas asignaciones tienen un orden de O(1) las cuales no nos interesan puesto que habrá mayores costes posteriormente. Todos los elementos de *key[]* se inicializan con valores altos salvo el nodo raíz para que este sea el primero en ser introducido en el MST. Por consiguiente, *mstConstruido[i]* es inicializado como -1 porque el algoritmo comienza con el árbol de recubrimiento mínimo vacío. En la línea 5 se inicializa el montículo que contiene todos los vértices, esto consiste en insertar N tuplas [nodo *v*, *key[v]]* por lo que tendrá un coste de O(N).

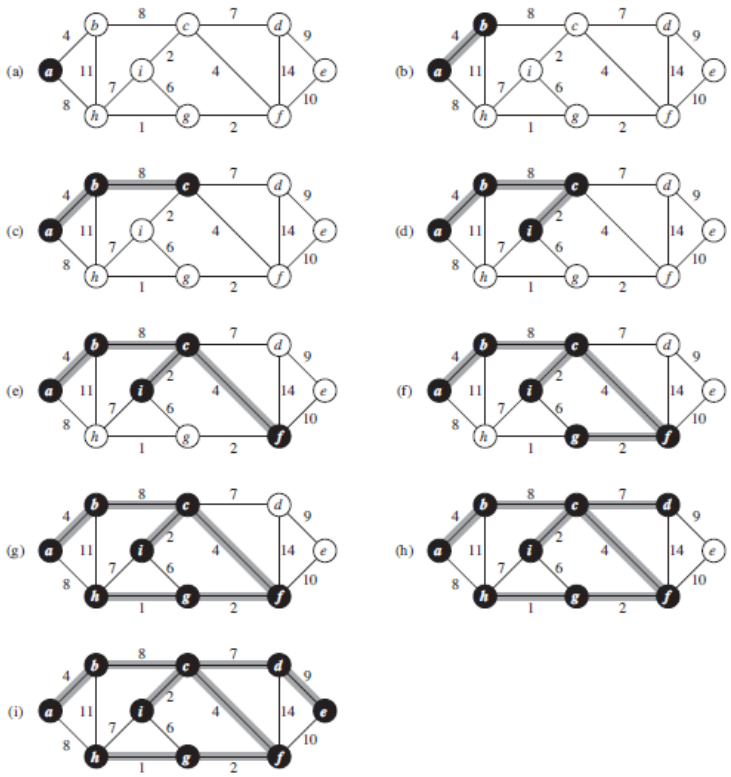
El bucle de las líneas 6-12 se ejecuta |N| veces puesto que el tamaño del montículo es igual al número de nodos y en cada iteración se selecciona un vértice para el MST el cual será eliminado del montículo. Dentro del bucle, cada operación en la que extraemos el mínimo

tiene un coste constante pero restaurar la propiedad de montículo cuesta O(logN) ya que la altura en un árbol binario de N nodos es igual a logN, por lo que el coste total necesario para realizar todas las llamadas a *extracción-minimo* es de O(NlogN).

El *for* de las líneas 8-12 se ejecuta 2*|A| puesto que cada arista es revisada dos veces, una vez por cada nodo que conecta. Por lo que este bucle tiene un coste de O(A). Podemos implementar la comprobación de si el nodo pertenece al montículo (línea 9) con un tiempo constante mediante un bit para cada vértice que indique si se encuentra en el montículo o no, actualizándolo en caso de ser eliminado de éste. Por último la asignación de la línea 11 implica una reorganización en el montículo de mínimos lo que supone un coste de O(logN).

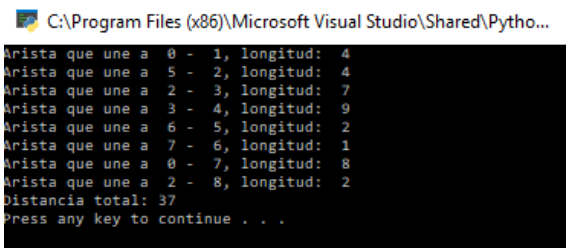
Así por consiguiente, el tiempo total necesitado para el algoritmo de Prim usando montículos es O(NlogN + AlogN) = O(AlogN). Siendo ya este coste asintóticamente igual que el de Kruskall.

Ejemplo gráfico:



Ejemplo de aplicación del algoritmo Prim a un grafo.

El resultado obtenido con mi programa sería el siguiente (vemos que difiere en que coge la arista de a-h en vez de b-c puesto que tienen la misma distancia):



Código:

Disponemos de dos clases, la clase Montículo y Prim.

En la clase Montículo el constructor consiste en un array en donde almacenaremos tuplas [*nodo v, key[v]*], otro atributo que nos indica el tamaño del montículo y por último un array llamado *posición[v]* que nos indica la localización del nodo v en el árbol binario. En esta clase hemos implementado diferentes métodos cuyas funcionalidades son explicadas a continuación:

-intercambiarNodos(a,b):

Intercambia dos nodos en el montículo. Esto es usado a la hora de reordenar el árbol binario una vez que se ha escogido un nodo para añadirlo al MST y es eliminado del montículo.

```
#intercambiar dos nodos en el montículo
def intercambioNodos(self, a, b):
    t = self.array[a]
    self.array[a] = self.array[b]
    self.array[b] = t
```

-constrMonticulo(índice):

Dado un índice se comprueba que se cumpla la propiedad de que los hijos no son menores que el padre, en caso de que no se cumpliera la propiedad se intercambiarían los nodos usando el método *intercambioNodos(padre,hijo)* y además se actualizaría el vector posición puesto que la localización de dichos nodos ha cambiado. Por último este método hace una llamada recursiva a sí mismo por si fuese necesario seguir ordenando el resto del montículo.

```
def constrMonticulo(self, indice):
    min = indice
    izq = 2 * indice + 1
    dch = 2 * indice + 2

    if izq < self.tamano and self.array[izq][1] < self.array[min][1]:
        min = izq

    if dch < self.tamano and self.array[dch][1] < self.array[min][1]:
        min = dch

    #si no coincide hay que reorganizar el monticulo
    if min != indice:

        # Intercambio de posiciones
        self.posicion[ self.array[min][0] ] = indice
        self.posicion[ self.array[indice][0] ] = min

        # Intercambio de  nodos
        self.intercambioNodos(min, indice)
        #recursivo hasta comprobar que se cumple la propiedad de monticulo
        self.constrMonticulo(min)
```

-minimo():

Este método nos devuelve el nodo con menor clave asociada, al tratarse de un montículo de mínimos, dicho elemento es el nodo raíz. Una vez seleccionado el nodo lo intercambiamos con el último elemento del árbol y disminuimos en una unidad el tamaño del mismo. Esto es debido a que una vez que sea escogido un nodo para formar el MST no puede volver a ser cogido. Hay que tener en cuenta que a causa de estas acciones será necesario la ordenación del árbol mediante el método *constrMonticulo(0).* Introducimos el índice 0 puesto que el elemento desordenado se encuentra en la raíz, de hecho es una buena forma de comprobar la propiedad de montículo de mínimos en todo el árbol. Recordar que el vector posición necesita ser actualizado para saber en todo momento la localización de los nodos.

```
#extraccion del nodo minimo del monticulo
def minimo(self):

    # guardamos el elemento raiz que tiene la menor distancia
    raiz = self.array[0]

    #lo remplazamos con el ultimo que es el que mayor distancia tiene para que luego se reordene
    ultimoNodo = self.array[self.tamano - 1]
    self.array[0] = ultimoNodo

    #actualizamos las posiciones
    self.posicion[ultimoNodo[0]] = 0
    self.posicion[raiz[0]] = self.tamano - 1

    #reducimos el tamaño del monticulo para saber qué nodos han sido ya incorporados al MST
    self.tamano -= 1
    #reorganizamos el monticulo
    self.constrMonticulo(0)

    return raiz
```

-esVacio():

Función booleana que nos dice si el montículo está vacío o no, devolviéndonos true en caso afirmativo.

```
def esVacio(self):
    return True if self.tamano == 0 else False
```

-filtrarArriba(v,key):

Dado un nodo *v* y su clave *key*, obtenemos su posición dentro del montículo mediante *posición[v]*. Comprobamos si la clave asociada a su padre es menor que la suya y en caso de no cumplirse se intercambiarían las nodos mediante *intercambioNodos(padre,hijo)* y actualizaríamos el valor de las posiciones.

```
def filtrarArriba(self, v, key):

    i = self.posicion[v]
    # actualizar el valor de la distancia
    self.array[i][1] = key

    while i > 0 and self.array[i][1] < self.array[(i - 1) // 2][1]:
        print(self.posicion)
        print(i)
        print(self.array[i][1] )
        print(self.array[(i - 1) // 2][1])
        # intercambiar posiciones del nodo con su padre
        self.posicion[ self.array[i][0] ] = (i-1)//2
        self.posicion[ self.array[(i-1)//2][0] ] = i
        self.intercambioNodos(i, (i - 1)//2 )
        i = (i - 1) // 2 #movemos el indice al padre
        print(self.posicion)
```

-estaMonticulo(v):

Función booleana que nos sirve para saber si un nodo sigue estando en el montículo o ha sido ya incorporado al MST. Recordar que en métodos anteriores (*minimo())* una vez había sido seleccionado un nodo se le asignaba una posición igual a la longitud menos uno y se decrementaba en una unidad dicha longitud. De tal forma que podemos determinar si un nodo sigue estando disponible o no mediante su valor *posición[v]*.

```
#ver si el nodo v aun no ha sido seleccionado
def estaMonticulo(self, v):

    if self.posicion[v] < self.tamano:
        #se encuentra en el monticulo
        return True
    #no se encuentra en el monticulo
    return False
```

-printArr (mstConstruido, n, dist):

Este método nos sirve para imprimir los resultados del algoritmo Prim. El vector *mstConstruido* representa las aristas que han sido seleccionadas para construir el árbol de recubrimiento mínimo. Es decir, *mstConstruido[v]=u* representa la arista que une al nodo *u* con el nodo *v*. El argumento *n* indica el número total de vértices del grafo y por último el array *dist[v]* muestra la distancia de la arista de *v* a *u*.

```
def printArr(mstConstruido, n,key):
    z=0
    for i in range(1, n):
        print( "Arista que une a % d - % d, longitud: % d" % (mstConstruido[i], i,key[i]) )
        z+=key[i]
    print("Distancia total:", z)
```

En la clase Prim el constructor consta de un atributo N que define el número de nodos, y un diccionario en el que la clave representa el nodo origen y el valor asociado es una lista de tuplas [nodo destino, distancia]. En esta clase los métodos definidos son los siguientes:

-ponerNodo(origen,destino,dist):

Añadir los nodos y las distancias de las aristas al diccionario. Cabe destacar que a la hora de introducir los datos el nodo que se elija como raíz tiene que ser el elemento de índice cero.

```
def ponerNodo(self, origen, destino, dist):

    #insert(index,elemento), se inserta en el indice cero asi no es
    #necesario conocer la longitud final, el resto de elementos se desplazarian sumando uno en el indice
    self.graph[origen].insert(0, [destino, dist]) #origen es la clave y a ella esta asociada [destino,distancia]
    #grafo no dirigido, por eso se introduce una segunda vez
    self.graph[destino].insert(0, [origen, dist])
```

-primMST():

Aplicación del algoritmo Prim a un grafo.

```
def primMST(self):

    #obtener el numero de nodos
    N = self.N
    # clave asociada a cada nodo para ordenar el monticulo
    key = []
    #mstConstruido sirve para almacenar el MST construido, representa las aristas que conectan el nodo i con mstConstruido[i]
    mstConstruido = []
    #creacion del monticulo
    monticulo = Monticulo()

    #Inicializacion del monticulo con todos los vertices.
    #Valor de las claves igual a infinito
    for v in range(N):
        mstConstruido.append(-1)
        key.append(9999999)
        monticulo.array.append( [v, key[v]] )
        monticulo.posicion.append(v)

    #para extraer primero el nodo raíz escogido
    key[0] = 0
    monticulo.tamano = N;
```

```
#Mientras haya nodos en el monticulo seguimos
while monticulo.esVacio() == False:
    #obtencion del nodo con menor distancia
    nodoMin = monticulo.minimo() #nodoMin=[v,key[v]]
    u = nodoMin[0] #u = nodo v

    #recorrer todos los vértices adyacentes al nodo u viendo sus distancias
    #recordar que graph se trata de un diccionario en el que la clave es un nodo origen y su valor una lista [nodo destino, distancia] -> [recorrido[0],recorrido[1]]
    for recorrido in self.graph[u]:

        v = recorrido[0] #obtenemos el nodo destino

        #comprobar si el nodo v se encuentra aun en el monticulo y si la distancia del nodo u a v es menor que la clave almacenada en key[v]
        #(recordar que las hemos inicializado a un valor muy alto)
        if monticulo.estaMonticulo(v) and recorrido[1] < key[v]:
            key[v] = recorrido[1]  #almacenamos la longitud de la arista que une a los nodos v y u
            mstConstruido[v] = u

            # actualizacion del monticulo
            monticulo.filtrarArriba(v, key[v])

printArr(mstConstruido, N,key)
```

Referencias:

[1] https://altenwald.org/2012/02/12/monticulos/

[2]http://interactivepython.org/runestone/static/pythoned/Trees/ImplementacionDeUnMonticuloBinario.html

[3] https://es.wikipedia.org/wiki/Algoritmo_de_Prim

[4] Fundamentos de algoritmia - G. Brassard & P. Bratley.

[5] Introduction to Algorithms- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein.