Ficha 08

Árbol de Expansión Mínimo: El Algoritmo de Prim

1.) Introducción: Árboles de Expansión Mínimos para un Grafo.

En general, se define como un árbol (o más general aún, un *árbol libre*) a un grafo que no tiene ciclos pero tiene conectados a todos sus vértices, y un rápido examen de esta definición muestra que entonces un árbol es un grafo que tiene la mínima cantidad de arcos posibles para manterlo conexo.

Muchas veces un grafo ponderado conexo y no dirigido es denso o incluso completo y en algunas situaciones podría ser necesario reducir el grafo a su "minima expresión" en cuanto a cantidad de arcos. Podría ser necesario, por ejemplo, crear otro grafo que sea un árbol con los mismos nodos que el grafo original, pero sólo los arcos del original que mínimamente sean necesarios para que el grafo siga siendo conexo. Ese segundo grafo comúnmente se designa como *árbol de expansión* del grafo original, o también *árbol recubridor* o *árbol abarcador*. Note que un grafo podría tener varios árboles de expansión distintos y válidos.

Si el árbol de expansión es tal que la suma de los pesos de sus arcos es la mínima posible, entonces el árbol de expansión se dice *árbol de expansión mínimo*. Note que un grafo podría tener varios árboles de expansión mínimos y válidos. La Figura 1 de la página siguiente muestra un grafo ponderado, conexo y no dirigido, junto a un árbol de expansión posible y un árbol de expansión mínimo.

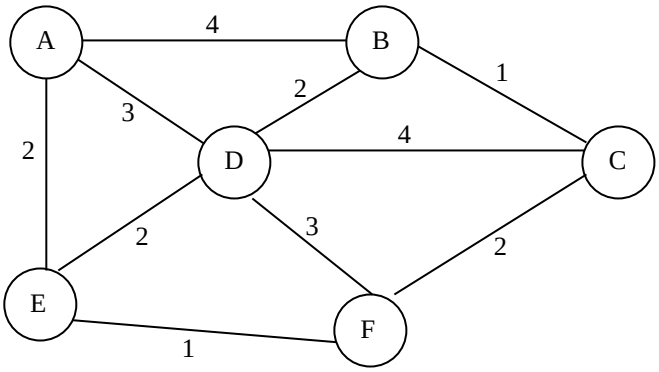
Un algoritmo que calcula un árbol de expansión mínimo para un grafo puede basarse en una idea o propiedad intuitivamente simple: si G1 y G2 son dos subconjuntos de nodos del grafo original, entonces el arco con menor peso del grafo original que permite unir algún nodo de G1 con algún nodo de G2, pertenece a un árbol de expansión mínimo. Esta propiedad es demostrable (se incluye esa demostración en la sección 2 de esta ficha), y se puede aplicar iterativamente para ir armando arco por arco el árbol de expansión mínimo.

El algoritmo resultante fue ideado por *Robert Prim* y se conoce por ello como *Algoritmo de Prim*. Se comienza haciendo que el subconjunto G1 contenga sólo a un nodo *x* del grafo, y se toma a G2 como conteniendo al resto del grafo. Luego se toma el arco con menor peso que permite unir a *x* con algún nodo *y* de G2. Ese arco pertenece al árbol de expansión mínimo (que se va construyendo dentro de G1). Por lo tanto, a partir de ahora se asume que G1 contiene a los nodos *x* e *y*, unidos por ese arco. Se aplica nuevamente el mecanismo, pero ahora se busca el arco con menor peso que permita unir a *x* o a *y* con algún otro nodo en G2. Y así se continúa hasta que todos los nodos de G2 pasaron a G1. El conjunto G1 finalmente armado, es un árbol de expansión mínimo.

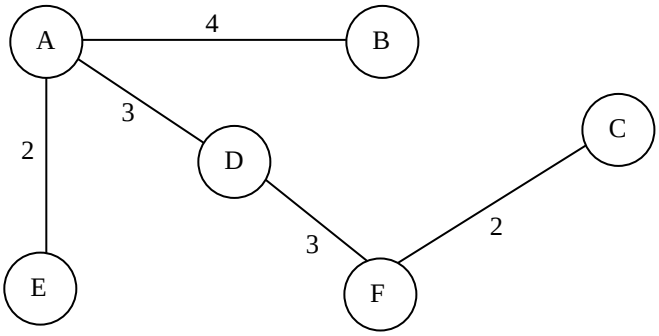
Si en la implementación del algoritmo se aplica alguna técnica veloz (de orden logarítmico) para encontrar el siguiente arco de menor peso entre todos los que restan (por ejemplo, usando un *heap* que es lo que finalmente hace el *Algoritmo de Prim*), entonces el algoritmo tiene tiempo de ejecución *O(m*log(n))*, siendo *m* el número de arcos del grafo, y *n* el número de vértices.

En el modelo *DLC-Prim* que acompaña a esta ficha, la clase *UndirectedGraph* incluye el método *getMSTValue_Prim()* que aplica estas ideas y calcula un árbol de expansión mínimo. El método simplemente retorna el valor de la suma de pesos del árbol de expansión mínimo obtenido. Dejamos el análisis del código fuente (y el planteo de las variaciones que pudiesen necesitar) para los estudiantes.

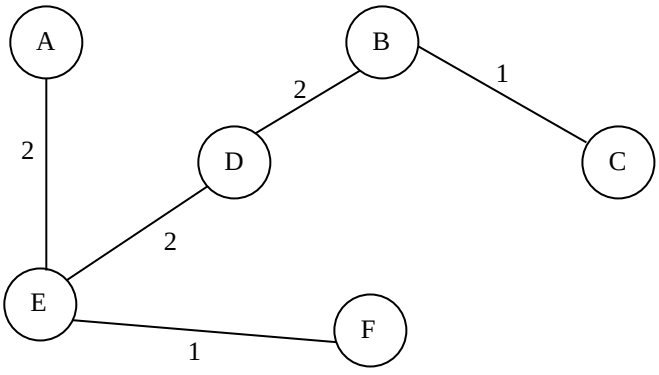
Figura 1: un grafo con algunos árboles de expansión



Grafo original: ponderado, conexo y no dirigido



Un árbol de expansión válido para el grafo original (suma de pesos: 14)



Ùn árbol de expansión mínimo para el grafo original (suma de pesos: 8) (mínima)

2.) Algoritmos Ávidos: Árbol de Expansión Mínimo - Algoritmo de Prim (Prueba de Validez).

Por lo pronto, observemos que el *algoritmo de Prim* tal como fue descripto en la sección anterior es un *algoritmo ávido*: se basa en la repetición sistemática de una regla local, con la esperanza de llegar finalmente a un resultado global correcto. En el caso del algoritmo de Prim, la regla ávida es la *propiedad del arco de cruce de peso mínimo*, que indica que si se tienen dos subgrafos *G1* y *G2* de un grafo *G* conexo y ponderado, entonces el *arco de cruce con menor peso* (esto es, el arco con un vértice en *G1* y el

otro en *G2* que tenga el menor peso) pertenece a un *AEM* para *G*. El algoritmo completo consiste en partir de un subconjunto *G1* que contenga a sólo un vértice *x* de *G*, tomar el arco de cruce con menor peso entre los que vinculan a *x* con el resto de los vértices, pasar el segundo vértice de este arco a *G1* y repetir la regla, una y otra vez, hasta que *G1* contenga a todos los vértices de *G*.

Sin embargo, hemos visto también que cuando se trata de algoritmos ávidos es estrictamente necesario demostrar que la aplicación de la regla ávida local efectivamente permitirá llegar siempre a un resultado final correcto. En el caso del algoritmo de Prim, la demostración de su validez es relativamente simple[1]:

- ✓ **Prueba de Validez del Algoritmo de Prim**: queremos probar que el *Algoritmo de Prim* siempre obtiene un *AEM* para el grafo *G = {V, A}* de partida, sabiendo que *G* es no dirigido, conexo y ponderado, con *V* = el conjunto de *n* vértices de *G* y *A* = el conjunto de *m* arcos de *G*.

  Demostración:

  Dados dos subgrafos *G1* y *G2* de *G*, en cada iteración del algoritmo de Prim se debe encontrar un arco de cruce *a(u, v)* que permita conectar un vértice *u* del subgrafo *G1* con otro vértice *v* en el subgrafo *G2*. Estamos seguros que podremos encontrar al menos un arco de cruce *a(u, v)* debido a que *G es conexo* y esto implica que siempre existirá un camino que conecte a *u* con *v*.

  Por lo pronto, podemos probar rápidamente que la salida *T* del algoritmo de Prim es un *árbol de expansión para G* (esto es: un grafo que contiene a todos los vértices de *G*, sin ciclos y con todos sus vértices conectados, aunque aún no imponemos que sea *mínimo*) debido a los siguientes hechos:

  a.) Todos los vértices de *T* están conectados, ya que por definición los arcos y los vértices que se van agregando a *T* en cada pasada están conectados en *G*, y *T* contiene a todos los vértices de *G* ya que el propio proceso de construcción sólo finaliza cuando *T* llega a contener a todos los vértices de *G*.

  b.) El agregado de un arco *a(u, v)* en *T* nunca produce un ciclo: en efecto, si *T* contiene a los vértices que ya se tomaron para el *AEM* parcial, entonces el conjunto *X = {V - T}* contiene a los vértices restantes que aún no entraron al *AEM* parcial. Suponga que se toma el arco de cruce *a(u, v)* para agregarlo al *AEM* parcial, con *u ε T* y *v ε X*. Entonces *a(u, v)* es el primer arco de cruce encontrado que une a *u* con *v*, pues de otro modo se habría encontrado antes a otro arco *a'(u,v)* y *v* ya estaría en *T*. Como *a(u,v)* es el primero que une a *u* con *v*, al agregarlo al *AEM* parcial estamos seguros que no había otro arco *a'(u,v)* en el *AEM* parcial y por lo tanto no se produce un ciclo.

  Ahora intentemos probar que el árbol de expansión *T* producido por el algoritmo de Prim *es mínimo en cuanto a la suma de sus pesos*. Sea *T1* un árbol de expansión mínimo (*AEM*) para *G*. Pueden ocurrir dos situaciones:

  1.) Si *T = T1* entonces *T* es un árbol de expansión mínimo para *G*, ya que *T1* lo es.

  2.) Si *T != T1*, sea *e(u, v)* el primer arco agregado durante la construcción de *T* , tal que *e(u, v)* no está en *T1* y sea *P* el conjunto de vértices conectados por los arcos ya agregados al *AEM* parcial antes que *e*. Entonces un vértice de *e* está en *P* y el otro no. Ya que *T1* es un *AEM* de *G*, entonces hay un camino en *T1* que une esos dos vértices. Si nos desplazamos por ese camino se debe encontrar un arco *f* que une a un vértice en *P* con uno que no está en *P*. Entonces en la iteración en que *e* se

[1] La fuente en la que se inspira la demostración que sigue, está tomada de *Wikipedia* (de la que recomendamos la versión en inglés): http://en.wikipedia.org/wiki/Prim%27s_algorithm. Asimismo, la idea usada para probar que el agregado de un arco al AEM parcial nunca produce un ciclo, está tomada del material del curso "*Design and Analysis of Algorithms II*" – Stanford University (*Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses.

agrega a *P*, también se podría haber agregado *f*, y se hubiese agregado en vez de *e* si el peso de *f* [denotado como *w(f)*] fuera menor que peso el de *e* [denotado como *w(e)*]. Ya que *f* no se agregó se concluye que *w(f) >= w(e)*. Si ahora consideramos *T2* como el grafo que se obtiene al remover *f* de *T1* y agregar *e*, podemos ver que *T2* sigue siendo conexo, tiene la misma cantidad de arcos que *T1*, y la suma total de los pesos de sus arcos no es mayor que la suma de pesos de *T1*. Por lo tanto *T2* es también un *AEM* para G.

Esto quiere decir que la salida *T* obtenida por la aplicación repetida del proceso local de tomar cada vez el arco de cruce de menor peso, siempre será finalmente un *AEM* para *G* y se demuestra así la validez del algoritmo de Prim.

3.) Algoritmo de Prim: Detalles de Implementación[2].

Si el grafo se representa en forma de listas de adyacencia, entonces los *m* arcos quedan disponibles en una lista ligada, sin usar espacio extra para arcos que no existen, y por lo tanto, si se utiliza un heap para encontrar el arco de menor peso en tiempo logarítmico, el tiempo de ejecución para el peor caso puede reducirse a

$$O(m * \log(m)) = O(m * \log(n^2)) = O(m * 2 \log(n)) = \mathbf{O(m * \log(n))}$$

El uso de un heap para ir recuperando en forma rápida el siguiente arco de menor peso hace más veloz el proceso, al costo de utilizar espacio extra para el propio heap. La idea es comenzar con un subconjunto *G1* del grafo *G*, tal que *G1* contenga sólo a uno de los vértices del grafo, y otro subconjunto *G2* que contenga al resto de los vértices. Luego tomar el arco *a* de cruce entre *G1* y *G2* que tenga el menor peso, y pasar a *G1* el vértice de *a* que estuviese en *G2*. Continuar aplicando esa regla ávida hasta que *G1* contenga a todos los vértices del grafo original.

Un esquema en pseudocódigo para un algoritmo que calcule un *AEM* y retorne la suma de sus pesos, sería:

Entrada: un grafo no dirigido, ponderado, conexo y denso *G = {V, A}* donde *V* es el conjunto de *n* vértices y *A* es el conjunto de *m* arcos *a = (x, y, w)* (tales que *x, y ε V* y *w* es el peso del arco *a*).

Salida: la suma de pesos *s* de los arcos de un árbol de expansión mínimo (*AEM*) para *G*.

Algoritmo esencial:

- Sea una lista de vértices *G1* con un vértice cualquiera *u* de *V*: *G1 = [u]*.
- Sea *s = 0*.
- Sea una lista de arcos *T* vacía, para los arcos del AEM parcial: *T = []*.
- Mientras *G1* != *V*:
  - Tome el arco *a(u,v,w)* **de menor peso *w***, con (*u* ∈ G1) y (v ∉ G1).
  - Agregue el arco *a(u,v,w)* a la lista *T*.
  - Sume *w* a *s*.
  - Agregue el vértice *v* al conjunto *G1*.
- Retorne la suma de pesos *s* de los arcos en *T*.

[2] La fuente en la que está basado el algoritmo que sigue, es el material del curso "*Design and Analysis of Algorithms II*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses.

Está claro que el paso crítico en este algoritmo esencial, es la **recuperación del arco *a(u,v,w)* con menor peso** entre todos los arcos de cruce. Si la búsqueda del arco de menor peso se hace en forma ingenua, comparando contra todos los arcos por fuerza bruta, entonces el algoritmo tendrá un tiempo de ejecución *O(m * n)*. Si el grafo no es muy grande un algoritmo de fuerza bruta podría obtener el resultado buscado en un tiempo aceptable. Pero como se sugirió, una mejor implementación consiste en usar un *heap* para almacenar los arcos de cruce, de forma que ese *heap* se ordene de acuerdo a los pesos de los arcos y obtener así el arco de menor peso en tiempo logarítmico en cada iteración del ciclo. Usando correctamente un *heap*, el tiempo de ejecución total del *algoritmo de Prim* baja (como vimos) a *O(m * log(n))* [aunque al costo de usar memoria extra para el heap].

Observe que dado que los pesos de los arcos pueden ser negativos, entonces la suma de pesos del AEM podría ser cualquier número entero *negativo*, *cero* o *positivo*. La implementación de este algoritmo viene incluida en el proyecto *DLC-Prim* que acompaña a esta ficha (y en rigor es el mismo modelo que acompañaba a la ficha en la que se presentó la implementación de grafos mediante listas de adyacencia, pero agregando en la clase *UndirectedGraph* el método que implementa el *Algoritmo de Prim*).