Algoritmo de Prim

El **algoritmo de Prim** es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, **no** dirigido y cuyas aristas están etiquetadas.

En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

El algoritmo fue diseñado en 1930 por el matemático Vojtech Jarnik y luego de manera independiente por el científico computacional Robert C. Prim en 1957 y redescubierto por Dijkstra en 1959. Por esta razón, el algoritmo es también conocido como **algoritmo DJP** o **algoritmo de Jarnik**.

Descripción conceptual

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar.

Pseudocódigo del algoritmo

- Estructura de datos auxiliar: Cola = Estructura de datos **Cola de prioridad** (se puede implementar con un heap)

```
Prim (Grafo G)
    // Inicializamos todos los nodos del grafo. La distancia la ponemos a infinito y el padre de cada nodo a NULL
    // Encolamos, en una cola de prioridad donde la prioridad es la distancia, todas las parejas <nodo,distancia> del grafo
    por cada u en V[G] hacer
        distancia[u] = INFINITO
        padre[u] = NULL
        Añadir(cola,<u,distancia[u]>)
    distancia[u]=0
    mientras !esta_vacia(cola) hacer
        // OJO: Se entiende por mayor prioridad aquel nodo cuya distancia[u] es menor.
        u = extraer_minimo(cola) //devuelve el minimo y lo elimina de la cola.
        por cada v adyacente a 'u' hacer
            si ((v ∈ cola) && (distancia[v] > peso(u, v)) entonces
                padre[v] = u
                distancia[v] = peso(u, v)
                Actualizar(cola,<v,distancia[v]>)
```

Código en C++

```cpp
// Declaraciones en el archivo .h
const int INF = -1;
int cn; //cantidad de nodos
vector< vector<int> > ady; //matriz de adyacencia

// Devuelve la matriz de adyacencia del arbol minimo.
```

```
vector< vector<int> > Grafo :: prim(){
    // uso una copia de ady porque necesito eliminar columnas
    vector< vector<int> > adyacencia = this->ady;
    vector< vector<int> > arbol(cn);
    vector<int> markedLines;
    vector<int> :: iterator itVec;

    // Inicializo las distancias del arbol en INF.
    for(int i = 0; i < cn; i++)
        arbol[i] = vector<int> (cn, INF);

    int padre = 0;
    int hijo = 0;
    while(markedLines.size() + 1 < cn){
        padre = hijo;
        // Marco la fila y elimino la columna del nodo padre.
        markedLines.push_back(padre);
        for(int i = 0; i < cn; i++)
            adyacencia[i][padre] = INF;

        // Encuentro la menor distancia entre las filas marcadas.
        // El nodo padre es la linea marcada y el nodo hijo es la
columna del minimo.
        int min = INF;
        for(itVec = markedLines.begin(); itVec != markedLines.end();
itVec++)
            for(int i = 0; i < cn; i++)
                if(min > adyacencia[*itVec][i]){
                    min = adyacencia[*itVec][i];
                    padre = *itVec;
                    hijo = i;
                }

        arbol[padre][hijo] = min;
        arbol[hijo][padre] = min;
    }
    return arbol;
}
```

Funciona perfectamente. Obviamente la variable ady debe estar inicializada, una forma es hacerlo en el constructor; por ejemplo, si la clase se llama Grafo:

```
Grafo :: Grafo(int nodos){
    this->cn = nodos;
    this->ady = vector< vector<int> > (cn);

    for(int i = 0; i < cn; i++)
        ady[i] = vector<int> (cn, INF);
```

```
}
```

Código en JAVA

```
public class Algorithms
{
    public static Graph PrimsAlgorithm (Graph g, int s)
    {
      int n = g.getNumberOfVertices ();
      Entry[] table = new Entry [n];
      for (int v = 0; v < n; ++v)
       table [v] = new Entry ();
      table [s].distance = 0;
      PriorityQueue queue =
       new BinaryHeap (g.getNumberOfEdges());
      queue.enqueue (
       new Association (new Int (0), g.getVertex (s)));
      while (!queue.isEmpty ())
      {
       Association assoc = (Association) queue.dequeueMin();
       Vertex v0 = (Vertex) assoc.getValue ();
       int n0 = v0.getNumber ();
       if (!table [n0].known)
       {
            table [n0].known = true;
            Enumeration p = v0.getEmanatingEdges ();
            while (p.hasMoreElements ())
            {
             Edge edge = (Edge) p.nextElement ();
             Vertex v1 = edge.getMate (v0);
             int n1 = v1.getNumber ();
             Int wt = (Int) edge.getWeight ();
             int d = wt.intValue ();
             if (!table[n1].known && table[n1].distance>d)
             { table [n1].distance = d;
                  table [n1].predecessor = n0;
                  queue.enqueue (
                   new Association (new Int (d), v1));
             }
            }
       }
      }
      Graph result = new GraphAsLists (n);
      for (int v = 0; v < n; ++v)
       result.addVertex (v);
      for (int v = 0; v < n; ++v)
       if (v != s)
            result.addEdge (v, table [v].predecessor);
```

```
        return result;
      }
}
```

Otra versión sin usar Colas

```
    public int[][] AlgPrim(int[][] Matriz) {  //Llega la matriz a la
que le vamos a aplicar el algoritmo
        boolean[] marcados = new boolean[ListaVertices.size()];
//Creamos un vector booleano, para saber cuales están marcados
        String vertice = ListaVertices.get(0); //Le introducimos un
nodo aleatorio, o el primero
        return AlgPrim(Matriz, marcados, vertice, new
int[Matriz.length][Matriz.length]); //Llamamos al método recursivo
mandándole
    }

//un matriz nueva para que en ella nos



//devuelva el árbol final
    private int[][] AlgPrim(int[][] Matriz, boolean[] marcados, String
vertice, int[][] Final) {
        marcados[ListaVertices.indexOf(vertice)] = true;//marcamos el
primer nodo
        int aux = -1;
        if (!TodosMarcados(marcados)) { //Mientras que no todos estén
marcados
            for (int i = 0; i < marcados.length; i++) { //Recorremos sólo las filas de los nodos marcados
                if (marcados[i]) {
                    for (int j = 0; j < Matriz.length; j++) {
                        if (Matriz[i][j] != 0) {         //Si la arista
existe
                            if (!marcados[j]) {          //Si el nodo no
 ha sido marcado antes
                                if (aux == -1) {         //Esto sólo se
hace una vez
                                    aux = Matriz[i][j];
                                } else {
                                    aux = Math.min(aux, Matriz[i][j]);
//Encontramos la arista mínima
                                }
                            }
                        }
                    }
                }
            }
            //Aquí buscamos el nodo correspondiente a esa arista mínima
```

```
 (aux)
            for (int i = 0; i < marcados.length; i++) {
                if (marcados[i]) {
                    for (int j = 0; j < Matriz.length; j++) {
                        if (Matriz[i][j] == aux) {
                            if (!marcados[j]) { //Si no ha sido marcado
 antes
                                Final[i][j] = aux; //Se llena la matriz
 final con el valor
                                Final[j][i] = aux;//Se llena la matriz
final con el valor
                                return AlgPrim(Matriz, marcados,
ListaVertices.get(j), Final); //se llama de nuevo al método con



//el nodo a marcar
                            }
                        }
                    }
                }
            }
        }
        return Final;
    }
    public boolean TodosMarcados(boolean[] vertice) { //Método para
saber si todos están marcados
        for (boolean b : vertice) {
            if (!b) {
                return b;
            }
        }
        return true;
    }
```

Demostración

Sea G un grafo conexo y ponderado.

En toda iteración del algoritmo de Prim, se debe encontrar una arista que conecte un nodo del subgrafo a otro nodo fuera del subgrafo.

Ya que G es conexo, siempre habrá un camino para todo nodo.

La salida Y del algoritmo de Prim es un árbol porque las aristas y los nodos agregados a Y están conectados.

Sea Y el árbol recubridor mínimo de G .

Si $Y_1 = Y \Rightarrow Y$ es el árbol recubridor mínimo.

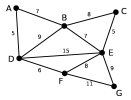
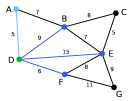
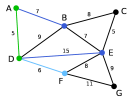
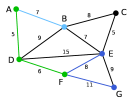
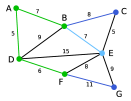
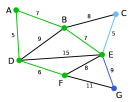
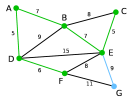
Si no, sea e la primera arista agregada durante la construcción de Y, que no está en Y_1 y sea V el conjunto de nodos conectados por las aristas agregadas antes que e . Entonces un extremo de e está en V y el otro no. Ya que Y_1 es el árbol recubridor mínimo de G hay un camino en Y_1 que une los dos extremos. Mientras que uno se mueve por el camino, se debe encontrar una arista f uniendo un nodo en V a uno que no está en V. En la iteración que

e se agrega a Y, f también se podría haber agregado y se hubiese agregado en vez de e si su peso fuera menor que el de e. Ya que f no se agregó se concluye:

$$P(f) \geq P(e)$$

Sea Y_2el grafo obtenido al remover f y agregando $e \in Y_1$. Es fácil mostrar que Y_2conexo tiene la misma cantidad de aristas que Y_1, y el peso total de sus aristas no es mayor que el de Y_1, entonces también es un árbol recubridor mínimo de G y contiene a e y todas las aristas agregadas anteriormente durante la construcción de V.
Si se repiten los pasos mencionados anteriormente, eventualmente se obtendrá el árbol recubridor mínimo de G que es igual a Y.
Esto demuestra que Yes el árbol recubridor mínimo de G .

Ejemplo de ejecución del algoritmo

Image	Descripción	No visto	En el grafo	En el árbol
	Este es el grafo ponderado de partida. No es un árbol ya que requiere que no haya ciclos y en este grafo los hay. Los números cerca de las aristas indican el peso. Ninguna de las aristas está marcada, y el vértice **D** ha sido elegido arbitrariamente como el punto de partida.	C, G	A, B, E, F	D
	El segundo vértice es el más cercano a **D**: **A** está a 5 de distancia, **B** a 9, **E** a 15 y **F** a 6. De estos, 5 es el valor más pequeño, así que marcamos la arista **DA**.	C, G	B, E, F	A, D
	El próximo vértice a elegir es el más cercano a **D** o **A**. **B** está a 9 de distancia de **D** y a 7 de **A**, **E** está a 15, y **F** está a 6. 6 es el valor más pequeño, así que marcamos el vértice **F** y a la arista **DF**.	C	B, E, G	A, D, F
	El algoritmo continua. El vértice **B**, que está a una distancia de 7 de **A**, es el siguiente marcado. En este punto la arista **DB** es marcada en rojo porque sus dos extremos ya están en el árbol y por lo tanto no podrá ser utilizado.	null	C, E, G	A, D, F, B
	Aquí hay que elegir entre **C**, **E** y **G**. **C** está a 8 de distancia de **B**, **E** está a 7 de distancia de **B**, y **G** está a 11 de distancia de **F**. **E** está más cerca, entonces marcamos el vértice **E** y la arista **EB**. Otras dos aristas fueron marcadas en rojo porque ambos vértices que unen fueron agregados al árbol.	null	C, G	A, D, F, B, E
	Sólo quedan disponibles **C** y **G**. **C** está a 5 de distancia de **E**, y **G** a 9 de distancia de **E**. Se elige **C**, y se marca con el arco **EC**. El arco **BC** también se marca con rojo.	null	G	A, D, F, B, E, C
	**G** es el único vértice pendiente, y está más cerca de **E** que de **F**, así que se agrega **EG** al árbol. Todos los vértices están ya marcados, el árbol de expansión mínimo se muestra en verde. En este caso con un peso de 39.	null	null	A, D, F, B, E, C, G

Referencias

- R. C. Prim: *Shortest connection networks and some generalisations*. In: *Bell System Technical Journal*, 36 (1957), pp. 1389–1401
- D. Cherition and R. E. Tarjan: *Finding minimum spanning trees*. In: *SIAM Journal of Computing*, 5 (Dec. 1976), pp. 724–741
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 23.2: The algorithms of Kruskal and Prim, pp.567–574.

Enlaces externos

- Ejemplos usando JAVA (incluyen código) [1] por Kenji Ikeda, Ph.D
- Create and Solve Mazes by Kruskal's and Prim's algorithms [2]
- Animated example of Prim's algorithm [3]
- Ejemplo interactivo (Java Applet) [4]
- Ejemplo interactivo en español(Java Applet) [5]
- Prim's algorithm code [6]

Referencias

[1] http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Prim.shtml
[2] http://www.cut-the-knot.org/Curriculum/Games/Mazes.shtml
[3] http://students.ceid.upatras.gr/~papagel/project/prim.htm
[4] http://www.mincel.com/java/prim.html
[5] http://www.dma.fi.upm.es/java/matematicadiscreta/
[6] http://www.algorithm-code.com/wiki/Prim%27s_algorithm

Fuentes y contribuyentes del artículo

Algoritmo de Prim *Fuente*: http://es.wikipedia.org/w/index.php?oldid=63381910 *Contribuyentes*: Abelacoa, Akkan, Alvarovmz, Cesarsorm, Dariog88, Davidrodriguez, Definol, Farisori, GermanX, Jevy14214, LordT, Macarse, Miguelio, Paintman, Platonides, Porao, Riviera, Rondador, Tano4595, Tirithel, 69 ediciones anónimas

Fuentes de imagen, Licencias y contribuyentes

Archivo:Prim Algorithm 0.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Prim_Algorithm_0.svg *Licencia*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contribuyentes*: Alexander Drichel

Archivo:Prim Algorithm 1.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Prim_Algorithm_1.svg *Licencia*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contribuyentes*: Alexander Drichel, Stefan Birkner

Archivo:Prim Algorithm 2.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Prim_Algorithm_2.svg *Licencia*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contribuyentes*: Alexander Drichel, Stefan Birkner

Archivo:Prim Algorithm 3.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Prim_Algorithm_3.svg *Licencia*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contribuyentes*: Alexander Drichel, Stefan Birkner

Archivo:Prim Algorithm 4.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Prim_Algorithm_4.svg *Licencia*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contribuyentes*: Alexander Drichel, Stefan Birkner

Archivo:Prim Algorithm 5.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Prim_Algorithm_5.svg *Licencia*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contribuyentes*: Alexander Drichel, Stefan Birkner

Archivo:Prim Algorithm 6.svg *Fuente*: http://es.wikipedia.org/w/index.php?title=Archivo:Prim_Algorithm_6.svg *Licencia*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contribuyentes*: Alexander Drichel, Stefan Birkner

Licencia