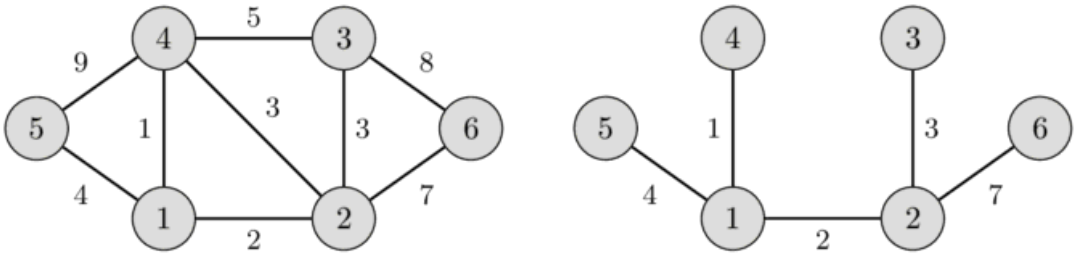
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO PRIM

1. Introducción

Dado un grafo no dirigido ponderado. Queremos encontrar un subárbol de este grafo que conecte todos los vértices (es decir, es un árbol de expansión) y tiene el menor peso (es decir, la suma de los pesos de todos las aristas es mínima) de todos los árboles de expansión posibles. Este árbol de expansión se denomina árbol de expansión mínimo.

En la imagen de la izquierda puede ver un grafo no dirigido ponderado, y en la imagen de la derecha puede ver el árbol de expansión mínimo correspondiente.



2. Conocimientos previos

2.1. Representación de grafos

Existen diferentes formas de representar un grafo. La estructura de datos usada depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

La estructura de datos usada depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

2.2. Grafo denso

En teoría de grafos, la densidad de un grafo es una propiedad que determina la proporción de aristas que posee. Un grafo denso es un grafo en el que el número de aristas es cercano al número máximo de aristas posibles, es decir, a las que tendría si el grafo fuera completo. Al contrario, un grafo disperso es un grafo con un número de aristas muy bajo, es decir, cercano al que tendría si fuera un grafo vacío.

2.3. Cola con prioridad

Una cola de prioridad es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen, puede llegar a ofrecer la extracción constante de tiempo del elemento más grande (por defecto), a expensas de la inserción logarítmica, o utilizando greater<int>causaría que el menor elemento aparezca como la parte superior con .top(). Trabajar con una cola de prioridad es similar a la gestión de un heap

2.4. Árbol de expasión mínima

Dado un grafo conexo y no dirigido, un árbol recubridor, árbol de cobertura o árbol de expansión de ese grafo es un subgrafo que tiene que ser un árbol y contener todos los vértices del grafo inicial. Cada arista tiene asignado un peso proporcional entre ellos, que es un número representativo de algún objeto, distancia, etc.; y se usa para asignar un peso total al árbol recubridor mínimo computando la suma de todos los pesos de las aristas del árbol en cuestión. Un árbol recubridor mínimo o un árbol de expansión mínimo es un árbol recubridor que pesa menos o igual que todos los otros árboles recubridores. Todo grafo tiene un bosque recubridor mínimo.

3. Desarrollo

Este algoritmo fue descubierto originalmente por el matemático checo Vojtěch Jarník en 1930. Sin embargo, este algoritmo se conoce principalmente como el algoritmo de Prim en honor al matemático estadounidense Robert Clay Prim, quien lo redescubrió y volvió a publicar en 1957. Además, Edsger Dijkstra publicó este algoritmo en 1959.

Aquí describimos el algoritmo en su forma más simple. El árbol de expansión mínimo se construye gradualmente añadiendo aristas de una en una. Al principio, el árbol de expansión consta de un solo vértice (elegido arbitrariamente). Luego, el borde de peso mínimo que sale de este vértice se selecciona y se agrega al árbol de expansión. Después de eso, el árbol de expansión ya consta de dos vértices. Ahora seleccione y agregue el borde con el peso mínimo que tiene un extremo en un vértice ya seleccionado (es decir, un vértice que ya es parte del árbol de expansión) y el otro extremo en un vértice no seleccionado. Y así sucesivamente, es decir, cada vez que seleccionamos y agregamos el borde con peso mínimo que conecta un vértice seleccionado con un vértice no seleccionado. El proceso se repite hasta que el árbol de expansión contenga todos los vértices (o de manera equivalente hasta que tengamos $n - 1$ aristas).

El siguiente ejemplo ilustra el funcionamiento del algoritmo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo. La primera imagen muestra el grafo pesado y las siguientes muestran el funcionamiento del algoritmo de Prim y como va cambiando el conjunto U durante la ejecución.

Al final, el árbol de expansión construido será mínimo. Si el gráfico no estaba conectado originalmente, entonces no existe un árbol de expansión, por lo que el número de aristas seleccionadas será menor que $n - 1$.

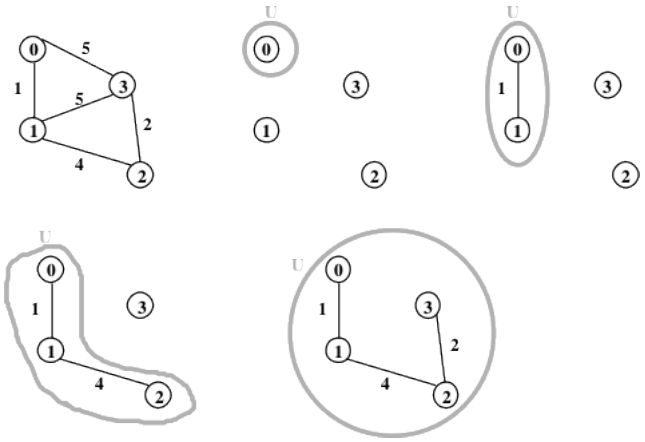


Figura 1: Ejecucción del algoritmo Prim.

Lo siguiente es un pseudocódigo del algoritmo que utiliza como estructura de datos auxiliar una cola com prioridad la cual 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
      Adicionar(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 pertenece cola) && (distancia[v] > peso(u, v)) entonces
            padre[v] = u
            distancia[v] = peso(u, v)
            Actualizar(cola,<v,distancia[v]>)
```

La complejidad del algoritmo depende de cómo busquemos la siguiente arista mínima entre las aristas apropiadas. Hay múltiples enfoques que conducen a diferentes complejidades y diferentes implementaciones.

3.1. Solución trivial

Si buscamos la arista iterando sobre todas las aristas posibles, entonces toma $O(m)$ tiempo para encontrar la arista con el mínimo peso. La complejidad total será $O(nm)$. En el peor de los casos

esto es $O(n^3)$, realmente lento.

Este algoritmo se puede mejorar si solo miramos un borde de cada vértice ya seleccionado. Por ejemplo, podemos clasificar las aristas de cada vértice en orden ascendente de sus pesos y almacenar un puntero en la primera arista válida (es decir, una arista que va a un vértice no seleccionado). Luego, después de encontrar y seleccionar el borde mínimo, actualizamos los punteros. Esto da una complejidad de $O(n^2 + m)$, y para ordenar los bordes un adicional $O(m \log n)$, lo que da la complejidad $O(n^2 \log n)$ En el peor de los casos.

A continuación, consideramos dos algoritmos ligeramente diferentes, uno para gráficos densos y otro para gráficos dispersos, ambos con una mayor complejidad.

3.2. Grafos densos

Abordamos este problema desde un ángulo diferente: para cada vértice aún no seleccionado almacenaremos el borde mínimo en un vértice ya seleccionado. Luego, durante un paso, solo tenemos que mirar estos bordes de peso mínimo, que tendrán una complejidad de $O(n)$. Después de agregar un borde, se deben volver a calcular algunos punteros de borde mínimos. Tenga en cuenta que los pesos solo pueden disminuir, es decir, el borde de peso mínimo de cada vértice aún no seleccionado puede permanecer igual, o se actualizará por un borde al vértice recién seleccionado. Por lo tanto, esta fase también se puede hacer en $O(n)$.

3.3. Grafos dispersos

En el algoritmo descrito anteriormente es posible interpretar las operaciones de encontrar el mínimo y modificar algunos valores como operaciones de conjunto. Estas dos operaciones clásicas son compatibles con muchas estructuras de datos, por ejemplo, *set* en C ++ (que se implementan a través de árboles rojo-negro).

El algoritmo principal sigue siendo el mismo, pero ahora podemos encontrar la arista mínimo en $O(\log n)$ tiempo. Por otro lado, volver a calcular los punteros ahora tomará $O(n \log n)$ tiempo, que es peor que en el algoritmo anterior.

4. Implementación

4.1. C++

4.1.1. Grafos densos

```
int n; //cantidad de nodos
vector<vector<int> > adj; //matrix de adyacencia
const int INF = 1000000000; //el peso INF significa que no hay arista
struct Edge { int w = INF, to = -1; };

void prim() {
   int total_weight = 0;
   vector<bool> selected(n, false);
```

```
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for(int i=0;i<n;++i){
        int v = -1;
        for(int j = 0; j < n; ++j) {
            if(!selected[j] && (v==-1 || min_e[j].w<min_e[v].w))
                v = j;
        }
        if(min_e[v].w == INF) {
            cout << "No existe MST!" << endl;
            i=n+1;
            continue;
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if(min_e[v].to!=-1)
            cout <<v<<" "<<min_e[v].to<<endl;
        for(int to=0;to<n;++to){
            if(adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }
    cout << total_weight << endl;
}
```

La matriz de adyacencia adj[][] de tamaño $n \times n$ almacena los pesos de las aristas, y usa el peso INF si no existe una arista entre dos vértices. El algoritmo utiliza dos matrices: el vector selected[], que indica qué vértices ya hemos seleccionado, y la matriz min_e[]que almacena la arista con peso mínimo en un vértice seleccionado para cada vértice aún no seleccionado (almacena el peso y el vértice final). El algoritmo hace n pasos, en cada iteración se selecciona el vértice con el peso de borde más pequeño y min_e[]se actualiza el de todos los demás vértices.

4.1.2. Grafos dispersos

```
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to);
    }
};

int n;
vector<vector<Edge>> adj;
```

```
void prim() {
   int total_weight = 0;
   vector<Edge> min_e(n);
   min_e[0].w = 0;
   set<Edge> q;
   q.insert({0, 0});
   vector<bool> selected(n, false);
   for(int i = 0; i<n; ++i){
      if(q.empty()) {
         cout << "No existe MST!" << endl;
         i=n+1;
         continue;
      }
      int v = q.begin()->to;
      selected[v] = true;
      total_weight += q.begin()->w;
      q.erase(q.begin());
      if(min_e[v].to != -1)
         cout << v << " " << min_e[v].to << endl;
      for(Edge e : adj[v]) {
         if (!selected[e.to] && e.w < min_e[e.to].w) {
            q.erase({min_e[e.to].w, e.to});
            min_e[e.to] = {e.w, v};
            q.insert({e.w, e.to});
         }
      }
   }
   cout << total_weight << endl;
}
```

Aquí, el grafo se representa a través de una lista de adyacencia adj[], donde adj[v] contiene todas las aristas (en forma de pares de peso y vértice) para el vértice v. min_e[v] almacenará el peso del borde más pequeño desde el vértice v hasta un vértice ya seleccionado (nuevamente en forma de un par de peso y vértice). Además, la cola q se llena con todos los vértices aún no seleccionados en orden de peso creciente min_e. El algoritmo realiza n pasos, en cada uno de los cuales selecciona el vértice v con el peso más pequeño min_e (extrayéndolo del principio de la cola), y luego examina todas las aristas este vértice y actualiza los valores en min_e (durante una actualización también también es necesario eliminar la antigua arista de la cola q y coloque la nueva arista).

4.2. Java

4.2.1. Grafos densos

```
public long mstPrim(int[][] d) {
   int n = d.length;
   int[] prev = new int[n];
   int[] dist = new int[n];
```

```
   Arrays.fill(dist, Integer.MAX_VALUE);
   dist[0] = 0;
   boolean[] visited = new boolean[n];
   long res = 0;
   for(int i=0;i<n;i++){
      int u = -1;
      for(int j=0;j<n;j++){
         if(!visited[j] && (u==-1 || dist[u]>dist[j])) u = j;
      }
      res += dist[u]; visited[u] = true;
          for(int j=0;j<n;j++){
         if(!visited[j] && dist[j]>d[u][j]){
            dist[j] = d[u][j]; prev[j] = u;
         }
      }
   }
   return res;
}
```

4.2.2. Grafos dispersos

```
public long mstPrim(List<Edge>[] edges, int[] pred) {
   int n = edges.length;
   Arrays.fill(pred, -1);
   boolean[] used = new boolean[n];
   int[] prio = new int[n];
   Arrays.fill(prio, Integer.MAX_VALUE);
   prio[0] = 0;
   PriorityQueue<Long> q = new PriorityQueue<>();
   q.add(0L);
   long res = 0;

   while (!q.isEmpty()) {
      long cur = q.poll();
      int u = (int) cur;
      if (used[u]) continue;
      used[u] = true;
      res += cur >>> 32;
      for (Edge e : edges[u]) {
         int v = e.t;
         if(!used[v] && prio[v] > e.cost) {
            prio[v] = e.cost; pred[v] = u;
            q.add(((long) prio[v] << 32) + v);
         }
      }
   }
   return res;
}
```

```
private class Edge {
   int t, cost;
   public Edge(int t, int cost) {
      this.t = t; this.cost = cost;
   }
}
```

5. Complejidad

5.1. Grafos densos

La versión del algoritmo de Prim para este tipo de grafos tendría una complejidad $O(n^2)$.

5.2. Grafos dispersos

Como consideramos que solo necesitamos actualizar $O(m)$ veces en total, y realizar $O(n)$ busca el borde mínimo, entonces la complejidad total será $O(m \log n)$. Para grafos dispersos esto es mejor que el algoritmo anterior, pero para grafos densos será más lento.

6. Aplicaciones

Este algoritmo aparece de forma bastante natural como solución en muchos problemas. Por ejemplo, en el siguiente problema: hay n ciudades y para cada par de ciudades se nos da el costo de construir una carretera entre ellas (o sabemos que es físicamente imposible construir una carretera entre ellas). Tenemos que construir caminos, de modo que podamos ir de una ciudad a otra ciudad, y el costo de construir todos los caminos es mínimo.

En particular, esta implementación para grafos densos es muy conveniente para el problema del árbol de expansión mínimo euclidiano:tenemos n puntos en un plano y la distancia entre cada par de puntos es la distancia euclidiana entre ellos, y queremos encontrar un árbol generador mínimo para este grafo completo. Esta tarea puede ser resuelta por el algoritmo descrito para este tipo de grafo en $O(n^2)$ tiempo y $O(n)$ memoria, lo que no es posible con el algoritmo de Kruskal .

El algoritmo de Prim resuelve el mismo problema que el algoritmo de Kruskal pero es más optimo cuando trabajamos sobre un grafo denso mientras el Kruskal es mas óptimo para grafos dispersos.

7. Ejercicios propuestos

A continuación una lista de que se pueden resolver aplicando este algoritmo:

- DMOJ - MooCast
- DMOJ - Pozos de Agua