Árboles de cubrimiento mínimo (*Minimum Spanning Trees*)

Algoritmos de Kruskal y de Prim

Definición

Sea

- G=<V,E> un grafo conexo, no dirigido.
- A cada arista tiene un costo o una longitud (c_i) positiva.

Problema:

- Encontrar $T \subseteq E$ tal que utilizando solo las aristas en T todos los vértices queden conectados y que

$$\sum_{i \in T} c_i$$

sea tan pequeña como sea posible.

Características de una solución óptima

Consideremos el grafo G'=<V,T>

- Si T tiene menos de V-1 aristas, entonces no sería conexo.
- Si T contiene más de V-1 aristas, se le puede quitar una, sigue siendo conexo y el costo total se reduce.
- Un grafo de V vértices con más de V-1 aristas contiene al menos un ciclo.

Se concluye entonces

- T debe ser un árbol. Por esta razón G' se denomina *árbol de cubrimiento mínimo*.

Idea general de solución mediante un algoritmo voráz

- Empezar con T vacío
- Añadir a T la arista más corta que no forme un ciclo y no haya sido seleccionada o rechazada previamente, (1)

ó

- Seleccionar un nodo y construir un árbol agregando la arista más corta que extienda el árbol. (2)

Algoritmos voraces (*Greedy Algorithms*)

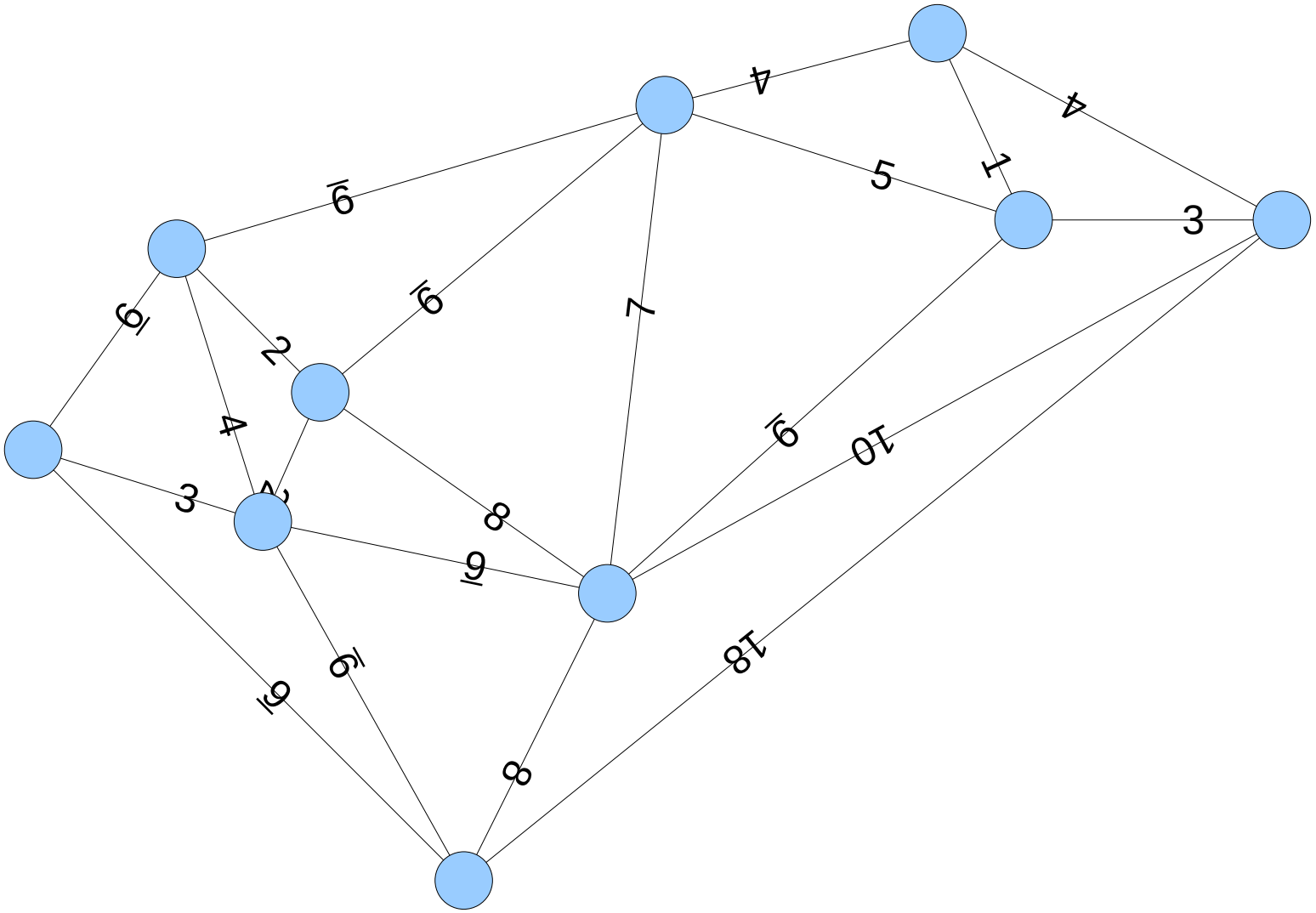
- Algoritmos que parten de:
  - Un conjunto de candidatos,
  - Un conjunto solución inicialmente vacío
- En cada iteración seleccionan un candidato, el cual puede ser descartado o agregado al conjunto solución.
- El algoritmo siempre termina, como máximo al agotar el conjunto de candidatos.

Discusión

En ambos casos se tienen:

- Candidatos: E, conjunto de aristas de G
- Solución: Las aristas seleccionadas T son un árbol de cubrimiento mínimo
- Factible: El conjunto seleccionado no contiene ningún ciclo
- Selección: Las estrategias (1) ó (2)
- Objetivo: Minimizar la suma de las longitudes de las aristas seleccionadas

Ejemplo



Por qué estas estrategias voraces llegan a la solución óptima?

Lema:

- Sea G=<V,A> un grafo conexo, no dirigido con $c_i \geq 0$.
- Sea $B \subset V$ un subconjunto de vértices.
- Sea $T \subseteq A$ un conjunto *prometedor* de aristas tal que ninguna arista de T sale de B.
- Sea v la arista más corta que sale de B.

entonces

$T \cup \{v\}$ es *prometedor*

Cortes de un grafo

- Si se particionan los vértices en dos conjuntos, el corte es el conjunto de aristas entre ambos conjuntos.
- La arista más corta de un corte pertenece a un árbol de cubrimiento mínimo.
- Ambas estrategias de solución agregan la arista más corta de un corte a T en cada iteración.

Algoritmo de Prim

- Se empieza en un vértice cualquiera
- Se agregan las aristas que salen de este vértice a una MinPQ: Aristas que salen del conjunto conexo.
- En cada paso se toma la menor arista de la MinPQ: {v,w}
- Se agrega el otro vértice w de esta arista
- Se agregan a la MinPQ las aristas que salen de w

Una primera implementación LazyPrimMST

Variables e inicialización del algoritmo:

```
private double weight;
private Queue<Edge> mst;
private boolean[] marked;
private MinPQ<Edge> pq;

public LazyPrimMST(EdgeWeightedGraph
G) {
    mst = new Queue<Edge>();
    pq = new MinPQ<Edge>();
    marked = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
        if (!marked[v]) prim(G, v);
}
```

Fuente del texto guía: LazyPrimMST.java

LazyPrimMST

```
private void prim(EdgeWeightedGraph G, int s) {
    scan(G, s);
    while (!pq.isEmpty()) {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        assert marked[v] || marked[w];
        if (marked[v] && marked[w]) continue;
        mst.enqueue(e);
        weight += e.weight();
        if (!marked[v]) scan(G, v);
        if (!marked[w]) scan(G, w);
    }
}

private void scan(EdgeWeightedGraph G, int v) {
    assert !marked[v];
    marked[v] = true;
    for (Edge e : G.adj(v))
        if (!marked[e.other(v)]) pq.insert(e);
}
```

Fuente del texto guía: LazyPrimMST.java

Análisis de LazyPrimMST

- Observamos que todas y cada una de las aristas se agregaron y se eliminaron una vez de la cola: ~ 2 lg(E)
- Aristas entre vértices ya seleccionados se descartan: ~c
- Aristas del MST se agregan a una cola simple: ~c
- El tiempo total del algoritmo es:

$$\sim E \lg(E)$$

- Requerimiento de espacio: ~ 2*V + E

Implementación proactiva (*eager* Prim)

- La cola MinPQ contiene muchas aristas innecesarias
- Solo se necesita la arista más corta hacia vértices que aún no estén en el árbol

PrimMST

Variables e inicialización del algoritmo:

```
private Edge[] edgeTo;
private double[] distTo;
private boolean[] marked;
private IndexMinPQ<Double> pq;

public PrimMST(EdgeWeightedGraph G) {
    edgeTo = new Edge[G.V()];
    distTo = new double[G.V()];
    marked = new boolean[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;

    for (int v = 0; v < G.V(); v++)
        if (!marked[v]) prim(G, v);
}
```

Fuentes texto guía: PrimMST.java

PrimMST

```
private void prim(EdgeWeightedGraph G, int s) {
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        scan(G, v);
    }
}

private void scan(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v)) {
        int w = e.other(v);
        if (marked[w]) continue;
        if (e.weight() < distTo[w]) {
            distTo[w] = e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }
}
```

Fuentes texto guía: PrimMST.java

Análisis del PrimMST

- El ciclo principal visita cada vértice una vez: V operaciones delMin: ~V lg(V)
- El proceso de escaneo considera cada arista una vez, pero la longitud de la cola será como máximo V: ~E lg(V)
- Tiempo total:

$$\sim E \lg(V)$$

- Espacio adicional utilizado: 4 estructuras auxiliares de tamaño kV.

Algoritmo de Kruskal

- Se procesan todas las aristas en orden de su peso. Para esto se usa una MinPQ.
- Para cada arista se verifica que la no forme un ciclo, es decir no conecte vértices que ya se encuentren conectados en T. Para esto hace uso de la estructura Union/Find.
- El conjunto de aristas seleccionadas se almacena en una cola simple: Queue.

KruskalMST

```
private double weight;
private Queue<Edge> mst = new Queue<Edge>();

public KruskalMST(EdgeWeightedGraph G) {
    MinPQ<Edge> pq = new MinPQ<Edge>();
    for (Edge e : G.edges()) {
        pq.insert(e);
    }

    UF uf = new UF(G.V());
    while (!pq.isEmpty() && mst.size() < G.V() - 1) {
        Edge e = pq.delMin();
        int v = e.either();
        int w = e.other(v);
        if (uf.find(v) != uf.find(w)) {
            uf.union(v, w);
            mst.enqueue(e);
            weight += e.weight();
        }
    }
}
```

Fuente del texto guía: KruskalMST.java

Análisis de Kruskal

- Se recorren todas las aristas del grafo
- Por cada arista se hace una operación `insert` y una operación `delMin`: ~ 2*E*lg(E)
- Por cada arista se hacen 2 operaciones `find` y una operación `union`: ~ 3*E*lg(V)
- Tiempo total:

$$\sim 2*E*\lg(E) + 3*E*\lg(V)$$

- Espacio total:
  - MinPQ: ~ $k_1 E$
  - UF: ~$k_2 V$
  - Queue: ~$k_3 V$