Semillero de Programación

Árbol de mínima expansión

Ana Echavarría Juan Francisco Cardona

Universidad EAFIT

19 de abril de 2013

Contenido

Contenido

Problema A - Minesweeper

Para cada una de las celdas que no tienen minas, recorrer las 8 celdas adyacentes y contar el número de esas celdas que tengan una mina.

Implementación I

```
1  string board [105];
2  int rows, cols;
3
4  int di [] = {-1, -1, -1, +0, +0, +1, +1, +1};
5  int dj [] = {-1, +0, +1, -1, +1, -1, +0, +1};
6
7  bool in(int i, int j){
8      if (i >= 0 and i < rows and j >= 0 and j < cols) return true;
9      return false;
10 }
11
12 void solve (){
13     for (int i = 0; i < rows; i++){
14         for (int j = 0; j < cols; j++){
15             if (board[i][j] == '.') board[i][j] = '0';
16         }
17     }
18
```

Implementación II

```
19      for (int i = 0; i < rows; i++){
20          for (int j = 0; j < cols; j++){
21              if (board[i][j] == '*') continue;
22              for (int k = 0; k < 8; k++){
23                  int next_i = i + di[k];
24                  int next_j = j + dj[k];
25                  if (!in(next_i, next_j)) continue;
26                  if (board[next_i][next_j] == '*'){
27                      board[i][j]++;
28                  }
29              }
30          }
31      }
32
33      for (int i = 0; i < rows; i++){
34          for (int j = 0; j < cols; j++){
35              printf("%c", board[i][j]);
36          }
37          puts("");
```

Implementación III

```
38      }
39  }
40
41
42  int main(){
43      int run = 1;
44      while (cin >> rows >> cols){
45          if (rows == 0 and cols == 0) break;
46          if (run != 1) printf("\n");
47          for (int i = 0; i < rows; i++){
48              cin >> board[i];
49          }
50
51          printf("Field #%d:\n", run++);
52          solve();
53      }
54      return 0;
55  }
```

Problema B - The Tourist Guide

- Hallar el camino entre la ciudad s y ls d que pueda transportar el mayor número de personas posible.
- De todos los caminos entre s y d hay que hallar el que la ruta de bus que pueda transportar menos gente sea lo más grande posible.
- En otras palabras, hay que hallar el camino cuya mínima arista sea lo más grande posible (maximin).

- Luego de hallar la capacidad de dicha ruta hay que ver cuántos viajes hay que hacer teniendo en cuenta que el guía debe estar en cada viaje.
- Esto implica que en realidad la capacidad es 1 menor (el puesto que ocupa el guía).
- El número de viajes que hay que hacer es el techo de la división entre el número de pasajeros y la capacidad de la ruta.

Función techo

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a+b-1}{b} \right\rfloor = \left\lfloor \frac{a-1}{b} \right\rfloor + 1$$

Implementación I

```
1  const int MAXN = 105;
2  const int INF = 1 << 30;
3  int g[MAXN][MAXN];
4
5  int main(){
6     int n, m;
7     int scenario = 1;
8
9     while (cin >> n >> m){
10       if (n == 0 and m == 0) break;
11
12       for (int i = 0; i <= n; ++i){
13          for (int j = 0; j <= n; ++j){
14             g[i][j] = -INF;
15          }
16          g[i][i] = INF;
17       }
18
```

Implementación II

```
19
20
21          for (int i = 0; i < m; ++i){
22              int u, v, p;
23              cin >> u >> v >> p;
24              u--; v--;
25              g[u][v] = g[v][u] = p;
26          }
27
28          for (int k = 0; k < n; ++k){
29              for (int i = 0; i < n; ++i){
30                  for (int j = 0; j < n; ++j){
31                      g[i][j] = max(g[i][j], min(g[i][k] , g[k][j]));
32                  }
33              }
34          }
35
36
37
```

Implementación III

```
38          int s, d, t;
39          cin >> s >> d >> t;
40
41          // Recordar que el guia debe viajar en el bus
42          int max_cap = g[s-1][d-1] - 1;
43          // Tomar el techo de la division
44          int trips = (t + max_cap - 1) / max_cap;
45
46          printf("Scenario #%d\n", scenario++);
47          printf("Minimum Number of Trips = %d\n\n", trips);
48      }
49       return 0;
50  }
```

Problema C - Page Hopping

- Los nodos pueden no ser consecutivos, usar un mapa para guardarlos.
- Hallar la distancia más corta entre cualquier para de nodos.
- Para cada nodo, sumar la distancia a los demás $n - 1$ nodos.
- Para sacar el promedio, dividir entre el número de distancias sumadas $n * (n - 1)$.

Implementación I

```
1  const int MAXN = 105;
2  const int INF = 1 << 25;
3  map <int, int> m;
4  int d[MAXN][MAXN];
5
6  int main(){
7     int run = 1;
8
9     int a, b;
10    while(cin >> a >> b){
11       if (a == 0 and b == 0) break;
12
13       for (int i = 0; i < MAXN; ++i) {
14          for (int j = 0; j < MAXN; ++j){
15             d[i][j] = INF;
16          }
17          d[i][i] = 0;
18       }
```

Implementación II

```
19          m.clear();
20
21          int node_count = 0;
22          do{
23              if (m.count(a) == 0) m[a] = node_count++;
24              if (m.count(b) == 0) m[b] = node_count++;
25              d[m[a]][m[b]] = 1;
26              cin >> a >> b;
27          }while(a != 0 or b != 0);
28
29          for (int k = 0; k < node_count; ++k){
30              for (int i = 0; i < node_count; ++i){
31                  for (int j = 0; j < node_count; ++j){
32                      d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
33                  }
34              }
35          }
36
37
```

Implementación III

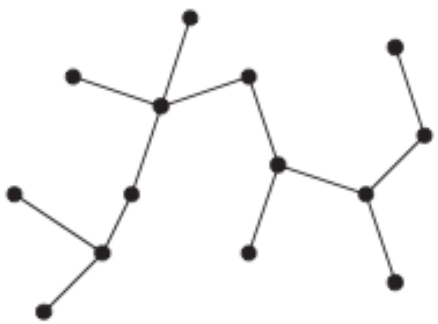
```
38          int sum = 0;
39          for (int i = 0; i < node_count; ++i){
40              for (int j = 0; j < node_count; ++j){
41                  sum += d[i][j];
42              }
43          }
44          // Multiplicar por 1.0 para convertir a doble y hacer
                division decimal
45          double ans = 1.0 * sum / (node_count * (node_count - 1));
46          // Imprimir con tres decimales de precision
47          printf("Case %d: average length between pages = %.3lf
                    clicks\n", run++, ans);
48
49      }
50       return 0;
51  }
```

Contenido

Árboles

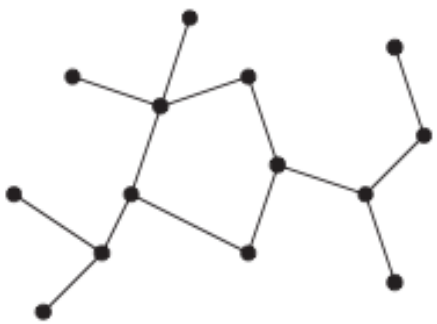
Árboles

Un árbol es un grafo **no dirigido, conexo, y acíclico**.
Un grafo no dirigido y acíclico pero no necesariamente conexo (todos los nodos están conectados) es un bosque.



(a)

(b)



(c)

Propiedades de los árboles

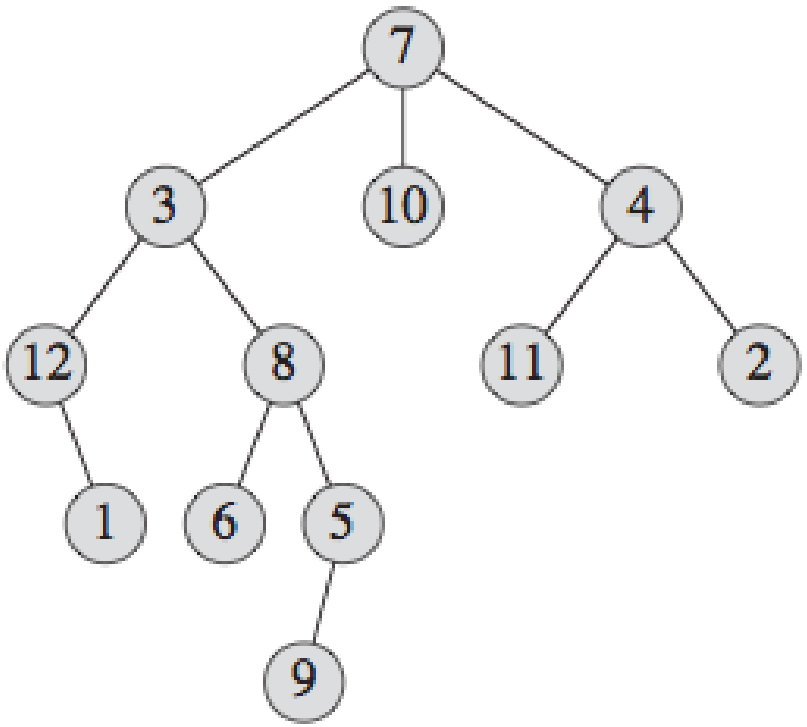
Propiedades

Sea $G = (V, E)$ un grafo no dirigido. Los siguientes enunciados son equivalentes:

- G es un árbol
- Hay un único camino entre cualquier par de nodos $u, v \in V$
- G es un grafo conexo y $|E| = |V| - 1$

Árboles con raíces

Un árbol $T = (V, E)$ en donde uno de sus nodos se diferencia de los demás es un árbol con raíz (rooted tree). El nodo r que se diferencia de los demás se llama raíz.



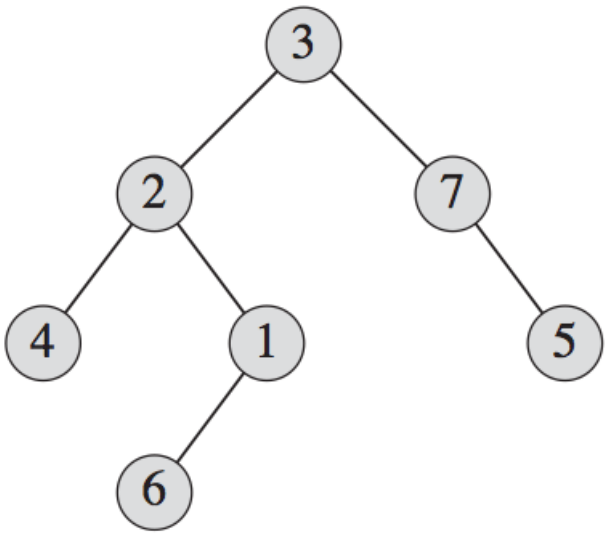
Árboles con raíces

Sea $T = (V, E)$ un árbol con raíz $r \in V$. Sean $x, y \in V$.

- Cualquier nodo y que pertenezca al camino de r a x (recordemos que sólo hay un camino) es llamado **ancestro** de x y x a su vez es llamado **descendiente** de y.
- Si y es el último nodo en el camino de r a x, y es llamado **padre** de x y x es llamado **hijo** de y.
- Si dos nodos tienen el mismo padre estos son llamados **hermanos**.
- Un nodo que no tiene hijos es llamado **hoja**.

Árboles binarios

Un árbol binario T es un árbol con raíz en el que todos sus nodos tienen 0, 1 o 2 hijos.
Los árboles binarios tienen gran aplicación en las estructuras de datos.
Algunas estructuras de datos que utilizan árboles binarios para almacenar sus datos son: set, map y heap (`priority_queue`).



Contenido

Minimum Spanning Tree

Entrada

Un grafo $G = (V, E)$ no dirigido y conexo
Una función de pesos $w(u, v)$ donde $(u, v) \in E$

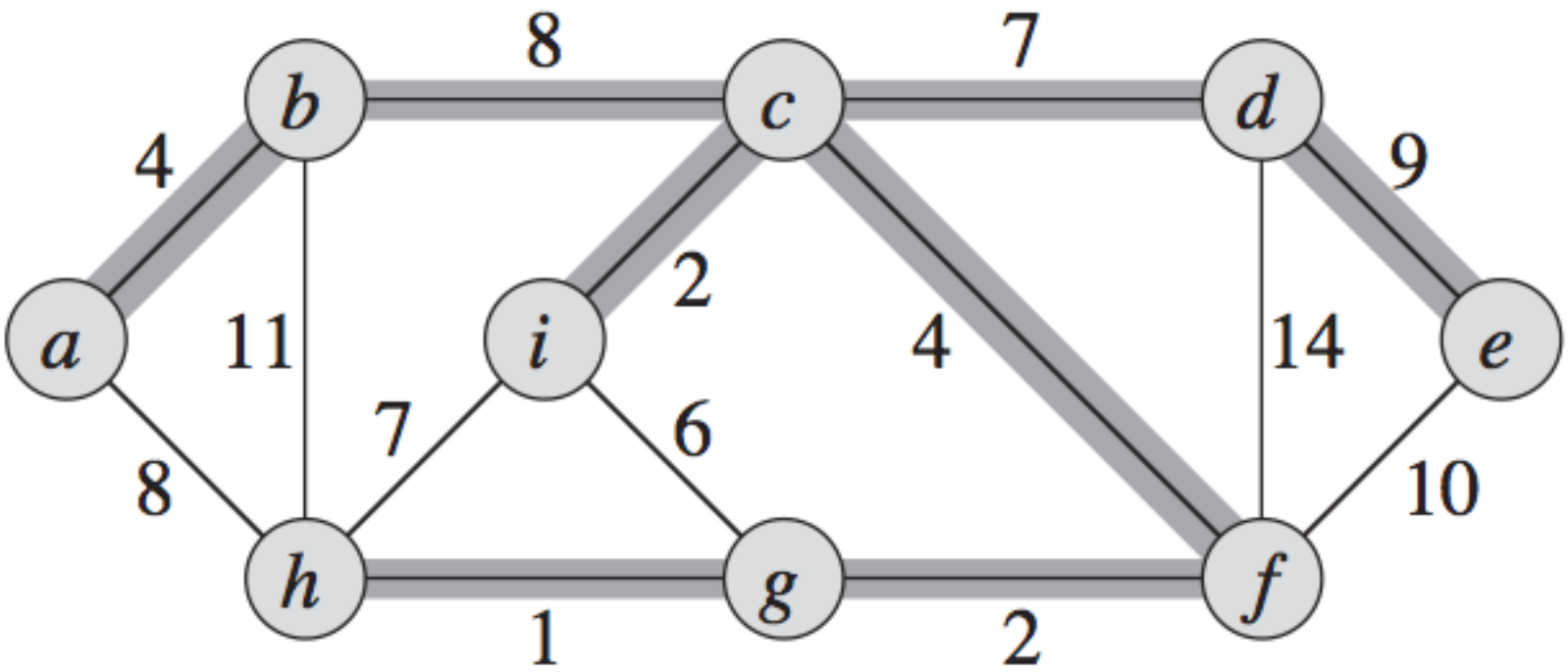
Objetivo

Hallar un conjunto no cíclico $T \subseteq E$ que conecte todos los nodos de G y que minimize su peso

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

Ya que T es no cíclico, conexo y no dirigido, es un árbol y es llamado el **árbol de mínima expansión**.

Minimum Spanning Tree

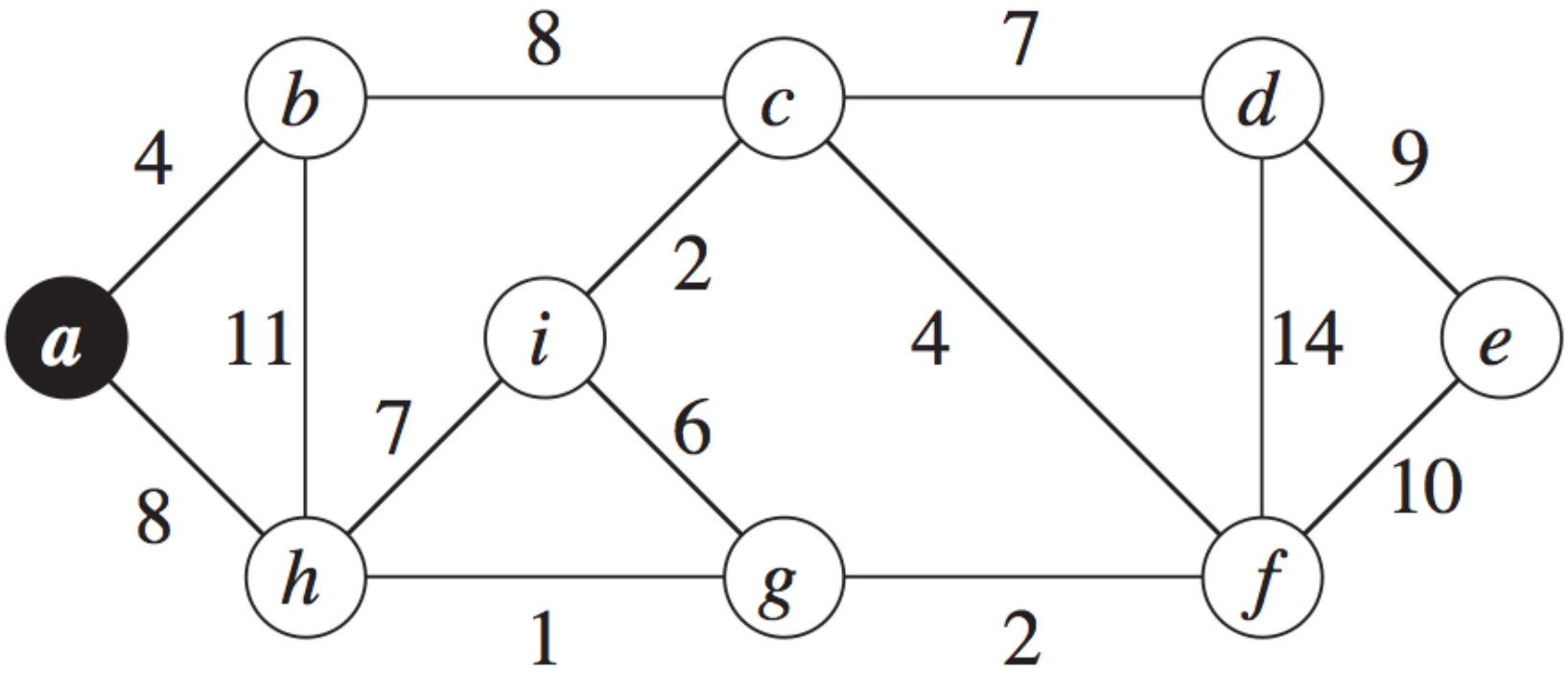


Contenido

Algoritmo de Prim

1. Seleccionar un nodo al azar y agregarlo al conjunto de visitados
2. Mientras que haya nodos sin visitar
    3. Tomar la arista con menor peso que conecte un nodo visitado u y uno no visitado v
    4. Agregar la arista (u, v) al MST
    5. Agregar v al conjunto de visitados

Ejemplo



¿Por qué funciona?

Veamos que el algoritmo produce un árbol.

- En cada iteración, se agrega al árbol un nodo nuevo y una arista que conecta a ese nodo con alguno de los ya elegidos anteriormente. Al final se tendrán todos los nodos conectados.
- Nunca se agrega una arista que una a dos nodos ya elegidos por lo que no se generan ciclos.
- Por las dos afirmaciones anteriores se puede concluir que el algoritmo produce un árbol.

La prueba de optimalidad sale de un teorema que dice que la mínima arista que cruza una partición de un grafo en dos grupos siempre está en el MST de ese grafo.

Optimización

En el algoritmo de Prim se necesita extraer el mínimo de un arreglo repetidas veces. ¿Cómo hacer esto rápidamente?

Optimización

En el algoritmo de Prim se necesita extraer el mínimo de un arreglo repetidas veces. ¿Cómo hacer esto rápidamente?

Estos llamados se pueden hacer rápidamente utilizando un **heap** como se hizo en el algoritmo de Dijkstra.

Implementación I

```
1  const int MAXN = 10005;
2  typedef pair <int, int> edge;
3  bool visited[MAXN];
4  // g[i] = lista de parejas (nodo, peso)
5  vector <pair <int, int> > g[MAXN];
6
7  int prim(int n){
8      for (int i = 0; i <= n; ++i) visited[i] = false;
9      int total = 0;
10     // Crear el heap de forma que se extraiga el de menor peso
11     // El heap es de parejas (peso, nodo), contrario al grafo
12     priority_queue<edge, vector <edge>, greater<edge> > q;
13     // Empezar el MST desde el nodo 0
14     q.push(edge(0, 0));
15     while (!q.empty()){
16         int u = q.top().second;
17         int w = q.top().first;
18         q.pop();
```

Implementación II

```
19          // Si es una arista entre dos nodos ya incluidos al MST
20          if (visited[u]) continue;
21
22          visited[u] = true;
23          total += w;
24          for (int i = 0; i < g[u].size(); ++i){
25              int v = g[u][i].first;
26              int next_w = g[u][i].second;
27              // Si v no pertenece todavia al MST
28              if (!visited[v]){
29                  // Insertar primero el peso y luego el nodo
30                  q.push(edge(next_w, v));
31              }
32          }
33      }
34      return total; // El costo total del MST
35  }
```

Complejidad

Complejidad

El algoritmo de Prim implementado con un heap tiene una complejidad de $O(E \log V)$

Contenido

Union-Find

Union-Find es una estructura de datos para almacenar una colección conjuntos disjuntos (no tienen elementos en común) que cambian dinámicamente.

Para hacer esto identifica en cada conjunto un "padre" que es un elemento al azar de ese conjunto y hace que todos los elementos del conjunto "apunten" hacia ese padre.

Inicialmente se tiene una colección donde cada elemento es un conjunto unitario.

Union-Find

Esta estructura de datos tiene dos operaciones:

Union(u, v) Une los conjuntos que contienen a u y a v en un solo conjunto.

Find(u) Retorna el padre del (único) conjunto que contiene a u.

Implementación

```
1  int p[MAXN]; //p[i] el padre del conjunto al que pertenece i
2
3  // Inicializa los conjuntos de los elementos 0 a n
4  void initialize(int n){
5      for (int i = 0; i <= n; ++i) p[i] = i;
6  }
7  // Retorna el padre del conjunto que contiene a u
8  int find(int u){
9      if (p[u] == u) return u;
10     return p[u] = find(p[u]);
11 }
12 // Une los conjuntos a los que pertenecen u y v
13 // Este nuevo conjunto tiene como padre el padre de v
14 void union(int u, int v){
15      int a = find(u);
16      int b = find(v);
17      if (a == b) return; // Son el mismo conjunto
18      p[a] = b; // El padre del padre de u es el padre de v
19 }
```

Ejemplo

Se tiene inicialmente una colección con 5 conjuntos $\{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}\}$ y se realizan las siguientes operaciones:

1. union(0, 1)
2. union(0, 2)
3. union(1, 3)
4. union(0, 4)

Complejidad

Complejidad

Sean m el número total de operaciones de initialize, union y find realizadas.
La implementación de union-find mostrada tiene una complejidad aproximada de $O(m)$

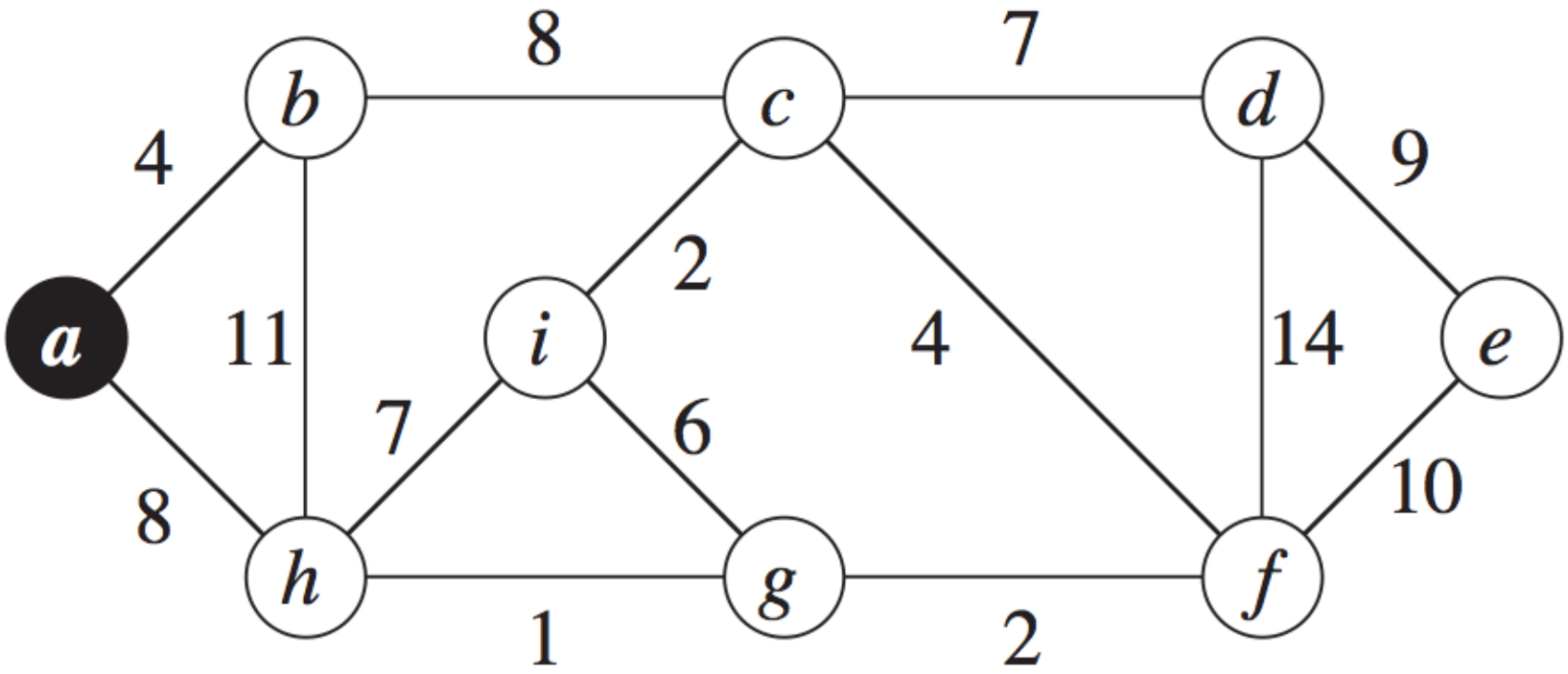
Contenido

Algoritmo de Kruskal

El algoritmo de Kruskal utiliza Union-Find para verificar que al agregar una arista no esté generando un ciclo.

1. Ordenar las aristas de menor costo a mayor costo (desempate es indiferente)
2. Para cada arista (u, v) en el orden de menor a mayor costo
    3. Si agregar la arista (u, v) no genera un ciclo
        4. Agregar la arista (u, v) al MST

Ejemplo



¿Por qué funciona?

- Claramente el grafo resultante no tiene ciclos
- El grafo está conectado porque si no lo estuviera es porque quedaron dos grupos de nodos, que en el grafo original están conectados por al menos una arista y que el algoritmo de Kruskal no seleccionó esa arista pero el algoritmo tuvo que haber seleccionado esa arista porque no generaba un ciclo.
- Por los dos afirmaciones anteriores se concluye que el algoritmo produce un árbol.

La prueba de optimalidad sale de que el algoritmo de Kruskal va a tomar siempre la va a tomar la mínima arista que cruza una una partición del grafo porque empieza por la menor.

Optimización

En el algoritmo de Kruskal se necesita verificar que no se genere un ciclo repetidas veces. ¿Cómo hacer esto rápidamente?

Optimización

En el algoritmo de Kruskal se necesita verificar que no se genere un ciclo repetidas veces. ¿Cómo hacer esto rápidamente?

- A medida que se ejecuta el algoritmo de Kruskal se empiezan a unir los nodos en conjuntos disjuntos.
- Un ciclo se genera si se unen dos nodos que pertenecen al mismo conjunto.
- Se utiliza union-find para guardar el conjunto al que pertenece cada nodo.
- Para verificar se se genera un ciclo es ver si el conjunto de u es igual al conjunto de v.

Implementación I

```
1  // Estrutura personalizada para manejar las aristas.
2  // En el algoritmo de Kruskal el grafo se guarda como lista de
       aristas y no como lista de adyacencia
3  struct edge{
4     // Atributos
5     int start, end, weight;
6     // Constructor
7     cable(int u, int v, int w){
8        start = u; end = v; weight = w;
9     }
10    // Comparador menor
11    bool operator < (const edge &other) const{
12       return weight < other.weight;
13    }
14 }; // No olvidar este ;
15
16
17
```

Implementación II

```
18  const int MAXN = 100005;
19  vector <edge> edges;
20  int p[MAXN];
21
22  int find(int u){
23      if (p[u] == u) return u;
24      return p[u] = find(p[u]);
25  }
26
27  void union(int u, int v){
28       int a = find(u);
29       int b = find(v);
30       if (a == b) return;
31       p[a] = b;
32  }
33
34
35
36
```

Implementación III

```
37  int kruskal(int n){
38      // Inicializar el arreglo de union-find
39      for (int i = 0; i <= n; ++i) p[i] = i;
40      // Ordenar las aristas de menor a mayor
41      sort(edges.begin(), edges.end());
42
43      int total = 0;
44      for (int i = 0; i < edges.size(); ++i){
45          int u = edges[i].start;
46          int v = edges[i].end;
47          int w = edges[i].weight;
48          if (find(u) != find(v)){ // Si no genera un ciclo
49              total += w;
50              union(u, v);
51          }
52      }
53      return total;
54  }
```

Complejidad

Complejidad

El algoritmo de Kruskal implementado con union-find tiene una complejidad de $O(E \log V)$

Contenido

Tarea

Tarea

Resolver los problemas de

Ayudas

Problema A

Problema B

Problema C