Semillero de Programación

Algoritmo de Dijkstra

Ana Echavarría Juan Francisco Cardona

Universidad EAFIT

15 de marzo de 2013

Contenido

1. Problemas semana anterior
   - Problema A - Hardwood Species
   - Problema B - Parentheses Balance
   - Problema C - Dominos
   - Problema D - Word Transformation
2. Grafos con pesos
3. Algoritmo de Dijkstra
4. Tarea

Contenido

1. Problemas semana anterior
   - Problema A - Hardwood Species
   - Problema B - Parentheses Balance
   - Problema C - Dominos
   - Problema D - Word Transformation

Problema A - Hardwood Species

- Para cada especie de árbol hay que guardar cuántas veces aparece.
- Hay que tener un contenedor que guarde esa información cuando se le dé una especie.

Problema A - Hardwood Species

- Para cada especie de árbol hay que guardar cuántas veces aparece.
- Hay que tener un contenedor que guarde esa información cuando se le dé una especie.

Solución

Utilizar un mapa de string (nombre de la especie) a entero (cuántas veces aparece). Luego, recorrer el mapa y hallar la frecuencia relativa de cada especie.

Implementación I

```
1  map <string, int> m;   // Mapa con las frecuencias
2
3  int main(){
4     int cases;  cin >> cases;
5     string tmp;
6     getline(cin, tmp); // Leer el final de la linea de los casos
7     getline(cin, tmp); // Leer la siguiente linea en blanco
8
9     while (cases--){
10       m.clear();     // Limpiar los valores del mapa
11
12       string line;
13       int total = 0; // El numero total de arboles
14       while (getline(cin, line)){
15          if (line == "") break; // Fin del caso de prueba
16          m[line]++;  // Si no existe el valor, lo crea en 0
17          total++;
18       }
```

Implementación II

```
19
20          // Recorrer los elementos del mapa
21          // Los elementos estan ordenados en orden alfabetico
22          map <string, int> :: iterator it;
23          for (it = m.begin(); it != m.end(); it++){
24             // Utilizar 100.0 para que la division no sea entera
25             double percent = 100.0 * it->second / total;
26             // Para imprimir un string con printf usar .c_str()
27             // Imprimir el porcentaje con 4 posiciones decimales
28             printf("%s %.4lf\n", (it->first).c_str(), percent);
29          }
30          // Linea en blanco entre casos
31          if (cases != 0) cout << endl;
32       }
33       return 0;
34    }
```

Problema B - Parentheses Balance

- Cuando se cierra un paréntesis es porque el último que se abrió es del mismo tipo.
- Si se elimina la última pareja de paréntesis / corchetes que se abrió y luego cerró se tiene el mismo problema pero ya hay que mirar que el nuevo paréntesis que cierra sea la pareja de el penúltimo que se abrió (porque ya el último se utilizó en el paso anterior).
- En otras palabras, el último paréntesis / corchete que se abrió va a ser el primero que hay que ver cuando haya uno que cierra (LIFO).

Solución

Solución

Utilizar un stack de caracteres. Insertar un paréntesis / corchete que abre y sacarlo cuando haya uno que cierre.
Detalles de la implementación:

- Verificar que sí haya elementos en el stack cuando se quiera sacar un elemento.
- Verificar que el elemento que se saque sea del mismo tipo (paréntesis / corchete) que el que se tiene.
- Verificar que cuando se termine de recorrer la cadena no queden paréntesis / corchetes abiertos sin emparejar.

Implementación I

```
1  // Hallar el tipo 1-Parentesis, 2-Corchete
2  int type(char c){
3     if (c == '(' or c == ')') return 1;
4     return 2;
5  }
6
7  int main(){
8     int cases;
9     cin >> cases;
10    string endline;
11    getline(cin, endline); // Leer \n despues de los casos
12    while (cases--){
13       string s;
14       getline(cin, s); // Leer la linea de los parentesis
15       int n = s.size();
16       bool balanced = true;
17       stack <char> p;  // Stack para insertar los caracteres
18
```

Implementación II

```
19          for (int i = 0; i < n and balanced; ++i){
20              // Si es uno que abre insertarlo al stack
21              if (s[i] == '(' or s[i] == '['){
22                  p.push(s[i]);
23              }else{ // Es uno que cierra
24                  if (p.empty()){ // Si no hay elementos en el stack
25                      balanced = false;
26                      break;
27                  }
28                  // Veificar que sean del mismo tipo
29                  if (type(s[i]) == type(p.top())){
30                      p.pop();
31                  }else{
32                      balanced = false;
33                      break;
34                  }
35              }
36          }
37
```

Implementación III

```
38          // Verificar que no haya elementos en el stack
39          if (!p.empty()) balanced = false;
40
41          if (balanced) puts("Yes");
42          else puts("No");
43
44      }
45       return 0;
46  }
```

Problema C - Dominos

Solución

- Construir el grafo
- Hallar las componentes fuertemente conexas
- Hallar cuántas aristas entran a cada componente
- Retornar el número de componentes a las cuales no entra ninguna arista

Para la implementación ver las diapositivas de la semana pasada

Problema D - Word Transformation

- Pensar en que cada palabra es un nodo.
- Utilizar un mapa y darle a cada palabra un número empezando en 0. Ese número será el número del nodo.
- Dos nodos se conectan cuando difieren sólo en una letra (y tienen el mismo tamaño).
- Construir el mapa, el grafo y hacer BFS para hallar el mínimo número de intercambios

Implementación I

```
1  const int MAXN = 205;  // El maximo numero de nodos
2  map <string, int> m;   // El mapa de palabra a numero
3  string dic [MAXN];     // La lista de las palabras
4  vector <int> g[MAXN];  // El grafo dado como enteros
5  int d[MAXN];           // El arreglo de distancias para el BFS
6
7  int bfs(int s, int t){
8     for (int i = 0; i < MAXN; ++i) d[i] = -1;
9     queue <int> q;
10    q.push(s); d[s] = 0;
11    while (q.size() > 0){
12       int cur = q.front(); q.pop();
13       if (cur == t) break; // Dejar de buscar si llego a t
14       for (int i = 0; i < g[cur].size(); ++i){
15          int next = g[cur][i];
16          if (d[next] == -1){
17             q.push(next); d[next] = d[cur] + 1;
18          }
```

Implementación II

```
19          }
20      }
21      return d[t];
22  }
23
24  int main(){
25      int cases; cin >> cases;
26      while (cases--){
27          m.clear();
28          for (int i = 0; i < MAXN; ++i) g[i].clear();
29          // Agregar las palabras a la lista y al mapa
30          int total_words = 0;
31          string word;
32          while (cin >> word){
33              if (word == "*") break;
34              dic[total_words] = word;
35              m[word] = total_words;
36              total_words++;
37          }
```

Implementación III

```
38          // Construir el grafo
39          for (int i = 0; i < total_words; ++i){
40              string s1 = dic[i];
41              for (int j = i+1; j < total_words; ++j){
42                  string s2 = dic[j];
43                  if (s1.size() != s2.size()) continue;
44                  int diff = 0;
45                  // Contar el numero de letras diferentes
46                  for (int k = 0; k < s1.size(); ++k){
47                      if (s1[k] != s2[k]) diff++;
48                  }
49                  // Si solo difieren en una letra, unirlos
50                  if (diff == 1){
51                      g[m[s1]].push_back(m[s2]);
52                      g[m[s2]].push_back(m[s1]);
53                  }
54              }
55          }
56
```

Implementación IV

```
57          // Leer las palabras por las que preguntan
58          string line;
59          getline(cin, line);
60          while (getline(cin, line)){
61              if (line == "") break;
62              stringstream ss(line);
63              string s1, s2;
64              ss >> s1 >> s2;
65              // Hallar el nmero de las palabras usando el mapa
66              int s = m[s1];
67              int t = m[s2];
68              // Hacer el bfs desde s hasta t
69              cout << s1 << " " << s2 << " " << bfs(s, t) << endl;
70          }
71          if (cases != 0) cout << endl;
72      }
73      return 0;
74  }
```

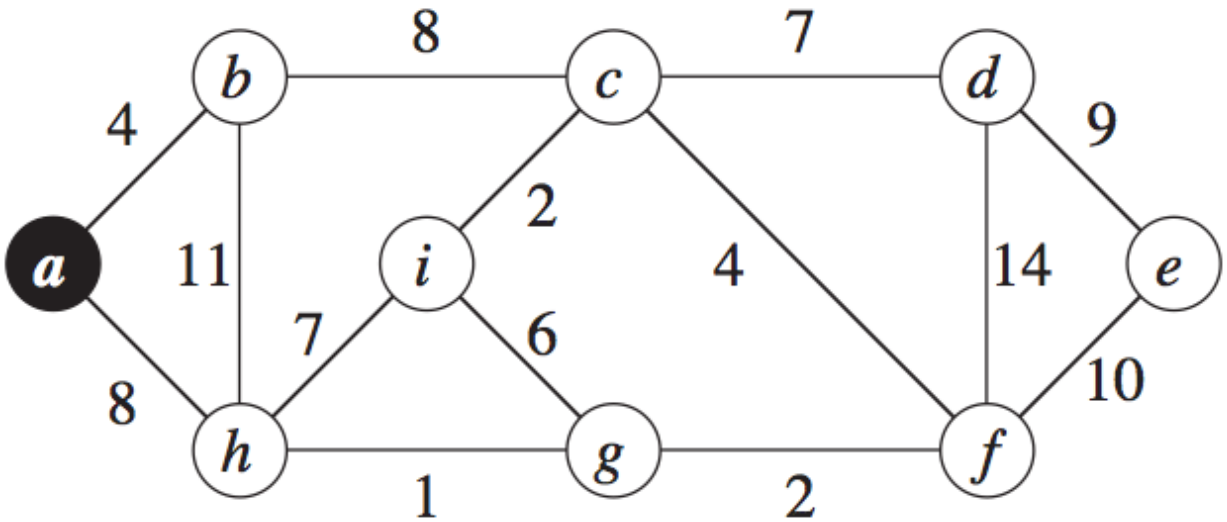
Contenido

2 Grafos con pesos

Grafos con pesos

Weighted graph

Un grafo con pesos (weighted graph) es un grafo cuyas aristas tienen asociado un peso. Si el peso de la arista $(u, v) \in E$ es w entonces ir de u a v tiene un costo w.



Representación con la lista de adyacencia

En la representación como lista de adyacencia se tiene un arreglo donde cada posición i es una lista con los nodos con los que se conecta el nodo i.
Cuando el grafo tiene pesos debo almacenar también el peso de esa conexión.
En cada posición del vector de $g[u]$ hay que tener entonces una pareja (v, w) que indica que del nodo u se va al nodo v con peso w.

Pair

Pair

Pair es un tipo de dato que permite almacenar una pareja de dos objetos que pueden ser de igual o de diferente tipo.
Declaración:

```
pair <tipo_de_dato1, tipo_de_dato2> nombre_pareja;
```

Constructor y acceso:

```
pair <int, int> p = make_pair(3, 5);
cout << p.first << " ," << p.second << endl;
// p.first es 3 y p.second es 5
```

Mayor información en:
`http://www.cplusplus.com/reference/utility/pair/`

Lista de adyacencia usando pair I

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4
5  typedef pair <int, int> edge; // Llamar edge a un pair <int, int>
6  const int MAXN = 105;
7  vector <edge> g[MAXN];
8
9  int main(){
10    int n, m;  cin >> n >> m;
11
12    for (int i = 0; i < m; ++i){
13       int u, v, w;
14       cin >> u >> v >> w;
15       // Agregar la arista de u a v con peso w
16       g[u].push_back(edge(v, w));
17    }
18
```

Lista de adyacencia usando pair II

```
19      for (int u = 0; u < n; ++u){
20          for (int i = 0; i < g[u].size(); ++i){
21              // Acceder al primer elemento de la arista
22              int v = g[u][i].first;
23              // Acceder al segundo elemento de la arista
24              int w = g[u][i].second;
25              printf("Edge from %d to %d with weight %d\n", u, v, w);
26          }
27      }
28      return 0;
29  }
```

Contenido

3 Algoritmo de Dijkstra

El camino más corto

El camino más corto

Dado un grafo $G = (V, E)$ y una función de pesos $w : E \to \mathbb{R}$
El peso $w(p)$ de un camino $p = (v_0, v_1, \ldots, v_k)$ es

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

El peso se un camino más corto $\delta(u, v)$ se define como

$$\delta(u, v) = \begin{cases} \infty & \text{si no hay ningún camino de } u \text{ a } v \\ \min \{w(p) : p \text{ es un camino de } u \text{ a } v\} & \text{en otro caso} \end{cases}$$

Un camino más corto de u a v se define como cualquier camino p tal que $w(p) = \delta(u, v)$

Single-source shortest-paths (SSSP)

Single-source shortest-paths

Dado un grafo $G = (V, E)$ y y una función de pesos $w : E \to \mathbb{R}$, el single-source shortest-paths problem es un problema que consiste en hallar el camino más corto de una fuente $s \in V$ a todos los demás nodos $v \in V$.

Algoritmos

El **BFS** resuelve este problema cuando los pesos de todas las aristas son 1.
El algoritmo de **Dijkstra** resuelve el problema cuando los pesos de las aristas son no negativos.
El algoritmo de **Bellman-Ford** resuelve el problema para el caso genérico.

Algoritmo de Dijkstra

Dijkstra

Dado un grafo $G = (V, E)$, el algoritmo de Dijkstra halla el camino más corto desde una fuente $s \in V$ a todos los nodos $v \in V$ cuando los pesos de las aristas son **no negativos**. Estes es el algoritmo **más rápido** conocido para resolver el problema de SSSP para grafos arbitrarios con pesos no negativos.

Algoritmo de Dijkstra

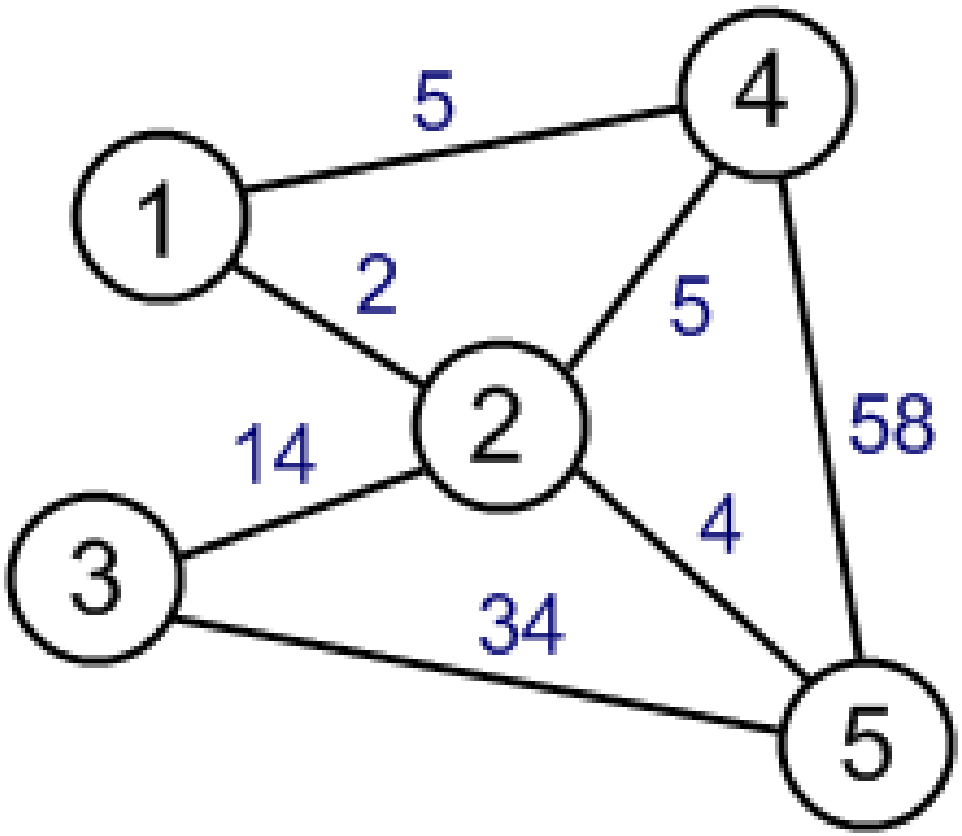
Sean s el nodo fuente

$d[v]$ la distancia del nodo s al nodo v

$p[v]$ el nodo predecesor a v en el camino más corto de s a v.

```
1  Hacer d[s] = 0 y d[v] = INF para todos los demas nodos
2  Hacer p[v] = -1 para todos los nodos
3  Agregar a la lista de nodos pendientes el nodo s con distancia 0
4  Mientras que haya nodos en la lista de pendientes
5     Extraer el nodo con la menor distancia (llamemoslo cur)
6     Recorrer cada vecino de cur (llamemoslo next)
7        Sea w_extra el peso de la arista de cur a next
8        Si d[next] > d[cur] + w_extra
9           d[next] = d[cur] + w_extra
10          p[next] = cur
11          Agregar next con d[next] a la lista de nodos pendientes
```

Ejemplo



Optimización del algoritmo

Pregunta

Repetidas veces se está buscando y extrayendo el menor elemento de la lista de pendientes.
¿Hay alguna estructura de datos que permita hacer esto rápidamente?

Optimización del algoritmo

Pregunta

Repetidas veces se está buscando y extrayendo el menor elemento de la lista de pendientes.
¿Hay alguna estructura de datos que permita hacer esto rápidamente?

Respuesta

El heap con un ordenamiento de mayor permite extraer el menor elemento en tiempo logarítmico.
La lista de pendientes se puede ver entonces como una cola de prioridades con ordenamiento de mayor.

Implementación I

```
1  typedef pair <int, int> dist_node; // Tipo de dato para el heap
2  typedef pair <int, int> edge;      // Arista = pareja de enteros
3  const int MAXN = 100005;           // El maximo numero de nodos
4  const int INF = 1 << 30;           // Infinito = 2^30
5  vector <edge> g[MAXN];             // g[u] = (v, w)
6  int d[MAXN];    // d[u] La distancia mas corta de s a u
7  int p[MAXN];    // p[u] El predecesor de u en el camino mas corto
8
9  // La funcion recibe la fuente s y el numero total de nodos n
10 void dijkstra(int s, int n){
11    // Limpiar los valores de las variables
12    for (int i = 0; i <= n; ++i){
13       d[i] = INF;  p[i] = -1;
14    }
15    // Construir la cola de prioridades con la funcion mayor
16    priority_queue < dist_node, vector <dist_node>,
17                     greater<dist_node> > q;
18    d[s] = 0;
```

Implementación II

```
19      q.push(dist_node(0, s));
20      while (!q.empty()){
21          int dist = q.top().first;
22          int cur = q.top().second;
23          q.pop();
24          // No volver a procesar el nodo
25          if (dist > d[cur]) continue;
26          for (int i = 0; i < g[cur].size(); ++i){
27              int next = g[cur][i].first;
28              int w_extra = g[cur][i].second;
29              if (d[cur] + w_extra < d[next]){
30                  d[next] = d[cur] + w_extra;
31                  p[next] = cur;
32                  q.push(dist_node(d[next], next));
33              }
34          }
35      }
36  }
```

Detalles de la implementación

- El constructor de la cola de prioridades recibe primero el tipo de dato, luego el contenedor que es por defecto un vector y finalmente la función para comparar que es por defecto menor.
- Si se quiere que la cola de prioridades extraiga el mínimo hay que poner el comparador como el máximo
- El comparador del tipo pair ordena primero por el primer elemento y si hay empate ordena por el segundo.
- A la cola de prioridades se ingresa primero la distancia para que el comparador ordene por la primera componente y se extraiga el valor menor.
- La línea `if (dist > d[cur]) continue;` es muy importante para reducir la complejidad. Con ella sólo se procesa el nodo `cur` una sola vez ya que este puede estar varias veces en la cola de prioridades.

Complejidad

Preguntas

- ¿Cuántas veces se visita (procesan sus vecinos) cada nodo?
- ¿Cuántas veces se mira cada arista?
- ¿Cuántos elementos hay en la cola de prioridades?

Complejidad

Preguntas

- ¿Cuántas veces se visita (procesan sus vecinos) cada nodo?
- ¿Cuántas veces se mira cada arista?
- ¿Cuántos elementos hay en la cola de prioridades?

Complejidad

El algoritmo de Dijkstra implementado con una lista de adyacencia y utilizando una cola de prioridades tiene complejidad $O((V + E) \log V)$

¿Por qué funciona?

- Cuando se saca un nodo de la cola de prioridades por primera vez, la distancia que tiene es la distancia del camino más corto desde la fuente.
- El valor que queda guardado en d[u] es la mínima distancia de s a u.
- El valor que queda guardado en p[u] es el predecesor de u en el camino más corto de s a u.

¿Cómo reconstruir el camino más corto?

¿Cómo se pueden hallar los nodos del camino más corto hasta t usando el arreglo p?

¿Cómo reconstruir el camino más corto?

¿Cómo se pueden hallar los nodos del camino más corto hasta t usando el arreglo p?

```
1  vector <int> find_path (int t){
2     vector <int> path;
3     int cur = t;
4     while(cur != -1){
5        path.push_back(cur);
6        cur = p[cur];
7     }
8     reverse(path.begin(), path.end());
9     return path;
10 }
```

Contenido

4 Tarea

Tarea

Tarea

Resolver los problemas de
`http://contests.factorcomun.org/contests/52`

Problema opcional

Opcional resolver el problema
`http://codeforces.com/problemset/problem/20/C`
Para este problema, la distancia debe ser un long long (puede ser mayor de 10^9)
Usar `INF = 1LL << 60` (2^{60}), el 1LL es para especificar que es un long long.