Resumen de algoritmos para maratones de programación

Universidad Tecnológica de Pereira

5 de octubre de 2012

Índice

1. Plantilla

```
#include <cstdio>
#include <cmath>
#include <functional>
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <list>
#include <map>

using namespace std;

#define all(x) x.begin(),x.end()
#define rep(i,a,b) for(int i=a;i<b;i++)
#define REP(i,n) rep(i,0,n)
#define foreach(x, v) for (typeof (v).begin() x = (v).begin(); \
x != (v).end(); ++x)
#define D(x) cout << #x " = " << x << endl;

typedef long long int lld;
typedef pair<int,int> pii;
typedef vector<int> vi;
typedef vector<pii> vpii;

int main(){

  return 0;
}
```

.....

2. Grafos

2.1. Depth First Search

Es un algoritmo para recorrer o buscar en un grafo. Empieza visitando la raiz y luego explora a fondo cada rama antes de hacer el backtraing.

Complejidad $O(|V| + |E|)$

```
/*
recordar
#define all(x) x.begin(),x.end()
typedef vector<int> vi;
typedef vector<vi> vvi;
*/
int N; // Número de vertices.
vvi G; // Grafo.
vi visited; // Vector de visitados.

void dfs(int i) {
  if(!visited[i]) {
    visited[i] = true;
    for_each(all(G[i]), dfs);
  }
}

bool check_graph_connected_dfs() {
  int start_vertex = 0;
  visited = vi(N, false);
  dfs(start_vertex);
  return (find(all(V), 0) == V.end());
}
```

.....

2.2. Breadth First Search

Es un algoritmo de búsqueda que empieza en la raíz y explora todos los nodos vecinos. Luego para cada vecino repite el proceso hasta encontrar la meta.
Complejidad $O(|V| + |E|)$

```
/*
```

```
Se considera un Grafo NO dirigido.
*/

int N; // number of vertices
vvi G; // lists of adjacent vertices


void bfs(){
  int start_vertex = 0;
  vi V(N, false);
  queue<int> Q;
  Q.push(start_vertex);
  V[start_vertex] = true;
  while(!Q.empty()) {
    int i = Q.front();
    Q.pop();
    foreach(G[i], it) {
      if(!V[*it]) {
        V[*it] = true;
        Q.push(*it);
      }
    }
  }
}
```

.....

2.3. Shortest path problem

El shortest path problem es el problema de encontrar el camino mas corto entre dos nodos de un grafo. Existen dos algoritmos para solucionarlo: Dijkstra y Bellman-Ford. Si bien Dijkstra tiene una complejidad mejor que Bellman-Ford, no funciona para grafos con aristas con peso negativo mientras que Bellman-Ford si lo hace.

2.3.1. Dijkstra

Calcula la ruta más corta a todos los nodos desde un origen.
Todas las aristas deben ser NO negativas, en ese caso usar Bellman-ford.
Complejidad $O(E\ logV)$

```
/***
* Recordar, typedef pair<int,int> pii;
* typedef vector<pii> vii;
* typedef vector<vii> vvii;
***/

vi D (N, 987654321);
vi P (N, -1);
// D[i] es la distancia desde el vértice inicial hasta i
// P[i] es el predecesor de i en la ruta más corta
priority_queue < pii, vector < pii >, greater < pii > >Q;
//Cola de prioridad con comparador Mayor,
//top() retorna la distancia más lejana.
//Inicializar el vértice inicial.
D[0] = 0;
Q.push (pii (0, 0));
while (!Q.empty ()) {
  pii top = Q.top ();
  Q.pop ();
  int v = top.second, d = top.first;
  if (d <= D[v]) {
    foreach(G[v], it) {
      int v2 = it->first, cost = it->second;
      if (D[v2] > D[v] + cost) {
        D[v2] = D[v] + cost;
        P[v2] = v;
        Q.push (pii (D[v2], v2));
      }
    }
  }
}

//Código basado en TopCoder Algorithms tutorials.
```

.....

2.3.2. Bellman-Ford

The Bellman–Ford algorithm computes single-source shortest paths in a weighted digraph. For graphs with only non-negative edge weights, the faster Dijkstra's algorithm also solves the problem. Thus, Bellman–Ford is used primarily for graphs with negative edge weights. The algorithm is named after its developers, Richard Bellman and Lester Ford, Jr.

If a graph contains a "negative cycle", i.e., a cycle whose edges sum to a negative value, then walks of arbitrarily low weight can be constructed, i.e., there can be no shortest path. Bellman-Ford can detect negative cycles and report their existence, but it cannot produce a correct answer if a negative cycle is reachable from the source.

Para solucionar el longest path, simplemente se aplica bellman-ford con los pesos de los caminos negativos.

```
procedure BellmanFord(list vertices, list edges, vertex source)
   // This implementation takes in a graph, represented as lists of vertices
   // and edges, and modifies the vertices so that their distance and
   // predecessor attributes store the shortest paths.
   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then v.distance := 0
       else v.distance := infinity
       v.predecessor := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge uv in edges: // uv is the edge from u to v
           u := uv.source
           v := uv.destination
           if u.distance + uv.weight < v.distance:
               v.distance := u.distance + uv.weight
               v.predecessor := u

   // Step 3: check for negative-weight cycles
   for each edge uv in edges:
       u := uv.source
       v := uv.destination
       if u.distance + uv.weight < v.distance:
           error "Graph contains a negative-weight cycle"
```

2.4. Shortest pair of edge disjoint paths

Edge disjoint shortest pair algorithm is an algorithm in computer network routing. The algorithm is used for generating the shortest pair of edge disjoint paths between a given pair of vertices as follows:

- Run the shortest pair algorithm for the given pair of vertices

- Replace each edge of the shortest path (equivalent to two oppositely directed arcs) by a single arc directed towards the source vertex

- Make the length of each of the above arcs negative

- Run the shortest path algorithm (Note: the algorithm should accept negative costs)

- Erase the overlapping edges of the two paths found, and reverse the direction of the remaining arcs on the first shortest path such that each arc on it is directed towards the sink vertex now. The desired pair of paths results.

2.5. All-pairs shortest paths

Es una versión del shortest path problem donde hay que hayar la menor distancia entre todos los nodos.

Existen dos algoritmos para solucionarlo, Floyd-Warshall y Jhonson. Floyd-Warshall lo hace en tiempo cubico, mientras que Jhonson en VxE, por lo que solo es util si E es asimptoticamente menor que VxV (es decir, el grafo no es muy denso).

2.5.1. Floyd-Warshall

In computer science, the Floyd–Warshall algorithm (sometimes known as the WFI Algorithm or Roy–Floyd algorithm) is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices. The algorithm is an example of dynamic programming.

Pseudocodigo:

Normal

/* Assume a function edgeCost(i,j) which returns the cost of the edge from i to j (infinity if there is none). Also assume that n is the number of vertices and edgeCost(i,i) = 0 */

int path[][]; /* A 2-dimensional matrix. At each step in the algorithm, path[i][j] is the shortest path from i to j using intermediate vertices $(1..k-1)$. Each path[i][j] is initialized to edgeCost(i,j) or infinity if there is no edge between i and j. */

procedure FloydWarshall () for k := 1 to n for i := 1 to n for j := 1 to n path[i][j] = min (path[i][j], path[i][k]+path[k][j]);

Con reconstrucción de path

procedure FloydWarshallWithPathReconstruction () for k := 1 to n for i := 1 to n for j := 1 to n if path[i][k] + path[k][j] ¡path[i][j] then path[i][j] := path[i][k]+path[k][j]; next[i][j] := k;

procedure GetPath (i,j) if path[i][j] equals infinity then return "no path"; int intermediate := next[i][j]; if intermediate equals 'null' then return ; /* there is an edge from i to j, with no vertices between */ else return GetPath(i,intermediate) + intermediate + GetPath(intermediate,j);

2.5.2. Johnson

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the Bellman–Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph.

Pseudocodigo:

Johnson's algorithm consists of the following steps:

First, a new node q is added to the graph, connected by zero-weight edges to each other node.

Second, the Bellman–Ford algorithm is used, starting from the new vertex q, to find for each vertex v the least weight h(v) of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.

Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from u to v, having lengthw(u,v), is given the new length $w(u, v) + h(u) - h(v)$.

Finally, q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.

In the reweighted graph, all paths between a pair s and t of nodes have the same quantity $h(s) - h(t)$ added to them, so a path that is shortest in the original graph remains shortest in the modified graph and vice versa. However, due to the way the values h(v) were computed, all modified edge lengths are non-negative, ensuring the optimality of the paths found by Dijkstra's algorithm. The distances in the original graph may be calculated from the distances calculated by Dijkstra's algorithm in the reweighted graph by reversing the reweighting transformation.

The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is O(V2log V + VE): the algorithm uses O(VE) time for the Bellman–Ford stage of the algorithm, and $O(VlogV + E)$ for each of V instantiations of Dijkstra's algorithm. Thus, when the graph is sparse, the total time can be faster than the Floyd–Warshall algorithm, which solves the same problem in time $O(V3)$.

2.6. Shortest pair of completely disjoint paths

In theoretical computer science and network routing, Suurballe's algorithm is an algorithm for finding two disjoint paths in a nonnegatively-weighted directed graph, so that both paths connect the same pair of vertices and have minimum total length. The algorithm was conceived by J. W. Suurballe and published in 1974. The main idea of Surballe's algorithm is to use Dijkstra's algorithm to find one path, to modify the weights of the graph edges, and then to run Dijkstra's algorithm a second time. The modification to the weights is similar to the weight modification in Johnson's algorithm, and preserves the non-negativity of the weights while allowing the second instance of Dijkstra's algorithm to find the correct second path.

Suurballe's algorithm performs the following steps:

Find the shortest path tree T rooted at node s by running Dijkstra's algorithm. This tree contains for every vertex u, a shortest path from s to u. Let P1 be the shortest cost path froms to t. The edges in T are called tree edges and the remaining edges are called non tree edges.

Modify the cost of each edge in the graph by replacing the cost w(u,v) of every edge (u,v) by $w'(u, v) = w(u, v) - d(s, v) + d(s, u)$. According to the resulting modified cost function, all tree edges have a cost of 0, and non tree edges have a non negative cost.

Create a residual graph Gt formed from G by removing the edges of G that are directed into s and by reversing the direction of the zero length edges along path P1.

Find the shortest path P2 in the residual graph Gt by running Dijkstra's algorithm.

Discard the reversed edges of P2 from both paths. The remaining edges of P1 and P2 form a subgraph with two outgoing edges at s, two incoming edges at t, and one incoming and one outgoing edge at each remaining vertex. Therefore, this subgraph consists of two edge-disjoint paths from s to t and possibly some additional (zero-length) cycles. Return the two disjoint paths from the subgraph.

Colección de códigos útiles para aritmética modular

3.2. Mayor exponente de un primo que divide a n!

```
int pow_div_fact(int n, int p) {
  int sd = 0;
  for (int t = n; t > 0; t /= p)
    sd += t % p;
  return (n-sd)/(p-1);
}
```

```
typedef unsigned int uint;

//retorna el siguiente entero con la mimsma
//cantidad de 1's en la representación binaria

uint next_popcount(uint n){
  uint c = (n & -n);
  uint r = n+c;
  return (((r ^ n) >> 2) / c) | r;
}

//retorna el primer entero con n 1's en binario
uint init_popcount(int n){
  return (1 << n) - 1;
}
```