Apuntes OIFem 2020

Grafos 2

Algoritmo de Dijkstra

El algoritmo de Dijkstra sirve para encontrar, partiendo desde un nodo de salida u, la distancia más corta entre u y el resto de nodos de un grafo. La versión que usaremos es la explicada en el libro **Competitive Programming 3**, basada en las rutas alternativas.

Empezaremos teniendo en cuenta que la distancia de u a u es 0. Tendremos una cola de prioridad en todo momento, donde guardaremos rutas alternativas. Las rutas alternativas consisten en formas más rápidas de llegar a un nodo desde la salida. Inicialmente, la única ruta alternativa descubierta es la de u a u con coste 0.

Mientras la cola no esté vacía, extraemos la pareja de un nodo i y su distancia d de la cola, usando negativos para volverla una cola que selecciona por mínimos y no máximos, y esparcimos el conocimiento de esta nueva ruta a las conexiones directas del nodo i, ya que puede serles útil. Por lo tanto, vamos nodo a nodo comprobando si la distancia entre u e i más el coste de la arista es inferior al récord actual de distancia entre la raíz y j. Si este es el caso, hemos descubierto una ruta alternativa para llegar de la raíz u al nodo j a través de i y añadimos la pareja de j y este nuevo récord a la cola para que llegue a las conexiones de j.

Puede darse el caso de que haya varias rutas alternativas para llegar al mismo nodo en la cola, ya que antes de explorar una ruta de distancia a de la raíz podremos haber encontrado una más corta de distancia b, donde $b < a$. Por lo tanto, para cada pareja que extraeremos de la cola, comprobaremos que efectivamente la distancia recorrida a través de esta ruta es el mínimo actual. A causa de que pueda haber estas duplicidades en la cola, no podremos retornar la primera ruta alternativa que encontremos al destino, ya que puede haberlas más cortas.

Una vez se quede vacía la cola, sabremos que hemos encontrado el camino más corto para llegar al resto de nodos (alcanzables) del grafo.

Extra: esta versión es de las pocas versiones que funciona con aristas con coste negativo PERO sin ciclos de coste negativo (estos hacen que siempre haya una ruta alternativa en la cola y que nunca se termine la función). Estos son ciclos donde la suma de los costes es negativa y, por tanto, puedes recorrerlos infinitas veces para reducir el coste final. ¿Cómo harías de forma eficiente para reconocer un ciclo así? Piensa una solución antes de bajar y ver Bellman-Ford.

Extra 2: Sabiendo las complejidades de priority queue, ¿cuál es la complejidad de esta implementación de Dijkstra?

Código- Patinete eléctrico

```cpp
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
#include <queue>
using namespace std;

int Dijkstra(int salida, int destino, vector<vector<pair<int, int>>> & listaAdy) {
    int N = (int) listaAdy.size();
    // vector donde guardaremos las distancias récord para llegar a cada nodo
    vector<int> distancia(N, 1e9);
    // la distancia de la salida a la salida es 0
    distancia[salida] = 0;
    priority_queue<pair<int, int>> pq;
    // añadimos la única ruta conocida a la cola
    pq.push(make_pair(0, salida)); // {distancia, nodo}
    while(!pq.empty()) {
        // extraemos la ruta más corta de la cola
        int nodo = pq.top().second;
        int distanciaAlt = -pq.top().first;
        pq.pop();
        // la descartaremos si ya hemos encontrado una mejor
        if (distanciaAlt > distancia[nodo])
            continue;
        distancia[nodo] = distanciaAlt;
        for (pair<int, int> vecino: listaAdy[nodo]) {
            int nodoVecino = vecino.first;
            int costeVecino = vecino.second;
            if (distancia[nodoVecino] > distancia[nodo] + costeVecino) {
                distancia[nodoVecino] = distancia[nodo] + costeVecino;
                pq.push(make_pair(-distancia[nodoVecino], nodoVecino));
            }
        }
    }
    return distancia[destino];
}


int main() {
    int T, N, M, a, b, c;
    vector<vector<pair<int, int>>> listaAdy;
    cin >> T;
    while(T--) {
        cin >> N >> M;
        listaAdy.assign(N, vector<pair<int, int>>());
        while(M--) {
            cin >> a >> b >> c;
            listaAdy[a].push_back(make_pair(b, c));
            listaAdy[b].push_back(make_pair(a, c));
        }
        cout << Dijkstra(0, 1, listaAdy) << '\n';
    }
```

```
    return 0;
}
```

Implementación de Dijkstra en Python

Algoritmo de Floyd-Warshall's

Este algoritmo, que funciona en $O(n^3)$, calcula el coste más corto para llegar de u a v para todas las parejas (u, v) de nodos del grafo.

Es un algoritmo, como veis, terriblemente ineficiente, pero muy sencillo de entender y rápido de programar. Con todo y con eso, es más eficiente por regla general que ejecutar Dijkstra desde todos los nodos de salida.

Empezaremos teniendo guardados en una matriz de adyacencia los costes de las aristas entre nodos. Es decir, `matrizAdyacencia[u][v]` guarda el coste de la arista más corta de u a v (en el caso de que haya varias; si solo hay una, será este el coste guardado). Además, guardará 0 en la diagonal e infinito entre todas las parejas de nodos donde no hay una arista en esa dirección.

Sabemos que, para llegar del nodo i al nodo j, hay tres posibilidades:

- No hay forma de llegar de i a j: mantenemos la distancia de infinito en la matriz de adyacencia
- La forma más corta es a través de una arista de i a j: mantenemos ese valor inicial
- La forma más corta es a través de una ruta que va del nodo i al nodo k y de k a j: encontramos dicho nodo k y guardamos la distancia final como la suma de las distancias

Como veis, en los dos primeros casos, no hay que alterar la matriz. Será solo en el tercero donde haya que hacer cambios. Probamos todos los nodos k entre 0 y $N-1$, guardando el mínimo. Así, si la distancia es infinito, se mantendrá, al igual que si es menor que a través de ningún nodo k.

Código

```
void apsp(vector<vector<int>> & matrizAdyacencia) {
        int N = (int) matrizAdyacencia.size();
        // matrizAdyacencia[u][v] = distancia de u a v si hay arista directa //
        ↪  infinito en caso contrario
        // matrizAdyacencia[i][i] = 0 para todo i
        for (int k = 0; k < N; k++)
            for (int i = 0; i < N; i++)
                for (int j = 0; j < N; j++)
                    matrizAdyacencia[i][j] = min(matrizAdyacencia[i][j],
                    ↪  matrizAdyacencia[i][k] + matrizAdyacencia[k][j]);
        // matrizAdyacencia[u][v] ahora guarda la forma más corta de llegar de u a
        ↪  v
}
```

Implementación de Floyd-Warshall's en Python

Grafos bipartitos

Los grafos bipartitos son aquellos donde podemos separar los nodos en dos conjuntos de forma que no haya una arista entre dos nodos del mismo conjunto. Esto puede modelarse como separar a los alumnos

en dos clases de forma que no haya dos amigos en la misma clase (es decir, que se separen todas las parejas de amigos). No todos los grafos pueden partirse de esta forma. Por ejemplo, si hay un trío de amigos no habrá forma de separarlos y habrá al menos una pareja de amigos en la misma clase.

La separación se hace a través de una DFS donde vamos separando a las parejas de amigos. Es decir, si estamos visitando al nodo u y este está en el equipo 0, metemos a todos los nodos v conectados a u en el equipo 1 y los visitamos, metiendo a sus conexiones en el 0. Si observamos alguna contradicción, marcaremos un flag booleano para dejar constancia de esto y dejaremos de recurrir más a fondo.

Código- CSES Building Teams

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
using namespace std;

bool esPosible;

void dfs(int nodo, vector<vector<int>> & listaAdyacencia, vector<int> & equipo) {
        // dejamos la recusión si ya sabemos que no hay solución
        if (!esPosible)
                return;
        for (int amigo: listaAdyacencia[nodo]) {
                if (equipo[amigo] == -1) {
                    // el amigo aún no tiene equipo -> le metemos en el contrario
                        equipo[amigo] = 1 - equipo[nodo];
                        dfs(amigo, listaAdyacencia, equipo); // visitamos las
                        ↪ conexiones del amigo
                } else if (equipo[amigo] == equipo[nodo]) {
                        // ¡imposible! Hay una contradicción en la lista de
                        ↪ adyacencia -> paramos la construcción de los equipos
                        esPosible = false;
                        return;
                }
        }
}

void partirClase(vector<vector<int>> & listaAdyacencia) {
        int N = (int) listaAdyacencia.size();
        // Inicialmente, no hay ningún nodo en un equipo
        vector<int> equipo(N, -1);
        esPosible = true; // la división es posible hasta que se demuestre lo
        ↪ contrario
        for (int i = 0; i < N; i++) {
                if (equipo[i] == -1) {
                    // aún no hemos visitado la componente conexa de i
                        equipo[i] = 0; // le asignamos al equipo 0 arbitrariamente
                        dfs(i, listaAdyacencia, equipo); // metemos a sus
                        ↪ conexiones en el equipo contrario y a las conexiones
                        ↪ de estas conexiones en el suyo...
                        if (!esPosible)
                                break;
                }
        }
```

```
        // Imprimimos la solución
        if (esPosible) {
                for (int i = 0; i < N; i++)
                        if (equipo[i] == 0) equipo[i] = 2;
                cout << equipo[0];
                for (int i = 1; i < N; i++)
                        cout << " " << equipo[i];
                cout << '\n';
        } else cout << "IMPOSSIBLE\n";
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int N, M, a, b;
    cin >> N >> M;
    vector<vector<int>> listaAdyacencia(N);
    while(M--) {
        cin >> a >> b;
        a--;
        b--;
        listaAdyacencia[a].emplace_back(b);
        listaAdyacencia[b].emplace_back(a);
    }
    partirClase(listaAdyacencia);
    return 0;
}
```

Implementación de la construcción de un grafo bipartito en Python

Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford sirve para encontrar ciclos de coste negativo en un grafo, es decir, que sumando los costes de las aristas del ciclo encontramos un total negativo.

Es bastante lento pese a tener un solo nodo de salida (y, por tanto, solo encuentra ciclos de coste negativo alcanzables desde este nodo de salida). Podemos repetirlo partiendo de todos los nodos si queremos asegurarnos de que no haya ciclos.

Inicializamos, como con Dijkstra, un vector de distancias infinitas para todos los nodos menos la raíz (o nodo de salida). Bellman-Ford se basa en el concepto de "relajar aristas". Es decir, que si por la ruta actual más corta al nodo u y la arista $u - v$ encontramos una ruta más corta que el récord actual a v (por lo tanto un nuevo récord), haremos que esta sea el nuevo récord, "relajando" la arista $u - v$.

Si, tras relajar todas las aristas $N - 1$ veces, lo hacemos una vez más y encontramos algún nuevo récord, esto significa que hay un ciclo negativo y que podremos estar relajando aristas para siempre.

La implementación sugerida retorna verdadero si hay un ciclo negativo y falso si no lo hay.

Código

```
bool bellmanFord(int raiz, vector<vector<pair<int, int>>> & listaAdyacencia) {
    // las parejas de la lista son {nodo, coste}
    int N = (int) listaAdyacencia.size();
```

```
    vector<int> distancia(N, 1e9);
    distancia[raiz] = 0;
    // relajamos costes N-1 veces
    for (int i = 0; i < N - 1; i++)
        for (int u = 0; u < N; u++)
            for (pair<int, int> vecino: listaAdyacencia[u]) {
                int v = vecino.first, coste = vecino.second;
                distancia[v] = min(distancia[v], distancia[u] + coste);
            }
    for (int u = 0; u < N; u++)
        for (pair<int, int> vecino: listaAdyacencia[u]) {
            int v = vecino.first, coste = vecino.second;
            if(distancia[u] + coste < distancia[v])
                return true; // hemos encontrado un ciclo negativo
        }
    return false; // no hay ningún ciclo negativo
}
```

Implementación de Bellman Ford en Python