Chapter 13 Algorithms of Best Paths

Yonghui Wu
School of Computer Science, Fudan University
yhwu@fudan.edu.cn
WeChat : 13817360465

- 13.1 Warshall Algorithm and Floyed-Warshall Algorithm
- 13.2 Dijkstra’s Algorithm

Given a weighted, directed graph $G=(V, E)$, each edge is with a real-valued weight. The weight of path $P=(v_0, v_1, \ldots\ldots, v_k)$ is the sum of weights of its constituent edges:

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

The weight of the shortest-path (longest-path) from vertex u to vertex v is defined as follow.

$$\delta(u,v) = \begin{cases} \min(\max)\{w(p) \mid u \xrightarrow{p} v\} & \text{if there is a path p from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

The best path from vertex u to vertex v is defined as any path with weight $w(p)=\delta(u, v)$.

13.1 Warshall Algorithm and Floyed-Warshall Algorithm

- Warshall algorithm is used to compute the transitive closure of a relation for a graph.
- Suppose relation *R* is represented by digraph *G*. All vertices in *G* are $v_1, v_2, \ldots, v_n$. The graph *G'* for the transitive closure *t*(*R*) can be gotten from *G* as follow. If there exists a path from vertex v_i to vertex v_j in *G*, then an arc from v_i to v_j is added in *G'*. The adjacency matrix *A* for *G'* is defined as follow. If there exists a path from vertex v_i to vertex v_j, then *A*[*i*][*j*]=1, and vertex v_j is reachable from vertex v_i; otherwise *A*[*i*][*j*]=0, and vertex v_j isn't reachable from vertex v_i. That is to say, computing the transitive closure of a relation is to determine whether every pair of vertices are reachable or not. It is a problem of transitive closure for a graph.

- Suppose there is a sequence of square matrices order n $A^{(0)}$, …, $A^{(n)}$, where each element in square matrices is 0 or 1. $A^{(0)}$ is the adjacency matrix for digraph G. For $1 \le k \le n$, $A^{(k)}[i][j]=1$ represents there exists paths from v_i to v_j passing just $v_1,\ldots,v_k$, and $A^{(k)}[i][j]=0$ represents there is no such a path.

- Warshall algorithm is as follow.
- $A^{(0)}$ is the adjacency matrix for digraph G.
- for (k=1; k<=n; k++)
- for (i=1; i<=n; i++)
- for (j=1; j<=n; j++)
- $A^{(k)}[i][j]$= =($A^{(k-1)}[i][k]$ & $A^{(k-1)}[k][j]$) | $A^{(k-1)}[i][j]$;

13.1.1 Frogger

- **Source: Ulm Local 1997**
- **IDs for Online Judge: POJ 2253, ZOJ 1942, UVA 534**

- Freddy Frog is sitting on a stone in the middle of a lake. Suddenly he notices Fiona Frog who is sitting on another stone. He plans to visit her, but since the water is dirty and full of tourists' sunscreen, he wants to avoid swimming and instead reach her by jumping. Unfortunately Fiona's stone is out of his jump range. Therefore Freddy considers to use other stones as intermediate stops and reach her by a sequence of several small jumps. To execute a given sequence of jumps, a frog's jump range obviously must be at least as long as the longest jump occuring in the sequence. The frog distance (humans also call it minimax distance) between two stones therefore is defined as the minimum necessary jump range over all possible paths between the two stones.
- You are given the coordinates of Freddy's stone, Fiona's stone and all other stones in the lake. Your job is to compute the frog distance between Freddy's and Fiona's stone.

- **Input**
- The input will contain one or more test cases. The first line of each test case will contain the number of stones *n* ($2 \le n \le 200$). The next *n* lines each contain two integers x_i, y_i ($0 \le x_i, y_i \le 1000$) representing the coordinates of stone #*i*. Stone #1 is Freddy's stone, stone #2 is Fiona's stone, the other *n*-2 stones are unoccupied. There's a blank line following each test case. Input is terminated by a value of zero (0) for *n*.
- **Output**
- For each test case, print a line saying "Scenario #*x*" and a line saying "Frog Distance = *y*" where *x* is replaced by the test case number (they are numbered from 1) and *y* is replaced by the appropriate real number, printed to three decimals. Put a blank line after each test case, even after the last one.

Analysis

- Stones in the lake are represented as vertices, and relations between stones are represented as edges, weights of edges are Euclidean distances between vertices. Suppose vertex 0 is the stone where Freddy initially sits on, and vertex 1 is the stone where Fiona initially sits on. The key to the problem is to determine whether vertex 1 is reachable from vertex 0 through a path in which the length of edges isn't greater than *K*.

Suppose the weighted adjacency matrix is L, where the length of the edge connecting (x_i, y_i) and (x_j, y_j) is $L[i][j]=\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, $0 \le i, j \le n-1$. And *con* is the reachability matrix, where *con*[*i*][*j*] shows whether there is a path from vertex *i* to vertex *j* and the length of edges in the path isn't greater than *K*. Warshall algorithm can be used to compute the reachability matrix *con*, that is, after edges whose length is greater than *K* are deleted, connectivity of a graph is computed.

Initially, the reachability matrix *con* is as follow.

$$con[i][j]=\begin{cases} false & L[i,j] > K \\ true & L[i,j) \le K \end{cases} \quad (0 \le i, j \le n-1);$$

Then Warshall algorithm is used to compute the transitive closure of the graph:

```
for (int k=0; k< n; k++)                    // Enumeration of intermediate vertices
  for (int i=0; i< n; i++)      // Enumeration starting vertices and terminal vertices in paths
    for (int j=0; j<n; j++)
      con[i][j] |= con[i][k]&con[k][j];    // Determine whether there is such a path or not
```

from vertex *i* to vertex *j*.

Binary search is used to compute Frog Distance. Suppose the interval for Frog Distance is $[l, r]$. Initially $l=0$, $r=10^5$.

```
while (r-l>=10^-5){
    K=⌊(l+r)/2⌋;        // computing the intermediate value for the interval
    the reachability matrix con under the upper limit K;
    if (con[0][1])  r =K;   // Binary search
        else l =K;
}
Output Frog Distance r;
```

- Based on Warshall algorithm, Floyed-Warshall algorithm is used to find the best paths between each pair of vertices in a weighted graph. In Floyed-Warshall algorithm, Boolean operator '&' is changed into arithmetic operator '+', and boolean calculation '|' is changed into comparing $A^{(k-1)}[i][k]+A^{(k-1)}[k][j]$ with $A^{(k-1)}[i][j]$. That is, Floyed-Warshall formula is as follow.
- $A^{(0)}[i][j]$= adjacency matrix M;
- $A^{(k)}[i][j]$= min(max){ $A^{(k-1)}[i][k]+A^{(k-1)}[k][j]$, $A^{(k-1)}[i][j]$ } , where i, j, k=1..n.
- That is, $A^{(k)}[i][j]$ is the length of the best path from vertex v_i to v_j passing just $v_1,\ldots,v_k$. $A^{(n)}[i][j]$ is the length of the best paths from vertex v_i to v_j.

- Floyed-Warshall algorithm can be used to compute best-paths for all-pairs in a graph. Its time complexity is $O(n^3)$. If the shortest path is required to calculate, there must be no negative weighted circuit. And if the longest path is required to calculate, there must be no positive weighted circuit. Otherwise it will lead to endless loop.

13.1.2 Arbitrage

- **Source: Ulm Local 1996**
- **IDs for Online Judge: POJ 2240, ZOJ 1092, UVA 436**

- Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 US Dollar buys 0.5 British pound, 1 British pound buys 10.0 French francs, and 1 French franc buys 0.21 US dollar. Then, by converting currencies, a clever trader can start with 1 US dollar and buy 0.5 * 10.0 * 0.21 = 1.05 US dollars, making a profit of 5 percent.
- Your job is to write a program that takes a list of currency exchange rates as input and then determines whether arbitrage is possible or not.

- **Input**
- The input will contain one or more test cases. Om the first line of each test case there is an integer n ($1 \leq n \leq 30$), representing the number of different currencies. The next n lines each contain the name of one currency. Within a name no spaces will appear. The next line contains one integer m, representing the length of the table to follow. The last m lines each contain the name c_i of a source currency, a real number r_{ij} which represents the exchange rate from c_i to c_j and a name c_j of the destination currency. Exchanges which do not appear in the table are impossible.
- Test cases are separated from each other by a blank line. Input is terminated by a value of zero (0) for n.
- **Output**
- For each test case, print one line telling whether arbitrage is possible or not in the format "Case case: Yes" respectively "Case case: No".

Analysis

- Currency exchange is represented as a weight directed graph. A currency is represented as a vertex. A currency exchange is represented as an arc. Vertex i represents the i-th currency c_i, arc (i, j) represents currency exchange from c_i to c_j, and its weight r_{ij} represents the exchange rate from c_i to c_j, $1 \le i, j \le n$. Suppose $dist[i][j]$ is the exchange rate from c_i to c_j. If currency c_i can be changed into currency c_k, and currency c_k can be changed into currency c_j, then the exchange rate from c_i to c_j through c_k is $dist[i][k]* dist[k][j]$. It can be regarded as the length of the path from vertex i to vertex j through vertex k. All possibilities of currency exchanges should be enumerated. Therefore the problem is to compute the all-pairs' longest paths in a graph. In the graph there are circuits. In order to avoid endless loop, there must be a restriction ($i!=j\&\&j!=k\&\&k!=i$). The process is as follow.
- for (int k=1; k<= n; k++) // Enumerate intermedia vertex k
- for (int i=1; i<= n; i++) //Enumerate all pairs (i, j)
- for (int j=1; j<= n; j++)
- if ($i!=j\&\&j!=k\&\&k!=i$)
- if ($dist[i][k]*dist[k][j]>dist[i][j]$)
- $dist[i][j]= dist[i][k]*dist[k][j]$;

- Then all currencies pairs are enumerated. If there exists such a currency *i*, by converting currencies currency *i* → currency j_1 → …… → currency j_m → currency *i* ($m \geq 1$), and a profit can be made, then arbitrage is possible. If there is no such a currency, arbitrage is impossible.
- *flag* = 0; //The flag whether arbitrage is possible or not
- for (int *i*=1; *i*<= *n*; *i*++) //Enumerate all currencies
- for (int *j*=1; *j*<= *n*; *j*++) //Enumerate intermedia currencies
- if (*dist*[*i*][*j*]**dist*[*j*][*i*]>1) *flag* = 1; //If making a profit, arbitrage is possible, and solution to the current test case(*flag*?"Yes":"No").

13.2 Dijkstra's Algorithm

- Dijkstra's algorithm is used to solve the single-source shortest-paths problem in a weighted, directed graph $G(V, E)$ for the case in which all arcs' weights are nonnegative. That is, for each arc $(u, v) \in E$, $w(u, v) \geq 0$.

Dijkstra's algorithm

- void *Dijkstra*(int *r*); //Dijkstra's algorithm: shortest-paths from vertex *r* to other vertices
- { for (*i*=0; *i*<*n*; *i*++)
- *dist*[*i*]=∞;
- *dist*[*r*]=0; // the length for the shortest-paths for *r* is 0
- *S*=∅;
- *Q* is a min-priority queue used to store *n* vertices;
- while (*Q*≠∅) //if *Q* isn't empty
- { vertex *u* isn't in *S* and *dist*[*u*] is minimal;
- *S*=*S*∪{*u*}; //*u* is added into the set of vertices *S* known the shortest paths
- for (each vertex *v* not in *S*)
- if (*dist*[*u*]+w_{uv}<*dist*[*v*])
- *dist*[*v*]=*dist*[*u*]+ w_{uv};
- }
- }

- If the min-priority queue is implemented by an array, the time complexity of Dijkstra's algorithm is $O(V^2+E)\approx O(V^2)$. If the min-priority queue is implemented by a binary min-heap, the time complexity of Dijkstra's algorithm is $O((V+E)*\ln V)\approx O(E*\ln V)$. If the graph is a sparse graph, the min-priority queue implemented by a binary min-heap is suitable.