Yonghui Wu
Stony Brook University
yhwu@fudan.edu.cn

ALGORITHMS OF BEST PATHS

- 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.

$$\delta(u,v) = \begin{cases} \min(\max)\{w(p) : u \overset{p}{\rightsquigarrow} v\} & [illegible]\ u\ [illegible]\ v, \\ [illegible] & [illegible] \end{cases} \quad (1)$$

- Warshall algorithm
  - is used to get the transitive closure for a graph;
- Floyed-Warshall algorithm
  - is used to get all-pairs best paths in a graph;
- Dijkstra algorithm, Bellman-Ford algorithm and SPFA algorithm
  - are used to get single-source shortest paths in a graph.

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 $A^{(0)}, \ldots, A^{(n)}$, 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; $j$<=$n$; $j$++)
$A^{(k)}$[$i$][$j$]= =($A^{(k-1)}$[$i$][$k$] & $A^{(k-1)}$[$k$][$j$]) | $A^{(k-1)}$[$i$][$j$];
```

- Warshall algorithm can be not only used to compute the transitive closure of a graph, but also used to solve the problem that there exits length limit for arcs in a graph.

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.

- 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 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.

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≤*n*≤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".

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 is as follow.

```
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
   { u is a vertex not in S and dist[u] is minimal;
     S=S∪{u};           //u is added into the set of vertices S known the shortest paths
     for ( all vertex v not in S)
if (dist[u]+w_uv<dist[v])
 dist[v]=dist[u]+ w_uv;
   }
}
```

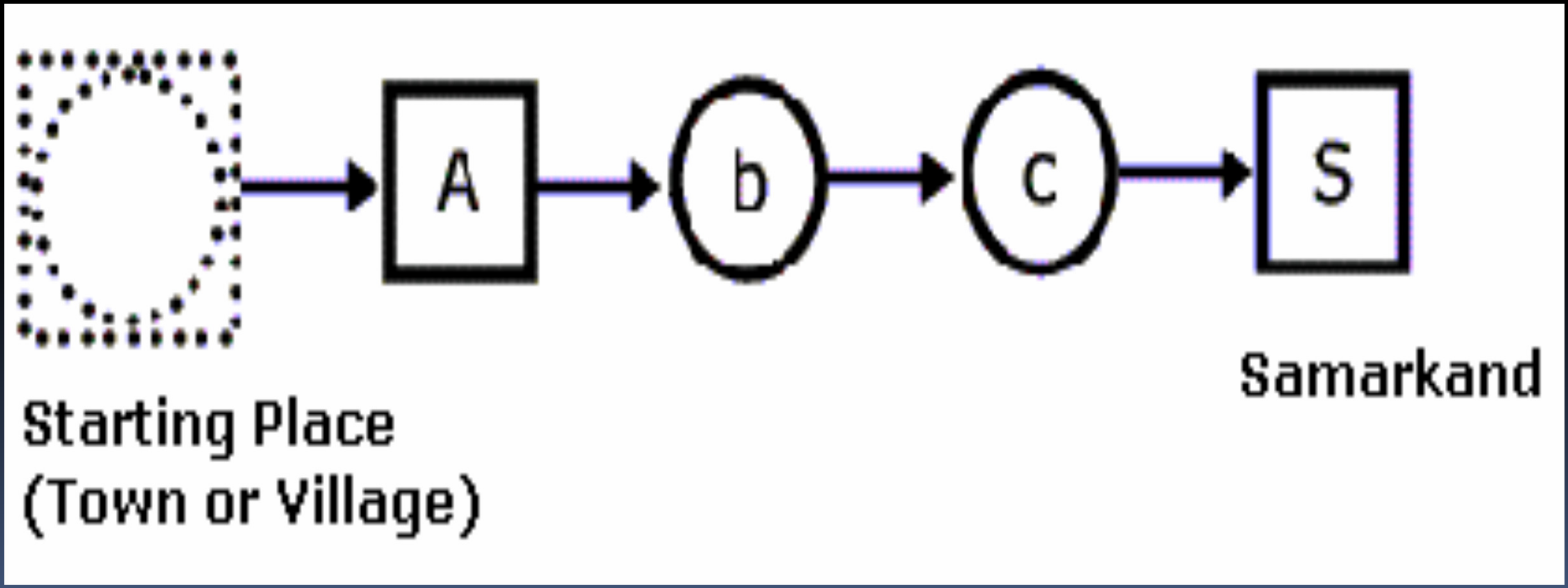
- If the min-priority queue is implemented by an array, the time complexity of Dijkstra’s algorithm is $O(V^2+E) \approx O(V^2)$. If the min-priority queue is implemented by a binary min-heap, the time complexity of Dijkstra’s algorithm is $O((V+E)*\ln V) \approx O(E*\ln V)$. If the graph is a sparse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

Toll

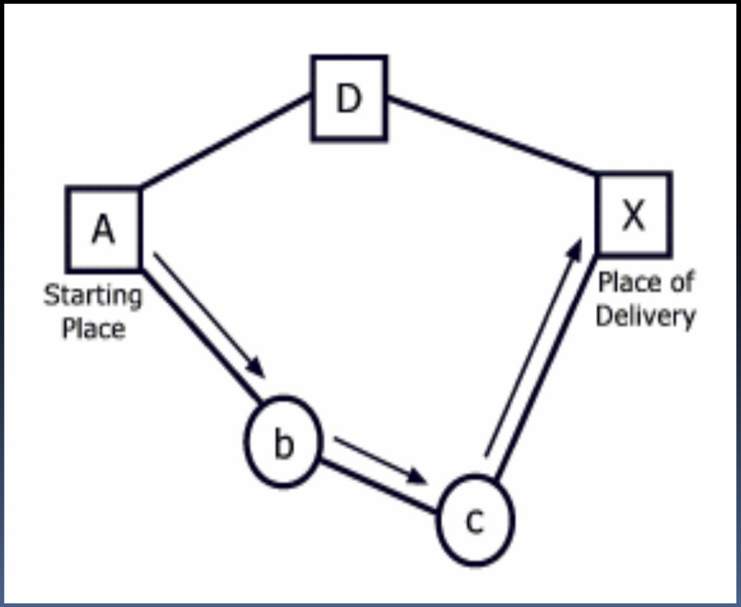
- Source: ACM World Finals - Beverly Hills - 2002/2003
- IDs for Online Judge: UVA 2730

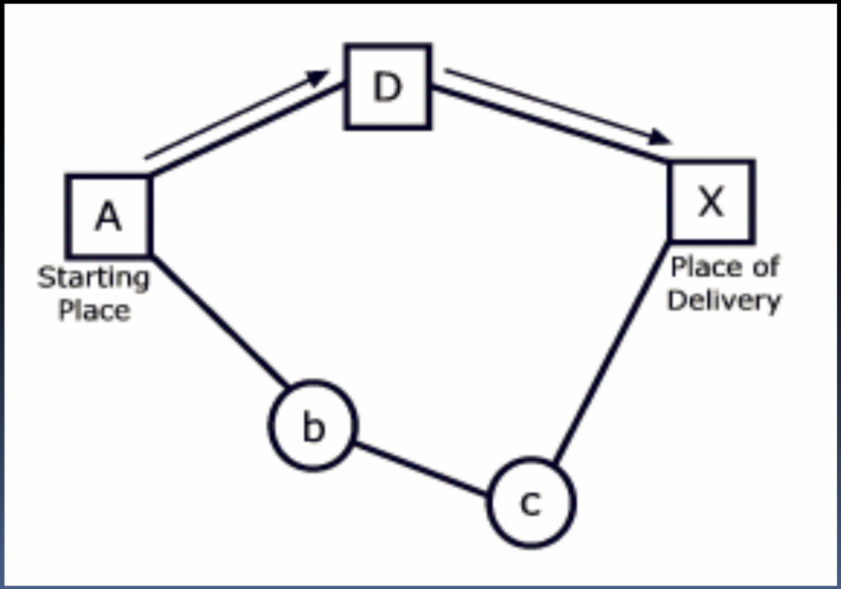
- Sindbad the Sailor sold 66 silver spoons to the Sultan of Samarkand. The selling was quite easy; but delivering was complicated. The items were transported over land, passing through several towns and villages. Each town and village demanded an entry toll. There were no tolls for leaving. The toll for entering a village was simply one item. The toll for entering a town was one piece per 20 items carried. For example, to enter a town carrying 70 items, you had to pay 4 items as toll. The towns and villages were situated strategically between rocks, swamps and rivers, so you could not avoid them.

- Figure 13.1: To reach Samarkand with 66 spoons, traveling through a town followed by two villages, you must start with 76 spoons.



- Figure 13.2 : The best route to reach X with 39 spoons, starting from A, is A→b→c→X, shown with arrows in the figure on the left. The best route to reach X with 10 spoons is A→D→X, shown in the figure on the right. The figures display towns as squares and villages as circles.





- Predicting the tolls charged in each village or town is quite simple, but finding the best route (the cheapest route) is a real challenge. The best route depends upon the number of items carried. For numbers up to 20, villages and towns charge the same. For large numbers of items, it makes sense to avoid towns and travel through more villages, as illustrated in Figure 13.2.
- You must write a program to solve Sindbad’s problem. Given the number of items to be delivered to a certain town or village and a road map, your program must determine the total number of items required at the beginning of the journey that uses a cheapest route.

- **Input**
- The input consists of several test cases. Each test case consists of two parts: the roadmap followed by the delivery details.
- The first line of the roadmap contains an integer *n*, which is the number of roads in the map ($0 \leq n$). Each of the next *n* lines contains exactly two letters representing the two endpoints of a road. A capital letter represents a town; a lower case letter represents a village. Roads can be traveled in either direction.
- Following the roadmap is a single line for the delivery details. This line consists of three things: an integer *p* ($0 < p \leq 1000$) for the number of items that must be delivered, a letter for the starting place, and a letter for the place of delivery. The roadmap is always such that the items can be delivered.
- The last test case is followed by a line containing the number -1.

- **Output**
- The output consists of a single line for each test case. Each line displays the case number and the number of items required at the beginning of the journey. Follow the output format in the example given below.

Bellman-Ford Algorithm

- Bellman-Ford algorithm is used to calculate the single-source shortest-paths in a weighted directed graph in which edges’ weights may be negative.
- Like Dijkstra’s algorithm, Bellman-Ford algorithm also uses relaxation. For each vertex $v \in V$, the estimate *dist*[*v*] on the weight of a shortest path from the source *s* to *v* is progressively decreased until it achieves the actual shortest-path weight. If there exists negative-weight cycles in the graph, Bellman-Ford algorithm should report there aren’t shortest paths.

```
Bool Bellman_Ford(int s)      //Bellman-Ford algorithm is used to compute the shortest paths from source s to other vertices
{
for ($i$=0; $i<n$; $i$++)          //Initialization
 { dist[$i$]=$\infty$; [$i$]=nil ;}
  dist[$s$]=0;
  for ($i$=1; $i<n$; $i$++)        //  $n$-1 iterations
for (each($u$, $v$)$\in E$ )        // For each edge, relaxation is used
 if ( dist[$v$] -$w_{uv}$ >dist[$u$] )
{ dist[$v$]=dist[$u$]+ $w_{uv}$; [$v$]=$u$; }
for ( each ($u$, $v$)$\in E$)        //If there exists negative-weight cycle, return false
    if ( dist[$v$] -$w_{uv}$ >dist[$u$] )  return  false;
return  true;
}
```

- The reason why there are n-1 iterations in Bellman-Ford Algorithm is that if there is a shortest past between two vertices, each vertex will appear at most one time in the path, that is, there are at most n-1 edges in the path.
- The time complexity for Bellman-Ford Algorithm is O(VE).

Minimum Transport Cost

- Source: ACM 1996 Asia Regional Shanghai
- IDs for Online Judge: UVA 523

- There are *N* cities in Spring country. Between each pair of cities there may be one transportation track or none. Now there is some cargo that should be delivered from one city to another. The transportation fee consists of two parts:
- 1. the cost of the transportation on the path between these cities, and
- 2. a certain tax which will be charged whenever any cargo passing through one city, except for the source and the destination cities.
- You must write a program to find the route which has the minimum cost.

- Input
- The data of path cost, city tax, source and destination cities are given in the input file, which is of the form:

a_{11}	a_{12}	…	a_{1n}
a_{21}	a_{22}	…	a_{2n}
…	…		…
a_{n1}	a_{n2}	…	a_{nn}
b_1	b_2	…	b_n
c	d		
e	F		
…	…		
g	h		

- where a_{ij} is the transport cost from city i to city j, a_{ij} = -1 indicates there is no direct path between city i and city j. b_i represents the tax of passing through city i. And the cargo is to be delivered from city c to city d, city e to city f, ..., and city g to city h.

- **Output**
- You must output the sequence of cities passed by and the total cost, which is of the form:
- From *c* to *d* :
- Path: c->c_1-> ->c_k->d
- Total cost : …
-
- …
-
- From *e* to *f* :
- Path: e->e_1-> ->e_k->f
- Total cost : …

Shortest Path Faster Algorithm (SPFA Algorithm)

- The Shortest Path Faster Algorithm (SPFA) is an improvement of the Bellman–Ford algorithm which computes single-source shortest paths in a weighted directed graph. SPFA is suitable for random sparse graphs and graphs containing negative-weight edges.

```
void spfa(int s)   // SPFA is used to compute shortest paths from single-source s to other vertices
{
  Queue Q is empty;
for (i=0; i<101; i++)      //Initialization
{ dist[i] =∞; [i]=nil;  }
     dist[s]=0;
Add s into Queue Q;
while ( Q is not empty)
{
     Delete the front element x from Q;
       for(i=1; i<=n; i++)    // Relaxation
       if (dist[i]-$w_{xi}$>dist[x])
         { dist[i]=dist[x]+$w_{xi}$; [i]=x;
             if (Vertex i is not in Q)
                Vertex i is added into Q;
             }
     }
}
```

- *Q* is a queue. The time complexity for deleting element *u* from *Q* and visiting its adjacent vertices is O(*d*), where *d* is the out-degree of vertex *u*. The average out-degree for a vertex is E/V. Therefore the time complexity of dealing with a vertex is O(E/V). Suppose the number of adding vertices to *Q* is *h*. And it is related to edges' weights. Suppose *h*=*kV*. Then the time complexity of SPFA is *T*=O(*h** E/V)=O(*kE*). In general, *k* is a little constant. Therefore the time complexity of SPFA is O(*E*).

Longest Paths

- IDs for Online Judge: UVA 10000

- It is a well known fact that some people do not have their social abilities completely enabled. One example is the lack of talent for calculating distances and intervals of time. This causes some people to always choose the longest way to go from one place to another, with the consequence that they are late to whatever appointments they have, including weddings and programming contests. This can be highly annoying for their friends.
- César has this kind of problem. When he has to go from one point to another he realizes that he has to visit many people, and thus always chooses the longest path. One of César's friends, Felipe, has understood the nature of the problem. Felipe thinks that with the help of a computer he might be able to calculate the time that César is going to need to arrive to his destination. That way he could spend his time in something more enjoyable than waiting for César.

- Your goal is to help Felipe developing a program that computes the length of the longest path that can be constructed in a given graph from a given starting point (César's residence). You can assume that the graph has no cycles (there is no path from any node to itself), so César will reach his destination in a finite time. In the same line of reasoning, nodes are not considered directly connected to themselves.

- **Input**
- The input consists of a number of cases. The first line on each case contains a positive number *n* (1<*n*≤100) that specifies the number of points that César might visit (i.e., the number of nodes in the graph).
- A value of *n* = 0 indicates the end of the input.
- After this, a second number *s* is provided, indicating the starting point in César's journey (1≤*s*≤*n*). Then, you are given a list of pairs of places *p* and *q*, one pair per line, with the places on each line separated by white-space. The pair "*p q*" indicates that César can visit *q* after *p*.
- A pair of zeros ("0 0") indicates the end of the case.
- As mentioned before, you can assume that the graphs provided will not be cyclic.

- **Output**
- For each test case you have to find the length of the longest path that begins at the starting place. You also have to print the number of the final place of such longest path. If there are several paths of maximum length, print the final place with smallest number.
- Print a new line after each test case.