ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Algorithms Lab HS22
Department of Computer Science
Prof. Dr. A. Steger, Prof. Dr. E. Welzl
cadmo.ethz.ch/education/lectures/HS22/algolab

Solution — San Francisco

Note: After noticing a slight inaccuracy, the following extra assumption was added to test cases 1 and 2 (in addition to the existing ones), in order to be more precise: "you may assume that for every hole u on the wooden board, there is some route from Angvariationu-toke to a Weayaya hole that passes through u". The test cases stay the same.

1 Modelling

We are given a labelled directed weighted *multigraph* G (parallel edges and loops are allowed) on the vertex set $\{0, \dots, n-1\}$ with m directed edges. Each edge e has a non-negative weight $p(e)$.

We consider a single player game on G in which a single marble is moved along the edges of the graph. The marble starts out at vertex 0. The player is allowed to repeatedly move it along any directed edge e from u to v, thereby *scoring* $p(e)$ many *points*. Traversing each such edge counts as one *move*. Note that each edge can be traversed and scored multiple times throughout the game. Additionally, if the marble ends up at a vertex with out-degree 0, it directly goes back to vertex 0. This action is not considered a move and no points are scored through it.

We are asked to determine whether it is possible to score at least x points in k moves, and if so, to find the minimum number of moves to score at least x points.

The first simplifying observation we can make is that we can eliminate parallel edges in G by only keeping the edge e with maximum $p(e)$ among all edges from u to v, for each ordered pair of vertices (u, v). This is because, if two edges have the same head and tail, it is always better or at least as good to traverse the one that scores more points. There is no downside to scoring more points since we are required to score *at least* x points. Thus, we can assume that G has no parallel edges (but we still need to allow loops).

Next, we turn to modelling the *free action* of moving the marble from a vertex with out-degree 0 to vertex 0. There are multiple ways to do this. The first one is as follows: for each edge e from u to v where v is a vertex of out-degree 0, add to G an edge from u to 0 directly with weight $p(e)$. This way, passing through vertex v and then 'jumping' to 0 is modelled as one move with equivalent score. Note that this modification of the graph increases the number of edges of the graph with a multiplicative factor of at most 2. This is in contrast to the following alternative modification of the graph: for each edge e from 0 to v and for each vertex u with out-degree 0, one could add a direct edge from u to v with weight $p(e)$. This would model the problem correctly, but could potentially increase the number of edges by a factor of n, thereby slowing down any solution significantly and leading to time limit on worst-case test sets. A second feasible modification of G that would not affect the time complexity would be to 'merge' all vertices of out-degree 0 with vertex 0. Finally, one could avoid modifying the graph G explicitly and instead simply treat vertices of out-degree 0 as identical to vertex 0. We opt for this last approach for the rest of the solution.

2 Algorithm Design

Taking a look at the constraints, we see that $n \leqslant 10^3$, $m \leqslant 4 \cdot 10^3$, $x \leqslant 10^{14}$, and $k \leqslant 4 \cdot 10^3$. The first thing we can conclude from here is that the time complexity of our algorithm should not be proportional to x, since that would be way too slow. Next, we also see that a solution that is exponential in any of these parameters could not get 100 points. A time complexity of order $O(nm)$, $O(nk)$, or $O(mk)$ would be a good candidate, but anything slower than this, such as $O(nmk)$ would be too slow.

$O(k(\frac{n}{k})^k)$ solution (30 points). *In the first group of test sets, we are guaranteed that $n \leqslant 40$ and $k \leqslant 20$, that all walks from vertex 0 to a vertex of out-degree 0 use precisely k edges, and that for every vertex u, there is some walk from vertex 0 to a vertex of out-degree 0 that passes through u.*

Since n and k are small, a naive brute-force algorithm would be enough for the first group of test sets. The problem exhibits the optimal subproblem structure, that is, we can break it down into smaller subproblems of the same type, by decreasing x or k. This implies we should look for an appropriate recursive function. Note that x can still be large, so the time complexity of our solution should not depend on it. Therefore, the number of points we aim to score should not be an argument of our recursive function. Better options for arguments of it are the number of moves we have yet to play, and the current vertex we are at. Thus, we define the quantity $f(u, r)$ which denotes the maximum number of points we can score with r moves if the marble starts at vertex u. Note that if we have a way of computing $f(u, r)$ for any u and r, we can then solve the problem by checking if $f(0, r) \geqslant x$ for each $r = 0, 1, \ldots, k$.

We now derive a recursive formula for $f(u, r)$. Note that if $r = 0$, then $f(u, r) = 0$ since we cannot do anything. For $r > 0$, we have

$$f(u, r) = \max_{e=(u,v)\in E(G)} \{p(e) + f(v, r-1)\}.$$

That is because from u, we have as many options as there are outgoing edges, and for each such edge $e = (u, v)$, we can score $p(e)$ points by traversing it and $f(v, r-1)$ more points by solving the smaller subproblem of playing the game for $r-1$ moves starting at v. This recursive formulation directly gives rise to an algorithm for computing $f(u, r)$, given in pseudo-code below, where $E(u)$ denotes the set of outgoing edges from vertex u in G.

```
int f(int u, int r){
  if r == 0 return 0
  int result = 0
  for every e = (u,v) in E(u):
    result = max(f(v,r-1) + p(e),result);
  return result
}
```

Now that we know how to compute $f(u, r)$, we can solve the problem as discussed above. The following pseudo-code makes this explicit.

```
int solve(){
    for i = 0 to k:
      if f(0,i) >= x:
        return i
    return "Impossible"
}
```

Note that since all walks from vertex 0 to a vertex of out-degree 0 use exactly k edges, it follows that every valid sequence of at most k moves starting from vertex 0 forms a walk that never 'jumps' back to vertex 0, so we do not need to worry about this in the solution. The correctness of the algorithm then follows from the fact that when computing $f(0,i)$, we consider all possible walks of length i starting at vertex 0 in G, and find the one that gives the highest score.

We now analyse the time complexity of this solution. The first important observation is that for each vertex u, all walks from 0 to u must have the same length. If u has out-degree 0, we already know that this is the case. Otherwise, there is some other vertex v of out-degree 0 reachable from u. Then if there are two walks of different length from 0 to u, they can be continued to walks of different length from 0 to v, reaching a contradiction. Thus, every vertex is at some fixed distance from vertex 0. Suppose there are v_i vertices at distance i from 0, for $1 \leqslant i \leqslant k$ and note that $\sum_{i=1}^{k} v_i = n-1$. Then the number of walks from 0 of length k is at most $\prod_{i=1}^{k} v_i$, which under the constraint $\sum_{i=1}^{k} v_i \leqslant n$ is maximised for $v_1 = \cdots = v_k = \frac{n}{k}$ (suppose instead some $v_i < v_j$, then picking $v_i' = v_j' = \frac{v_i+v_j}{2}$ instead gives $v_i' = v_i + d$ and $v_j' = v_j - d$ for $d = \frac{v_j - v_i}{2}$, so $v_i' + v_j' = v_i + v_j$ but $v_i v_j = v_i' v_j' - d^2$). Thus, the time complexity of our algorithm becomes $O(k\prod_{i=1}^{k} v_i) \in O\big(k(\frac{n}{k})^k\big)$. This in turn is maximised when $n = 40$ and $k = 20$, thus getting roughly $20 \cdot 2^{20}$ operations, which is feasible in under a second. However, since the time complexity of this solution is exponential, it is not enough to solve the second and third groups of test sets.

$O\big(n\log(n) + m\big)$ solution (60 points). *In the second group of test sets, we are only guaranteed that all walks from vertex 0 to a vertex of out-degree 0 use precisely k edges, and that for every vertex u, there is some walk from vertex 0 to a vertex of out-degree 0 that passes through u.*

As we are no longer guaranteed that n and k are small, the previous solution can require more than $500 \cdot 2^{500}$ operations (for $n = 1000$, $k = 500$) which would easily time out.

However, similarly to the previous solution, we can ignore all connections from zero-degree vertices to vertex 0 as walks of length k will never 'jump' back to vertex 0. Furthermore, we can again observe that for each vertex u, all walks from 0 to u must have the same length. We leverage this property to solve the second test sets. This in particular allows us to disentangle the search for the maximum achievable score for all paths ending at some vertex u, and the condition that the paths needs to be the shortest possible.

That is, we have

$$\begin{aligned} &\min\big\{\text{length}(\tau_u)\big|u \in V, \tau_u \in \mathcal{P}(0,u), \text{score}(\tau_u) \geqslant x\big\} \\ =&\min\big\{\text{dist}(0,u)\big|u \in V, \max_{\tau_u \in \mathcal{P}(0,u)} \text{score}(\tau_u) \geqslant x\big\} \end{aligned}$$

where $\mathcal{P}(0,u)$ is the set of all paths from 0 to u.

The problem thus decomposes into 1) finding the maximum score that can be achieved from 0 to every vertex u, and 2) picking the closest vertex to 0 whose maximum score is above x and returning the distance of u from 0.

The first step can be achieved with algorithms such as Dijkstra's algorithm, with the caveats that Dijkstra's algorithm can only be used to graphs with non-negatively weighted edges, and finds the least weighted path. In order to be able to apply the algorithm here, we can simply transform the weights p into $\hat{p}$ with the following transformation:

$$\forall e \in E(G), \hat{p}(e) = \max_{e' \in E(G)} p(e') - p(e)$$

This transformation inverts the weights and adds a constant to all edges to preserve non-negativity. However, as all the walks have the same length, the constant doesn't change the result. Thus, the shortest path in the transformed graph will correspond to the longest path in the original graph.

Once the maximum score is computed for all vertices, the second step can be achieved by simply running a BFS algorithm and returning the distance at which we first observe a vertex whose score is higher than the desired threshold, and "Impossible" if such a vertex does not exist.

```
void find_closest_solution(int n, long long x, int k,
const vector<long long>& precomputed_maximum_score,
vector<vector<int> >& graph){
    # precomputed_maximum_score: the precomputed score of all vertices using e.g.
    # Dijkstra's algorithm.
    vector<int> d(n,-1);
    d[0] = 0;
    queue<int> q;
    q.push(0);

    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(int v : graph[u]){
            if(d[v] == -1) {
                q.push(v);
                d[v] = d[u] + 1;
            }
            if(precomputed_maximum_score[v] >= x){
                cout << d[v] << endl;
                return;
            }
            if(d[v] > k) break;
        }
    }
    cout << "Impossible" << endl;
    return;
}
```

Given that Dijkstra's algorithm has complexity $\mathbf{O}(\mathbf{n}\log(\mathbf{n}) + \mathbf{m})$, and BFS $\mathbf{O}(\mathbf{n} + \mathbf{m})$, the total complexity of the algorithm is $\mathbf{O}(\mathbf{n}\log(\mathbf{n}) + \mathbf{m})$. In the worst case where $n = 1000$ and $m = 4000$, the number of operations is small enough. Unfortunately however, because we made use of the strong assumptions, the solution does not work in the general case.

$\mathbf{O}((\mathbf{n} + \mathbf{m})\mathbf{k})$ solution (100 points). *In the third group of test sets, there are no additional assumptions.*

We can extend the 30 point solution to deal with the nodes without outgoing edges. We just have to move back to the initial node without using a move when necessary. That would look as follows:

```
int f(int u, int r){
    if r == 0 return 0
    int result = 0
    for every e = (u,v) in E(u):
        result = max(f(v,r-1) + p(e),result);
    if |E(u)| == 0:
        result = f(0, r);
    return result
}
```

While this recursive function has an exponential run-time, we can observe that the range for u is $[0, n)$ and the range for r is $[0, k]$. This means that many sub-problems will overlap and we only need a memoization table of size $O(nk)$ to store all possible values of the function. If we add memoization to our function it will look like this:

```
int f(int u, int r){
    if r == 0 return 0;
    if memo(u, r) exists:
        return memo(u, r);
    int result = 0;
    for every e = (u,v) in E(u):
        result = max(f(v,r-1) + p(e),result);
    if |E(u)| == 0:
        result = f(0, r);
    memo(u, r) = result;
    return result
}
```

The correctness of this function is proved in the same way to the correctness of the 30 point solution. However, computing the time complexity is a little bit more tricky. A naive (**and wrong!**) approach would be to count the total number of function calls and to multiply it with the slowest possible function call. This would give a runtime of $O(nk \cdot n)$, which would be too slow for the given bounds in the problem statement. However, with a more careful analysis we can show that the runtime is actually $O((n + m)k)$. In order to show this, we have to analyze the runtime for filling the memoization table for a fixed value of r. In that case, we will fill one entry per node in the graph. On top of this, we will do one iteration of the for loop visiting each of the outgoing edges of every node. As every edge is outgoing for exactly one node, we will only go over each edge once. This means, that the runtime to fill the values for a fixed value of r is $O(n + m)$, which would give us a total runtime of $O((n + m)k)$.

3 Full Solution

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> edges;
vector<vector<long>> weights;
vector<vector<long>> memo;
```

```

long F(int current_node, int moves_left) {
        if(moves_left == 0) return 0;
        if (memo[current_node][moves_left] != -1)
        return memo[current_node][moves_left];

        long result = 0;
        for (int i = 0; i < edges[current_node].size(); i++) {
                int next_node = edges[current_node][i];
                int edge_weight = weights[current_node][i];
                result = max(edge_weight + F(next_node, moves_left - 1), result);
        }
        if (edges[current_node].size() == 0) result = F(0, moves_left);

        memo[current_node][moves_left] = result;
        return result;
}

void testcase() {
        int n,m,k;
        long x;
        cin >> n >> m >> x >> k;

        edges = vector<vector<int>>(n);
        weights = vector<vector<long>>(n);
        memo = vector<vector<long>>(n, vector<long>(k + 1, -1));

        for(int i = 0; i < m; ++i) {
                int u, v, p;
                cin >> u >> v >> p;
                // This solution can handle multi-edges naturally,
                // so there is no need to treat these differently.
                edges[u].push_back(v);
                weights[u].push_back(p);
        }

        for (int i = 0; i <= k; i++) {
                if (F(0, i) >= x) {
                        cout << i << endl;
                        return;
                }
        }
        cout << "Impossible" << endl;
}

int main() {
        ios_base::sync_with_stdio(false);

        int t; cin >> t;
        for (int i = 0; i < t; i++) {
                testcase();
        }
        return 0;
}
```