Synchronous Shopping editorial

Problem: https://www.hackerrank.com/challenges/synchronous-shopping/problem

Two cats, many fishes

Required knowledge: Dijkstra, bitmasks.

Editorial by zxqfd555: https://www.hackerrank.com/challenges/synchronous-shopping/editorial

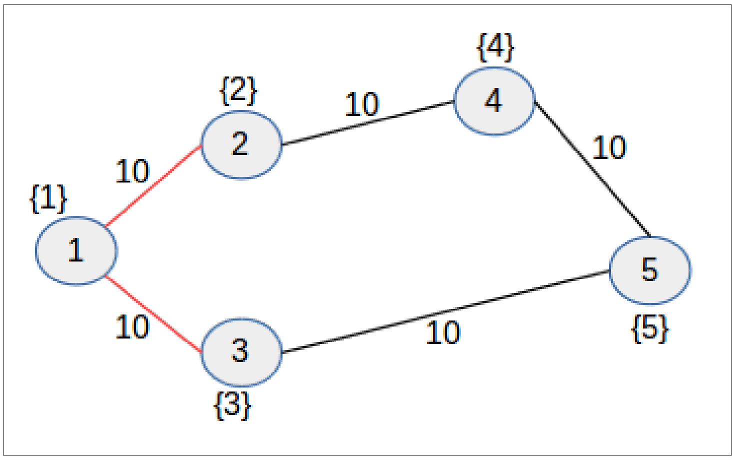
> This problem can be solved with Dijkstra's algorithm. Let's denote the state as (V, B), where V is the number of shopping centers and P is a bitmask of the first K bits denoting the kinds of fish which have already been bought.
>
> The starting state is $(1, 0)$, meaning we start at shopping center 1 and have not yet purchased any fish. The shortest distance to the state $D_{(V,B)}$ denotes the minimum time required to visit shopping center V with fish from the mask B bought.
>
> While spreading from the current (V, B) state, there are two possible options:
>
> 1. To state (V, B') with the time $D_{(V,B)}$ where $B' = B \text{ or } maskOfFishSoldAt[V]$. Recall that buying any amount of fish doesn't take any time, so it is always optimal to buy all fish sold in the shopping centers. This transition corresponds to buying the fish.
> 2. To state (Y, B) where Y is adjacent to V with the time $D_{(V,B)} + W$, and W is the time required to pass the road from V to Y. This transition corresponds to moving by a road.
>
> When all the minimal times are calculated, let's brute-force the mask B_1 of the fish that will be bought by Little Cat and mask B_2 of the fish that will be bought by Big Cat. The essential condition is B_1 or $B_2 = 2^K - 1$. Then, the minimal time for this configuration will simply be equal to $\min(D_{(N,B_1)}, D_{(N,B_2)})$. Among all these configurations, choose the best one (i.e., the one having the minimal answer).

My editorial

At each visited shop, we have to know the distance after the collection of some fishes.

For example, fishes {1, 2, 3} could be collected after running a distance of 30:

path = <1, 2, 1, 3>



As you see, the same shop could be visited multiple times. For each visit (to the same shop), the fishes's collections states could be changed.

So, for each shop, we need to store distances for different status.

To do that, we need a $n * all\_collections\_states$ matrix to store distances.

To compute $all\_collections\_states$, we use bitmaks.

For example:

for $k = 5$ (the number of types of fish sold in Bitville):

- 00001: fish of type 1 is collected.
- 01010: fishes of types 2 and 4 are collected
- …….

we need a $n * 2^k matrix$ ($2^k = 1 << k$)

Needed data structures (see code below)

- An array of lists (or vectors) to store the each vertex neighbors.
- An array of each vertex bitmask (or mask).
- A $n * (1 << k)$ matrix to save distances.
- A priority queue for Dijkstra algorithm.

How to do

(1) Fill the the array of each vertex bitmask.

(2) Build the graph.

(3) run Dijkstra to fill the $n * (1 << k)$ matrix.

(4) Simulate the two cats run into the $n * (1 << k)$ matrix to compute the minimum amount of time it will take for the cats to collectively purchase all k fish.

Dijkstra algorithm

The Dijkstra algorithm will compute the distance to reach a vertex v from the source (vertex 1), by considering the fact all collections states of v.

for this graph, we have a $3 * (1 << 3)$ matrix of distances:

d			
	1	2	3
001			
010			
011			
100			
101			
110			
111			

The principle of Dijkstra algorithm doesn’t change. Two thing will change compared to conventional Dijkstra algorithm:

The priority queue stores the distance computed a vertex v ($d[v][mask]$) with v and its $mask$. So, each vertex v, ($d[v][mask]$, (v, $mask$)) is stored.

The relaxation pseudo-code will be:

$$for\ each\ v \in adj[u]$$

$$\quad if\ d[v][masks[u] \mid masks[v]] > d[u][mask[u]] + w\ (u, v)$$

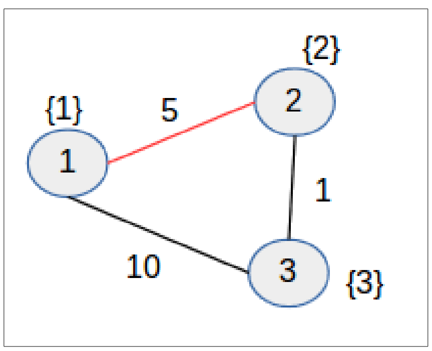
$$\qquad d[v][masks[u] \mid masks[v]] = d[u][mask[u]] + w\ (u, v)$$

$$\qquad q \leftarrow q \cup \{(d[v][mask], (v, mask))\}$$

why the bitwise operation $masks[u] \mid masks[v]$?

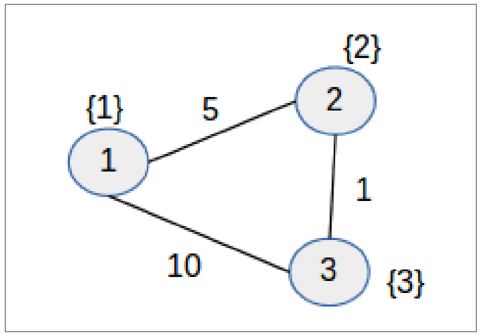
By moving from the vertex u to v, the collection states of fishes will change.

For example:

	In vertex 1, mask = 001. In vertex 2, mask = 010. By moving from vertex 1 to vertex 2, the mask will be equal to 001 | 010 = 011.

Run by hand on an example

let’s take this graph:



Fill the the array of each vertex bitmask.

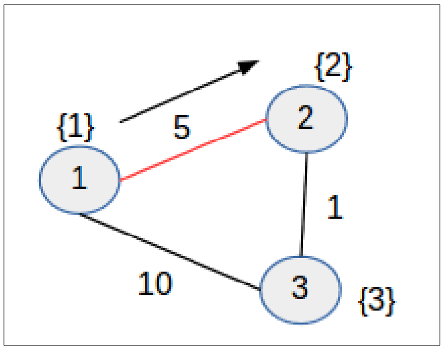
masks		
1	2	3
001	010	100

Run Dijkstra algorithm

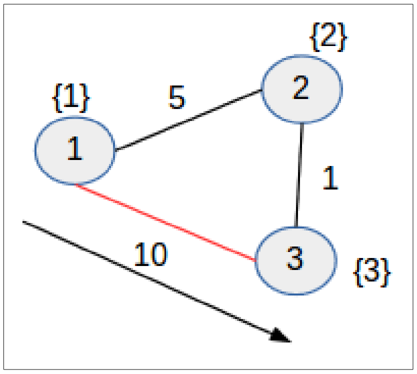
d			
	1	2	3
001	0		
010			
011	10	5	
100			
101	20	30	10
110			
111	~~16~~ 12	7	6

- d[1][masks[1]] = d[1][001] = 0
  q = {(0, (1, 001))}

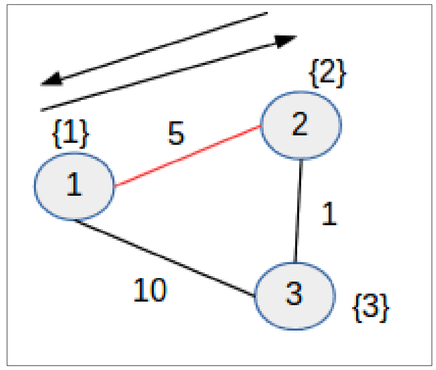
- 1’s neighbors, mask = 001:
  q = {}
  - 2:
    d[2][1’s mask | masks[2]] > d[1][masks[1]] + w (1, 2)
    d[2][001 | 010] > d[1][001] + w (1, 2)
    d[2][011] > 0 + 5
    ∞ > 5 => d[2][011] = 5
    q = {(5, (2, 011))}



  - 3:
    d[3][masks[1] | masks[3]] > d[1][masks[1]] + w (1, 3)
    d[3][001 | 100] > d[1][001] + w (1, 3)
    d[3][101] > 0 + 10
    ∞ > 10 => d[3][101] = 10
    q = {(5, (2, 011)), (10, (3, 101))}

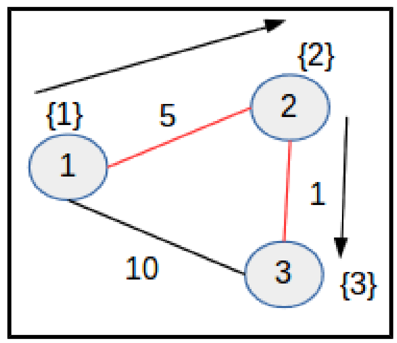


- 2’s neighbors, mask = 011:
  q = {(10, (3, 101))}



  - 1:

    d[1][011 | 001] > d[2][011] + w (2, 1)
    d[1][011] > 5 + 5
    ∞ > 10 => d[1][011] = 10
    q = {(10, (3, 101)), (10, (1, 011))}

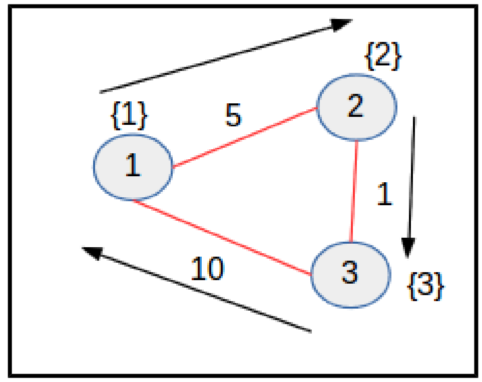


  - 3:

    d[3][011 | 100] > d[2][010] + w (2, 3)
    d[3][111] > 5 + 1
    ∞ > 6 => d[3][111] = 6
    q = {(6, (3, 111)), (10, (3, 101)), (10, (1, 011))}

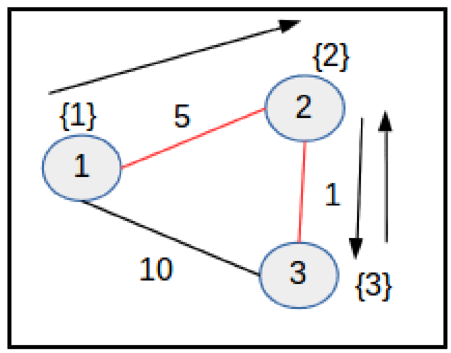
- 3's neighbors, mask = 111:
  q = {(10, (3, 101)), (10, (1, 011))}
  - 1:

    d[1][111 | 001] > d[3][111] + w (3, 1)
    d[1][111] > 6 + 10
    ∞ > 16 => d[1][111] = 16
    q = {(10, (3, 101)), (10, (1, 011)),
    (16, (1, 111))}



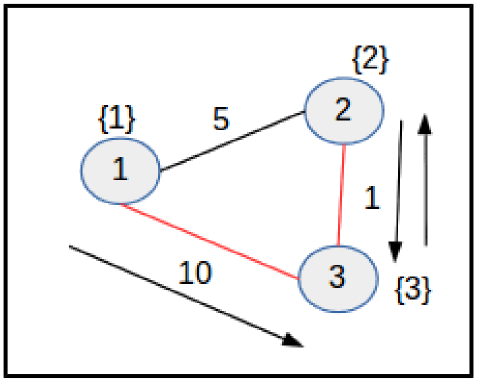
  - 2:

    d[2][111 | 010] > d[3][111] + w (3, 2)
    d[2][111] > 6 + 1
    ∞ > 7=> d[2][111] = 7
    q = {(7, (2, 111)), (10, (3, 101)), (10, (1, 011)),
    (16, (1, 111))}



- 2’s neighbors, mask = 111:
  q = {(10, (3, 101)), (10, (1, 011)), (16, (1, 111))}
  - 1:

    d[1][111 | 001] > d[2][111] + w (2, 1)
    d[1][111] > 7 + 5
    16 > 12 => d[1][111] = 12
    q = {(10, (3, 101)), (10, (1, 011)), (12, (1, 111)),
    (16, (1, 111))}

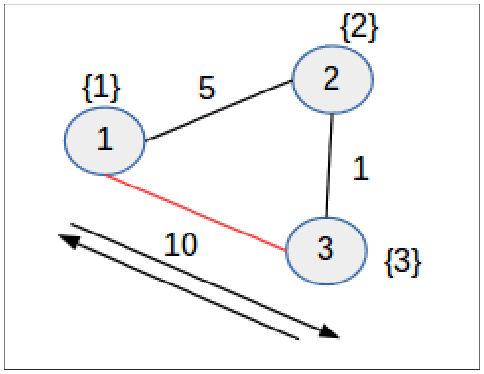


  - 3:

    d[3][111 | 100] > d[2][111] + w (2, 3)
    d[3][111] > 7 + 1
    6 > 8=> No.

- 3's neighbors, mask = 101:
  q = {(10, (1, 011)), (12, (1, 111)), (16, (1, 111))}
  - 1:

    d[1][101 | 001] > d[3][101] + w (3, 1)
    d[1][101] > 10 + 10
    ∞ > 20 => d[1][101] = 20
    q = {(10, (1, 011)), (12, (1, 111)), (16, (1, 111)),
    (20, (1, 101))}



  - 2:

    d[2][101 | 010] > d[3][101] + w (3, 2)
    d[2][111] > 10 + 1
    7 > 11=> No.

- 1’s neighbors, mask = 011:
  q = {(12, (1, 111)), (16, (1, 111)), (20, (1, 101))}
  - 2:
    d[2][011 | 010] > d[1][011] + w (1, 2)
    d[2][011] > 10 + 5
    5 > 15 => No.
  - 3:
    d[3][011 | 100] > d[1][011] + w (1, 3)
    d[3][111] > 10 + 10
    6 > 20=> No.

- 1’s neighbors, mask = 111:
  q = {(16, (1, 111)), (20, (1, 101))}
  - 2:

    d[2][111 | 010] > d[1][111] + w (1, 2)
    d[2][111] > 12 + 5
    7 > 17 => No.

  - 3:

    d[3][111 | 100] > d[1][111] + w (1, 3)
    d[3][111] > 10 + 10
    6 > 20=> No.

- 1’s neighbors, mask = 111:
  q = {(20, (1, 101))}
  - 2:

    d[2][111 | 010] > d[1][111] + w (1, 2)
    d[2][111] > 12 + 5
    7 > 17 => No.

  - 3:

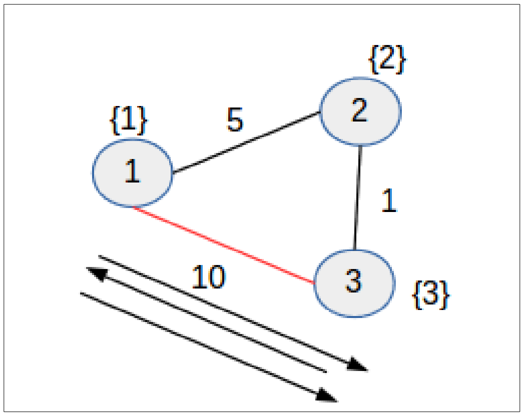
    d[3][111 | 100] > d[1][111] + w (1, 3)
    d[3][111] > 10 + 10
    6 > 20=> No.

- 1’s neighbors, mask = 101:
  q = {}
  - 2:

    d[2][101 | 010] > d[1][101] + w (1, 2)
    d[2][111] > 20 + 5
    7 > 25 => No.

  - 3:

    d[3][101 | 100] > d[1][101] + w (1, 3)
    d[3][101] > 20 + 10
    ∞ > 30=> d[3][101] = 30.
    q = {(30, (3, 101))}



- 3's neighbors, mask = 101:
  q = {}
  - 2:
    d[2][101 | 010] > d[3][101] + w (3, 2)
    d[2][111] > 30 + 1
    7 > 31 => No.
  - 1:
    d[1][101 | 001] > d[3][101] + w (3, 1)
    d[1][101] > 20 + 10
    20 > 30=> No.

q = ∅

The final d() values:

d			
	1	2	3
001	0	∞	∞
010	∞	∞	∞
011	10	5	∞
100	∞	∞	∞
101	20	30	10
110	∞	∞	∞
111	~~16~~ 12	7	6

Some distances were not computed, because there is no need to compute some of them or impossible to compute them. Like $d[i][100]$, $(1 \le i \le 3)$, is impossible to compute because it's not possible to get fish type #3, without getting fish type #1 (we start at shop #1)

Simulate the two cats run into the $n * (1 << k)$ matrix to compute the minimum amount of time it will take for the cats to collectively purchase all k fish.

"*If one cat finishes shopping before the other, he waits at shopping center* n *for his partner to finish; this means that the total shopping time is the maximum of Little and Big Cats' respective shopping times.*"

Little & Big Cat, will run throw all the matrix of the d() values and when they collect all the k type of fishes and reach the shopping center n, will take the minimum of the maximum of Little and Big Cat's respective shopping times.

$BestTime \leftarrow \infty$

$For\ cat1 \in [1, 1 << k\ ]$

$\quad for\ cat2\ [cat1, 1 << k\ ]$

$\quad\quad if\ (cat1 \mid cat2) = (1 << k) - 1$

$\quad\quad\quad bestTime \leftarrow min(bestTime, max(d[n][cat1], d[n][cat2]))$

Executing this pseudo-code on the d() values matrix, the computed $bestTime$ is equal to 6

C++ Code: $O(MlogN\,.\,2^k)$

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 1e+9;

int n, m, k;

int **dist = NULL;
int *masks = NULL;
vector<pair<int, int> > *adj = NULL;

void createAdjList(void) {
  for (int edge = 0 ; edge < m ; ++edge) {
    int x , y , z;
    scanf("%d%d%d", &x, &y, &z);
    x--;
    y--;
    adj[x].push_back(make_pair(y, z));
    adj[y].push_back(make_pair(x, z));
  }
}
```

```
void dijkstra(int start) {
  for(int i = 0; i < n; ++i)
    for(int j = 0; j < (1 << k); ++j)
      dist[i][j] = INF;

  dist[start][masks[start]] = 0;

  priority_queue<pair<int, pair<int, int> >,
    vector<pair<int, pair<int, int> > >,
    greater<pair<int, pair<int, int> > > > q;

  q.push({dist[start][masks[start]], {start, masks[start]}});
  while (!q.empty()){
    int current = q.top().second.first;
    int currentMask = q.top().second.second;

    q.pop();

    for (auto neighbors: adj[current]) {
      int v = neighbors.first;
      int w = neighbors.second;
      if (dist[v][currentMask | masks[v]] > dist[current][currentMask] + w ) {
        dist[v][currentMask | masks[v]] = dist[current][currentMask] + w;
        q.push( {dist[v][currentMask | masks[v]], {v, currentMask | masks[v]}} );

      }

    }

  }
}
```

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

  cin >> n >> m >> k;

  dist = new int*[n];
  for (int i = 0 ; i < n ; ++i)
    dist[i] = new int[1 << k];

  adj = new vector<pair<int, int> >[n];
  masks = new int[n];

  for(int i = 0; i < n; ++i) {
    int ti;
    cin >> ti;

    for(int j = 1; j <= ti; ++j) {
      int ai;
      cin >> ai;

      //Store the mask of each node.
      masks[i] |= (1 << (ai - 1));
    }
  }

  createAdjList();


  dijkstra(0);


  int ans = INF;
  for(int i = 0; i < (1 << k); ++i)
    for(int j = i; j < (1 << k); ++j)
      if ((i | j) == ((1 << k) - 1))
        ans = min(ans, max(dist[n-1][i], dist[n-1][j]));

  cout << ans << endl;

  return 0;
}
```