Contest 3 Editorial

COMP4128 25T2

August 10, 2025

A Modernise

Algorithm

We need to form two road networks (left- and right-hand-driven), assigning the each town to whichever network we prefer. Each network should be formed of the cheapest edges that will connect them, i.e. a minimum spanning tree. If we connect these two MSTs with the cheapest edge between them, we should end up with an MST for the whole graph, so our method is to find the MST and remove the most expensive edge to split it into two.

Subtask

The graph is already a tree, and therefore its own minimum spanning tree. We remove the most expensive edge and add up the remaining edge weights.

Full Task

We can use an MST algorithm (Kruskal's or Prim's) and add up all the edges, subtracting the most expensive one. Alternatively, halt Kruskal's algorithm early when there are only two regions remaining.

Implementation Notes

There are up to $100,000$ edges, each of weight up to $10,000$, so the total of all edge weights does not overflow a 32-bit integer.

Reference Solution(s)

```
// Subtask solution by Raveen

#include <algorithm>
#include <iostream>
using namespace std;

int main (void) {
    int n, m;
    cin >> n >> m;
    int sum = 0, worst = 0;
    for (int j = 0; j < m; j++) {
        int a, b, l;
        cin >> a >> b >> l;
        sum += l;
        worst = max(worst, l);
    }
    cout << sum - worst << '\n';
}
```

```
// Solution by Raveen

#include<algorithm>
#include<iostream>
#include<tuple>
```

```
#include<vector>
using namespace std;

const int N = 100100;
int p[N]; // parent
int s[N]; // subtree size

void init (int n) {
  for (int i = 0; i < n; i++) {
    p[i] = i;
    s[i] = 1;
  }
}

int root (int x) { // path compression is optional
  return p[x] == x ? x : p[x] = root(p[x]);
}

bool join (int x, int y) {
  x = root(x);
  y = root(y);
  if (x == y) { return false; }
  if (s[x] < s[y]) {
    p[x] = y;
    s[y] += s[x];
  } else {
    p[y] = x;
    s[x] += s[y];
  }
  return true;
}

int main (void) {
  int n, m;
  cin >> n >> m;
  init(n);

  vector<tuple<int,int,int>> e;
  for (int i = 0; i < m; i++) {
    int a, b, l;
    cin >> a >> b >> l;
    e.emplace_back(l,a-1,b-1);
  }
  sort(e.begin(),e.end());

  int z = 0; // number of roads chosen
  int ans = 0;
  for (int i = 0; z < n-2; i++) {
    auto [l, a, b] = e[i];
    if (join(a, b)) {
      z++;
      ans += l;
    }
  }
  cout << ans << '\n';
}
```

B Avatar Tour

Preliminaries

Denote $\delta_{uv} = |h_u - h_v|$ for the difference in height between two mountains.

It is natural to represent the mountains and roads as a graph, where we represent mountains as nodes, and for each road connecting mountains u and v with heights $h_u \geq h_v$ we add edges

- $u \to v$ with weight $-c \times \delta_{uv}$, and
- $v \to u$ with weight $d \times \delta_{uv}$.

Note that since it is guaranteed that $c \leq d$, there will be no negative cycles in this graph.

Seeing that this graph has negatively weighted edges, it is natural to consider using the Floyd-Warshall or Bellman-Ford algorithms. However, we also need to keep track of what nations we have visited so far, and we might need to revisit the same node multiple times in order to visit all nations at least once, so directly running either algorithm on this graph will not work.

Subtask

Suppose there are only two nations, and we are at a particular mountain. The only other relevant information is whether we have been to the other nation or not.

Our nodes will be of the form (u, z), where u is the current mountain and z is a boolean (false representing that the other nation has not been visited, true representing that it has). Then for each road connecting mountains u and v in the *same* nation with heights $h_u \geq h_v$, and for both boolean values of z, we add edges

- (u, z) to (v, z) with weight $-c \times \delta_{uv}$, and
- (v, z) to (u, z) with weight $d \times \delta_{uv}$,

and for each road connecting mountains u and v in *different* nations with heights $h_u \geq h_v$, and for both boolean values of z, we add edges

- (u, z) to (v, true) with weight $-c \times \delta_{uv}$, and
- (v, z) to (u, true) with weight $d \times \delta_{uv}$.

We now want the shortest path from any (u, false) to any (v, true). This can be achieved by attaching a super-source s to all nodes where $z = \mathsf{false}$ (and optionally all nodes where $z = \mathsf{true}$ to a super-sink t) with edges of weight zero, and running the Bellman-Ford algorithm (as there are negative edges but no negative cycles).

There are $2n$ vertices and $2m$ edges in the graph we constructed, both of which are very small, so the program will run in an instant.

Full task

The general idea here is to add an extra parameter S to our node states to keep track of the current set of nations visited so far.

Denote p_u for the nation containing mountain u.

Our nodes will be of the form (u, S), where u is the current mountain and S is the set of nations visited so far. For each road connecting mountains u and v with heights $h_u \geq h_v$, and for each $S \subseteq \{1, \ldots, k\}$ we add edges:

- $(u, S) \to (v, S \cup \{p_v\})$ with weight $-c \times \delta_{uv}$ if $p_u \in S$, and
- $(v, S) \to (u, S \cup \{p_u\})$ with weight $d \times \delta_{uv}$ if $p_v \in S$.

We now want the shortest path from any $(u, \{p_u\})$ to any $(v, \{1, \ldots, k\})$. This can be achieved by attaching a super-source s to all nodes where $S = \{p_u\}$ (and optionally all nodes where $S = \{1, \ldots, k\}$ to a super-sink t) with edges of weight zero, and running the Bellman-Ford algorithm (as there are negative edges but no negative cycles).

There are 2^k possible subsets S. For each subset S, we have one copy of every mountain u, giving us $O(n2^k)$ nodes. For each subset S, we also add two edges per road (one per direction), giving us $O(m2^k)$ edges. Just using these values, we get that Bellman-Ford runs in $O(VE) = O(nm4^k)$ time, which does not run in time.

However, we can make the time complexity tighter by making the following observations regarding the maximum number of edges in our shortest path.

Claim 1: The shortest path visits exactly k different subsets.

Proof 1: Starting from a vertex (u, S), if we move along one of its outgoing edges to another mountain v, we have two cases:

- if $p_v \in S$, we have already previously visited nation p_v, so the destination of this edge is vertex (v, S) with the same subset S.
- if $p_v \notin S$, the destination of this edge is vertex (v, S') with a different subset S' formed by inserting p_v to S.

Notice that the only way to visit a different subset is by moving to a larger subset, so it is not possible to reach a different subset $\hat{S}$ with the same number of nations as S by any sequence of edges. This means that the shortest path will only visit one subset per subset size. Since there are k possible subset sizes, only k different subsets will be visited by the shortest path.

Claim 2: For each visited subset S, the shortest path will use at most $n - 1$ edges within that subset.

Proof 2: Suppose u and v are the first and last mountains visited in the portion of our shortest path during which the set of nations is S. If we were to use n or more edges in this portion of path, we would encounter a vertex (w, S) two or more times. Excising the portion of the path from the first occurence to the last occurence of such a repeated vertex corresponds to deleting a cycle, and since there are no negative cycles, this cannot make the path weight any greater. So by contradiction, only $n - 1$ edges are needed for each set of nations.

Note that this is the same argument for why we only need $|V| - 1$ relaxations in Bellman-Ford (refer to shortest paths lecture slide 13).

Claim 3: The full shortest path from the super-source s to a vertex with all nations visited will use at most $O(kn)$ edges.

Proof 3: From claim 1, the full shortest path will visit k subsets. From claim 2, the full shortest path will only use at most $n - 1$ edges within each subset, so we can attribute at most n edges per subset (accounting for the edge entering the subset from a different subset). Thus, the full shortest path will use at most $O(kn)$ edges.

Final Time Complexity

Using the above observations, we now know that Bellman-Ford will only do $O(kn)$ relaxations before early exiting (since no distances will be reduced further beyond that). Thus, this algorithm will run in $O(knm2^k)$ time which is good enough for the given bounds.

Conjecture

The shortest path from s to a vertex with all nations visited will use at most $2n$ edges.

All examples we found had solutions with at most $2n$ edges. If the graph of nation connectivity is star-like, then we can force a path of length close to $2n$ where one repeatedly travels between a hub nation and $k - 1$ spoke nations. We haven't proven whether this bound holds in general.

Implementation Notes

For the full problem, represent set S using a bitmask.

It is recommended to compute the edge weights on the fly, rather than storing them in the graph representation.

Reference Solutions

```
// Subtask solution by Raveen

#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

const int INF = 1000*1000*1000+7;
const int N = 55, M = 250;
int h[N], p[N];
vector<int> adj[N];
int dist[N][2]; // (current mountain, been to other nation?)
int n, m, k, c, d;

// car is on when i->j is upwards, discharge rate d
// car is not on when i->j is downwards, charge rate c
void relax (int i, int j, bool on) {
    for (int t = 0; t < 2; t++) {
        int& val = dist[j][t||(p[i] != p[j])]; // the distance we are trying to update
        val = min (val, dist[i][t] + (on ? d : c) * (h[j] - h[i]));
    }
}

void relax (void) { // global relaxation
    for (int i = 0; i < n; i++) {
        for (int j : adj[i]) { relax(i, j, h[i] < h[j]); }
    }
}

int main (void) {
    cin >> n >> m >> k >> c >> d;
    for (int i = 0; i < n; i++) {
        dist[i][0] = 0;
        dist[i][1] = INF;
    }
    for (int i = 0; i < n; i++) { cin >> h[i]; }
    for (int i = 0; i < n; i++) { cin >> p[i]; }
    for (int j = 0; j < m; j++) {
        int a, b;
        cin >> a >> b;
        a--; b--;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    for (int t = 0; t < 2*n-1; t++) { relax(); }

    int ans = INF;
    for (int i = 0; i < n; i++) { ans = min(ans, dist[i][1]); }
    if (ans == INF) { cout << "impossible\n"; }
    else            { cout << ans << '\n';    }
}
```

```
// Solution by Raveen

#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

const int INF = 1000*1000*1000+7;
const int N = 55, M = 250, K = 11;
int h[N], p[N];
vector<int> adj[N];
int dist[N][1<<K]; // (current mountain, mask of nations seen)
int n, m, k, c, d;

// car is on when i->j is upwards, discharge rate d
// car is not on when i->j is downwards, charge rate c
bool relax (int i, int j, bool on) {
    bool relaxed = false;
    for (int mask = 0; mask < (1<<k); mask++) {
        if (mask & (1<<p[i])) {
            int& val = dist[j][mask|(1<<p[j])];
            int alt = dist[i][mask] + (on ? d : c) * (h[j] - h[i]);
            if (alt < val) {
                val = alt;
                relaxed = true;
            }
        }
    }
    return relaxed;
}

// global relaxation, returns whether any changes happened
bool relax (void) {
    bool relaxed = false;
    for (int i = 0; i < n; i++) {
        for (int j : adj[i]) {
            if (relax(i, j, h[i] < h[j])) { relaxed = true; }
        }
    }
    return relaxed;
}

int main (void) {
    cin >> n >> m >> k >> c >> d;
    for (int i = 0; i < n; i++) { fill(dist[i], dist[i] + (1<<k), INF); }
    for (int i = 0; i < n; i++) { cin >> h[i]; }
    for (int i = 0; i < n; i++) {
        cin >> p[i];
        p[i]--;
        dist[i][1<<p[i]] = 0;
    }
    for (int j = 0; j < m; j++) {
        int a, b;
        cin >> a >> b;
        a--; b--;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    // n * 2^k global relaxations
    // early exit if any relaxation does nothing
    for (int t = 0; t < (n<<k)-1; t++) { if (!relax()) { break; } }

    int ans = INF;
    for (int i = 0; i < n; i++) { ans = min(ans, dist[i][(1<<k)-1]); }
    // max legitimate answer is n * k * c * (hmax - hmin), approx 5e7
    if (ans > INF/10) { cout << "impossible\n"; }
    else              { cout << ans << '\n';    }
}
```

C Hippo Ponds

This problem is sourced from ICPC Latin America 2021/22, Brazil Subregional.

Subtask

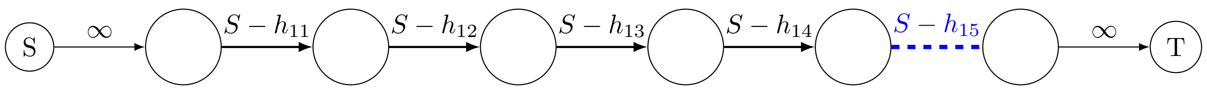
Each hippo has at most one best friend, so they can be solved in pairs. Since the numbers are small, we can try putting the two hippos in every pond and see which pair of positions gives the highest happiness. Friendless hippos are put into their favourite pond. Our answer will be the sum of the happinesses of each pair/singleton of hippos.

This gives us an $O(nm^2)$ solution, which is sufficient for $n, m \leq 40$.

Full task

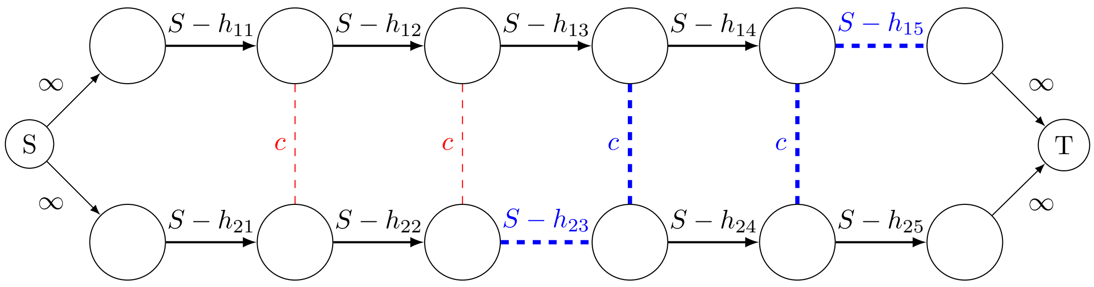
The task is to assign hippos to ponds, which strongly suggests using min-cut. Let's analyse simple scenarios and build up to the full answer.

Consider a single hippo i. We want min cut to choose a single pond, so we describe each pond j as an edge and put them in series from source to sink. The flow will be bottlenecked at the pond with the minimum capacity, but we want it to choose the one with the maximum happiness, so we can set each edge to have capacity $S - h_{ij}$ where S is some number large enough that every capacity is positive. Then the maximum happiness $S - f$.

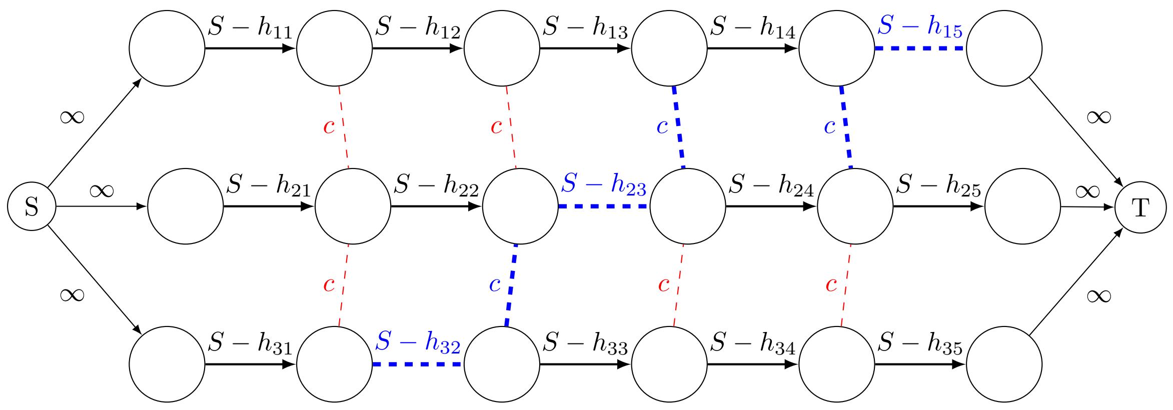


Consider a pair of friendly hippos. We start with the same construction as the single-hippo case, for each hippo. We want to penalise moving the hippos away from each other; we want to add edges that will also need to be cut if the chosen ponds are far from each other. After some experimentation, we settle on adding a perpendicular edge of capacity c between every pair of nodes that are the same distance from the source/sink.

So if the same pond is chosen by both hippos, none of the perpendicular edges are part of the bottleneck. If two adjacent ponds are chosen, then there will be one perpendicular edge that will also be a part of the cut. And so on, with d extra cut-edges with ponds at a distance of d. The total happiness becomes $2S - f$. This approach can also be used to solve the subtask.



For every pair of friendly hippos, we connect perpendicular edges between their chains. If their chosen ponds are a distance of d, then there is cd bottlenecking-flow from source to sink passing from the earlier hippo's chain to the latter. Our happiness is $nS - f$.



There is a complication that arises here if S is too small. Let the hippos be u, v and w, with friendships uv and vw. We could place u and w far away from each other, and place v in both of these ponds (i.e. cut the chain in two places). This means that we pay some $S - h_{bj}$ twice, but we skip paying the constant c possibly many times. To prevent this, make sure that $S > h_{ij} + nmc$.

Reference Solutions

```
// Subtask solution by Yifan

#include <bits/stdc++.h>
#define MAX_SIZE 55
using namespace std;

typedef long long ll;

set<pair<int, int>> pr;
set<int> other;

ll g[MAX_SIZE][MAX_SIZE], C;
int N, M, K;

int main() {
    scanf("%d%d%d%lld", &N, &M, &K, &C);
    for (int i = 1; i <= N; ++i) {
        for (int j = 1; j <= M; ++j) {
            scanf("%lld", &g[i][j]);
        }
        other.insert(i);
    }

    for (int i = 1; i <= K; ++i) {
        int a, b;
        scanf("%d%d", &a, &b);
        pr.emplace(a, b);
        other.erase(a);
        other.erase(b);
    }

    ll ans = 0;
    for (auto &v : other) {
        ll mx = g[v][1];
        for (int i = 1; i <= M; ++i) mx = max(mx, g[v][i]);
        ans += mx;
    }

    for (auto &p : pr) {
        ll mx = 0;
        for (int i = 1; i <= M; ++i) {
            for (int j = 1; j <= M; ++j) {
                mx = max(mx, -C * abs(j - i) + g[p.first][i] + g[p.second][j]);
            }
        }
        ans += mx;
    }
    printf("%lld\n", ans);
}
```

```cpp
// Solution by Yifan

#include <bits/stdc++.h>
#define MAX_SIZE 2611
using namespace std;

typedef long long ll;


struct FlowNetwork {
    int n;
    ll INF = 1e18;
    vector<vector<ll>> adjMat, adjList;
    vector<int> level, uptochild;

    FlowNetwork(int _n) : n(_n) {
        adjMat.resize(n);
        for (int i = 0; i < n; ++i) { adjMat[i].resize(n); }
        adjList.resize(n);
        level.resize(n);
        uptochild.resize(n);
    }

    void addEdge(int u, int v, ll c) {
        if (adjMat[u][v] == 0) {
            adjMat[u][v] += c;
            adjList[u].push_back(v);
            adjList[v].push_back(u);
        }
    }

    void flowEdge(int u, int v, ll c) {
        adjMat[u][v] -= c;
        adjMat[v][u] += c;
    }

    // this constructs the level graph and returns whether sink is still reachable
    bool constructLevelGraph(int s, int t) {
        fill(level.begin(), level.end(), -1);
        queue<int> q;
        q.push(s);
        level[s] = 0;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            uptochild[u] = 0;
            for (int v : adjList[u]) {
                if (level[v] == -1 && adjMat[u][v] > 0) {
                    level[v] = level[u] + 1;
                    q.push(v);
                }
            }
        }
        return level[t] != -1;
    }

    ll augment(int u, int t, ll f) {
        if (u == t) { return f; }
        for (int &i = uptochild[u]; i < (int) adjList[u].size(); i++) {
            int v = adjList[u][i];
            // use only edges in the level graph with remaining capacity
            if (level[v] == level[u] + 1 && adjMat[u][v] > 0) {
                ll revisedFlow = augment(v, t, min(f, adjMat[u][v]));
                if (revisedFlow > 0) {
                    flowEdge(u, v, revisedFlow);
                    return revisedFlow;
                }
            }
        }
        level[u] = -1;
        return 0;
    }
    ll dinic(int s, int t) {
        ll res = 0;
        while (constructLevelGraph(s, t)) {
            while (ll x = augment(s, t, INF)) { res += x; }
        }
        return res;
    }
};


const ll big = 1e9;
const ll verybig = 1e15;

int main() {
    FlowNetwork mf(MAX_SIZE);
    int S = MAX_SIZE - 2, T = MAX_SIZE - 1;
    int N, M, K;
    ll C;
    scanf("%d%d%d%lld", &N, &M, &K, &C);
    for (int i = 1; i <= N; ++i) {
        mf.addEdge(S, (i - 1) * (M + 1) + 1, verybig);
        for (int j = 1; j <= M; ++j) {
            ll d;
            scanf("%lld", &d);
            mf.addEdge((i-1) * (M + 1) + j, (i-1) * (M + 1) + j + 1, big - d);
        }
        mf.addEdge(i * (M + 1), T, verybig);
    }

    for (int i = 1; i <= K; ++i) {
        int a, b;
        scanf("%d%d", &a, &b);
        for (int j = 2; j <= M; ++j) {
            mf.addEdge((a-1) * (M + 1) + j, (b-1) * (M + 1) + j, C);
            mf.addEdge((b-1) * (M + 1) + j, (a-1) * (M + 1) + j, C);
        }
    }
    printf("%lld\n", big * N - mf.dinic(S, T));
}
```