Contest 3 Editorial

COMP4128 21T3

17th November 2021

A1. Champion (subtask)

Algorithm

We use brute force.

Maintain an array, which stores the highest known strength of any active player in each year (initially all zero). For every player, for every year in which they are active, if they are stronger than all earlier players active in this year then update the array element.

This algorithm runs in $O(nm)$ time, which is sufficient for $n, m \leq 5,000$.

Implementation Notes

A similar approach could for each year add all active players' strengths to a `set`, then pick the largest element. However, this runs in $O(nm \log n)$ and therefore gets `TIMELIMIT`.

Reference Solution

```
// Solution by Paula

#include <iostream>
using namespace std;

#define N 100010

// each item in the array holds the current highest rating
int years[N];

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

  for (int i = 1; i <= n; i++) {
    int r, a, b;
    cin >> r >> a >> b;

    for (int t = a; t <= b; t++)
      // if we have found a higher rated person
      if (years[t] < r)
        years[t] = r;
  }

  // for each year, output the highest rating
  for (int i = 1; i <= m; i++)
    cout << years[i] << "\n";
}
```

A2. Champion (full)

Algorithm

Setting Ranges

Perhaps the easiest solution is to recognise that this problem can be reduced to the Setting Ranges problem from the Data Structures II lecture. If we process the players in increasing order of rating, then every new player updates the max in their range of active years. Each of $O(n+m)$ range tree operations takes $O(\log m)$ time, which is sufficient for $n, m \leq 100,000$.

Range Update Point Query

We can also maintain a range tree which handles range updates and point queries, where each node stores the maximum over the range of responsibility. Since range updates only change the value in nodes where the new value is an improvement, no sorting is necessary. Again, each of $O(n+m)$ range tree operations takes $O(\log m)$ time, which is sufficient for $n, m \leq 100,000$.

Multiset

An alternative solution is to process the years from 1 to m, maintaining a multiset of ratings for the active players. Each year, we first add players who became active in this year, then query the maximum, and finally remove players who became inactive after this year. Each player is activated once and deactivated once, for $O(n)$ insertions and deletions to the set each in $O(\log n)$ time. Querying the max for each of m years takes only constant time. This solution is also fast enough for $n, m \leq 100,000$.

Implementation Notes

When using a range tree, take particular care with the bounds. You should allocate at least three times the range of responsibility of the root as the number of range tree nodes.

Global arrays are zero-initialised, so no extra work was required to handle the default value in the range tree solutions. Note however that the multiset solution handles the default value by adding a dummy player of rating 0 active in all years.

If a set is used instead of a multiset, we will not correctly handle cases where there are multiple active players of equal rating. They will be deleted one by one, leading to a `RUN-ERROR`.

Note that if the last line of the multiset solution was changed to `comp.erase(rating[i]);` then we would erase *all* players of this rating, not just one of them, leading to a `WRONG-ANSWER`.

Reference Solutions

```cpp
// Solution by Raveen, by adapting the solution to 'Setting Ranges'

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

const int MAXN = 100100;
const int UNSET = -1;
int lazyset[MAXN*3];
int maxrt[MAXN*3];

void recalculate(int i) {
  if (lazyset[i] != UNSET) maxrt[i] = lazyset[i];
  else maxrt[i] = max(maxrt[i*2], maxrt[i*2+1]);
}

void propagate(int i) {
  if (lazyset[i] == UNSET) return;
  lazyset[i*2] = lazyset[i*2+1] = lazyset[i];
  maxrt[i*2] = maxrt[i*2+1] = lazyset[i];
  lazyset[i] = UNSET;
}

void update(int uL, int uR, int v, int i=1, int cL=0, int cR=MAXN) {
  if (uL == cL && uR == cR) {
    lazyset[i] = maxrt[i] = v;
    return;
  }
  propagate(i);
  int mid = (cL + cR) / 2;
  if (uL < mid) update(uL, min(uR, mid), v, i*2, cL, mid);
  if (uR > mid) update(max(uL, mid), uR, v, i*2+1, mid, cR);
  recalculate(i);
}

int query(int qL, int qR, int i=1, int cL=0, int cR=MAXN) {
  if (qL == cL && qR == cR) {
    return maxrt[i];
  }
  propagate(i);
  int mid = (cL + cR) / 2;
  int ans = -1; // note all values are >= 0 in the question.
  if (qL < mid) ans = max(ans, query(qL, min(qR, mid), i*2, cL, mid));
  if (qR > mid) ans = max(ans, query(max(qL, mid), qR, i*2+1, mid, cR));
  return ans;
}

pair<int, pair<int,int> > players[MAXN];

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

  for (int i = 0; i < n; i++) {
    int r, a, b;
    cin >> r >> a >> b;
    players[i] = make_pair(r,make_pair(a-1,b));
  }

  sort(players, players+n);

  for (int i = 0; i < n; i++) {
    int r = players[i].first;
    int a = players[i].second.first;
    int b = players[i].second.second;
    update(a,b,r);
  }

  for (int j = 0; j < m; j++)
    cout << query(j,j+1) << '\n';
}
```

```cpp
// Solution by Raveen, using a range update point query range tree

#include <iostream>
using namespace std;

const int N = 100100;
int lazy[3*N];
int n, m;

void update(int uL, int uR, int v, int i = 1, int cLeft = 0, int cRight = m) {
  if (uL == cLeft && uR == cRight) {
    lazy[i] = max(lazy[i],v);
    return;
  }
  int mid = (cLeft + cRight) / 2;
  if (uL < mid) update(uL, min(uR, mid), v, i * 2, cLeft, mid);
  if (uR > mid) update(max(uL, mid), uR, v, i * 2 + 1, mid, cRight);
}

int query(int p, int i = 1, int cLeft = 0, int cRight = m) {
  if (cRight - cLeft == 1) {
    return lazy[i];
  }
  int mid = (cLeft + cRight) / 2;
  int ans = lazy[i];
  if (p < mid) ans = max(ans,query(p, i * 2, cLeft, mid));
  else ans = max(ans,query(p, i * 2 + 1, mid, cRight));
  return ans;
}

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

  for (int i = 0; i < n; i++) {
    int r, a, b;
    cin >> r >> a >> b;
    update(a-1,b,r);
  }

  for (int j = 0; j < m; j++)
    cout << query(j) << '\n';
}
```

```cpp
// Solution by Raveen, using a multiset

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

const int N = 100100;
int rating[N];
vector<int> activate[N], deactivate[N];

void process (int i, int r, int a, int b) {
  rating[i] = r;
  activate[a].push_back(i);
  deactivate[b].push_back(i);
}

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

  process(0,0,1,m);
  for (int i = 1; i <= n; i++) {
    int r, a, b;
    cin >> r >> a >> b;
    process(i,r,a,b);
  }

  multiset<int> comp;
  for (int j = 1; j <= m; j++) {
    for (int i : activate[j])
      comp.insert(rating[i]);
    cout << *comp.rbegin() << '\n';
    for (int i : deactivate[j])
      comp.erase(comp.find(rating[i]));
  }
}
```

B1. Partners (subtask)

Algorithm

With a radio range of 2×10^9, the spies can communicate between any pair of junctions. It is clear that the best solution is achieved if one spy walks the entire shortest path, then the other one also walks the same path. In an undirected graph with non-negative edge weights, the shortest path between two specified vertices can be found in $O((n + m) \log n)$ using Dijkstra's algorithm, which is sufficient for $n \leq 200$, $m < 600$. The answer is twice the length of this shortest path.

Implementation Notes

A general principle to follow is that floating-point errors should be avoided wherever possible. Since the coordinates of all junctions are given in integers, we can work out the squared distance along any beam as a long long, and only take square roots at the last step. As it turns out, `sqrt` is exact, so in fact a solution using doubles throughout is also acceptable, and this approach is taken in the reference solution.

The problem specified that no two beams intersected except at an endpoint, i.e. the graph is planar. This is the reason for the unusual bound on m: a planar graph on n vertices has at most $3n - 6$ edges (as a consequence of Euler's formula $V - E + F = 2$). This constraint had no bearing on the solution, but it did make it a little more difficult for us to programmatically check that each test case was valid, and much more difficult to programmatically generate test cases. For a challenge, think about how you might achieve either of these tasks.

Floating-point output

Many students got `WRONG-ANSWER` because of presentation errors. The simplest way to print floating-point numbers to a desired number of decimal places is to use C-style output, e.g. `printf("%.9lf\n", x);`. There are however a couple of common pitfalls using C++-style output. If x is a double, `cout << x << '\n'` prints it with *six* digits of precision by default. This is often insufficient, and in particular if x exceeds 10^6, then it will be printed in scientific notation (yes, really).

- To insist on fixed-point notation (i.e. not scientific notation), use the manipulator `std::fixed` from the `<ios>` header (note: this is included within `<iostream>`).
- To change the number of decimal places, using the mainipulator `std::setprecision()` from the `<iomanip>` header.

So to print x to 9 decimal places using C++-style output, we can write:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main (void) {
  /*
     ...
     */
  cout << fixed << setprecision(9) << x << '\n';
}
```

This information has also been added to the Tips page of the course website.

Reference Solutions

```cpp
// Solution by Raveen

#include <cmath>
#include <cstdio>
#include <iostream>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

typedef pair<double, int> state; // (distance, vertex)

const int N = 220;
const int M = 600;
double x[N], y[N];
vector<int> adj[N];
bool seen[N];

double sqr (double x) { return x*x; }

double dist (int c, int o) { return sqrt(sqr(x[c] - x[o]) + sqr(y[c] - y[o])); }

int main (void) {
  int n, m, _d; // _d ignored, assumed 2e9
  cin >> n >> m >> _d;

  for (int i = 0; i < n; i++)
    cin >> x[i] >> y[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);
  }

  priority_queue<state, vector<state>, greater<state>> pq;

  pq.push(state(0, 0));
  while (!pq.empty()) {
    state cur = pq.top();
    pq.pop();
    double d = cur.first;
    int v = cur.second;
    if (v == n-1) {
      printf("%.9lf\n",2*d);
      return 0;
    }
    if (seen[v]) continue;
    seen[v] = true;

    for (int w : adj[v])
      if (!seen[w])
        pq.emplace(d+dist(v,w),w);
  }

  cout << "-1\n";
}
```

B2. Partners (full)

Algorithm

In the full problem, we have to find the shortest path in an implicit graph. The states are unordered pairs of junctions. Ideally, we should compute the adjacencies on the fly (rather than doing them in advance). From a state $\{i, j\}$, we consider moving a spy from junction i to each neighbouring junction k within d of junction j, and vice versa. Note that if junctions j is within d of junctions i and k, then it is also at most d from any point along the beam being traversed (since circles are convex). The terminal state is $\{n, n\}$.

This problem should be reminiscent of the Climbing problem from the SHortest Paths lecture. The main differences are:

- only one legal starting state, i.e. no super-source required;
- only one legal finishing state, so we need to check that both vertices in the state are n;
- the state is a pair, not a triple, making the implementation a little simpler.

There are $O(n^2)$ vertices. Each beam contributes $O(n)$ edges, so there are $O(nm)$ edges overall. Dijkstra's algorithm runs in $O(E \log V) = O(nm \log n)$, which is easily sufficient for $n \leq 200$, $m < 600$.

Implementation Notes

Make sure the state correctly handles the ordering of vertices. The first reference solution keeps unordered pairs, and swaps them if they ever get out of order. The second reference solution instead keeps unordered pairs and duplicates the relaxation of outgoing edges - be very careful not to introduce bugs when duplicating sections of code.

Reference Solutions

```cpp
// Solution by Raveen

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int N = 2020;
long long x[N], y[N];
vector<int> adj[N];
double seen[N][N];

struct state {
  int id[2];
  double dist;
};

bool operator< (const state &a, const state &b) {
  return a.dist > b.dist;
}

long long dsq;

long long sqr (long long x) { return x*x; }

long long sqdist (int c, int o) { return sqr(x[c] - x[o]) + sqr(y[c] - y[o]); }

bool valid (state a) {
  int c = a.id[0];
  int o = a.id[1];
  return sqdist(c,o) <= dsq;
}

double dist (int c, int o) { return sqrt(sqdist(c,o)); }
```

```
int main (void) {
  int n, m; long long d;
  cin >> n >> m >> d;
  dsq = d*d;

  for (int i = 0; i < n; i++)
    cin >> x[i] >> y[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 i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      seen[i][j] = 1e15;

  state begin;
  begin.id[0] = 0;
  begin.id[1] = 0;
  begin.dist = 0;

  priority_queue<state> pq;
  pq.push(begin);
  while (!pq.empty()) {
    state cur = pq.top();
    pq.pop();

    if (cur.id[0] == n-1 && cur.id[1] == n-1) {
      printf("%.9lf\n",cur.dist);
      return 0;
    }

    for (int j = 0; j < 2; j++)
      for (int i : adj[cur.id[j]])
        if (i == n-1 || i != cur.id[!j]) {
          state tmp = cur;
          tmp.dist += dist(cur.id[j],i);
          tmp.id[j] = i;
          sort(tmp.id, tmp.id+2);

          if (valid(tmp) && seen[tmp.id[0]][tmp.id[1]] > tmp.dist) {
            pq.push(tmp);
            seen[tmp.id[0]][tmp.id[1]] = tmp.dist;
          }
        }
  }

  printf("-1\n");
}
```

```
// Solution by Kevin

#include <cstdio>
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <cmath>
using namespace std;
typedef long double ld;
typedef pair<ld, ld> pdd;
#define x first
#define y second

#define MAXN 2005

pdd ps[MAXN];
int adj[MAXN][MAXN]; // stores distance between connected nodes, -1 if not connected.
ld dist[MAXN][MAXN]; // dist[i][j] stores shortest distance to get Curt on i and Owen on
```

```
    j

struct Node {
    int a, b;
    ld d;
    bool operator<(const Node &oth) const {
        return d > oth.d;
    }
};

ld pdist(pdd a, pdd b) {
    return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

int main () {
    cin.tie(nullptr);
    cin.sync_with_stdio(false);

    // scan in everything
    int n, m;
    ld d;
    cin >> n >> m >> d;
    for (int i = 0; i < n; i++) {
        cin >> ps[i].x >> ps[i].y;
    }
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        adj[a-1][b-1] = adj[b-1][a-1] = 1;
    }

    // run dijkstras
    for (int i = 0; i < MAXN; i++) {
        for (int j = 0; j < MAXN; j++) {
            dist[i][j] = -1;
        }
    }

    priority_queue<Node> pq;
    pq.push({0, 0, 0.0});

    while (!pq.empty()) {
        Node curr = pq.top();
        pq.pop();
        if (dist[curr.a][curr.b] != -1) continue;
        dist[curr.a][curr.b] = curr.d;

        int a = curr.a;
        int b = curr.b;

        // move curt from a to c
        for (int c = 0; c < n; c++) {
            if (c == b && b != n-1) continue;        // no two people on the same
                junction
            if (adj[a][c] == 0) continue;            // exists an edge from a to c
            if (pdist(ps[c], ps[b]) > d) continue;   // both nodes are within range
            if (dist[c][b] != -1) continue;          // we've seen this state before
            pq.push({c, b, curr.d + pdist(ps[a], ps[c])});
        }

        // move owen from b to c
        for (int c = 0; c < n; c++) {
            if (c == a && a != n-1) continue;
            if (adj[b][c] == 0) continue;
            if (pdist(ps[a], ps[c]) > d) continue;
            if (dist[a][c] != -1) continue;
            pq.push({a, c, curr.d + pdist(ps[b], ps[c])});
        }
    }

    if (dist[n-1][n-1] < 0) printf("-1\n");
    else printf("%Lf\n", dist[n-1][n-1]);
}
```

C1. Bot Factory (subtask)

Algorithm

In the subtask, there is only one source station, so the answer is either 0 or 1. The answer is 1 only if a bot can start the source, get upgraded to eventually reach level k, and be marked complete at a sink station. This feels like a graph search, but it may not immediately be obvious how to construct an appropriate graph.

The required technique is known as *state explosion*. Our vertices are pairs (i, l) representing bots at station i and level l. For each belt from station i to station j, for each level l, we place an edge from (i, l) to (j, l). The exception is if j is an l-upgrader, i.e. a level l bot arriving at station j is automatically upgraded to level $l + 1$. In this case we connect the edge from (i, l) to $(j, l + 1)$ instead.

The resulting graph is unweighted, and we need only test whether any vertex (t_i, k) is reachable from $(s_1, 1)$. We can therefore perform any graph search, for example DFS. The number of vertices is nk, and the number of edges is mk, so DFS runs in $O((n + m)k)$ which is sufficient for $n, m \leq 600$ and $k \leq 150$.

Implementation Notes

Once you reach a state (t_i, l), you should end the search to avoid getting a `WRONG-ANSWER` by outputting a value greater than 1.

You may wish to assign each state a single integer ID by mapping (i, l) to $n(l-1)+(i-1)$ to use 0-based indexing.

Reference Solution

```cpp
// Solution by Raveen

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

const int N = 606;
const int M = 606;
const int K = 155;
const int V = N*K;
int n, m, k;

// global arrays are initialised to zero for you
bool seen[V];
vector<int> edges[V];
bool sink[N];
int upgrade[N];

bool dfs(int u) {
  if (seen[u]) return false;
  if (u/n == k-1 && sink[u%n]) return true;
  seen[u] = true;
  for (int v : edges[u])
    if (dfs(v))
      return true;
  return false;
}

int main (void) {
  cin >> n >> m >> k;

  int x, y, z;
  cin >> x >> y >> z;

  if (x > 1) {
    cout << "-1\n";
    return 0;
  }

  int s;
  cin >> s;

  for (int i = 0; i < y; i++) {
    int t;
    cin >> t;
    sink[t-1] = true;
  }

  fill(upgrade,upgrade+N,-1);
  for (int i = 0; i < z; i++) {
    int p, q;
    cin >> p >> q;
    upgrade[p-1] = q-1;
  }

  for (int j = 0; j < m; j++) {
    int a, b;
    cin >> a >> b;
    for (int l = 0; l < k; l++) {
      int u = n*l+a-1;
      int v = n*l+b-1;
      if (upgrade[b-1] == l)
        v += n;
      edges[u].push_back(v);
    }
  }

  cout << (dfs(s-1) ? '1' : '0') << '\n';
}
```

C2. Bot Factory (full)

Algorithm

Now that there may be several source stations, we are looking for the largest number of edge-disjoint paths in the same graph. This problem is solved by max flow!

Create a super-source s, connected to each $(s_i, 1)$ by an edge of capacity 1, since source stations spawn at most one new bot each second. Also create a super-source t, connected from each (t_i, k) by an edge of capacity ∞, as any number of level k bots may be marked complete each second at a sink. The maximum flow in this graph is the answer.

The value of the maximum flow clearly does not exceed x, which is the total outgoing capacity from the source s. As a result, the running time of the Ford-Fulkerson algorithm (or any specialisation such as Edmonds-Karp or Dinitz) is bounded by $O(Ef) = O(mkx)$, which is sufficient for $x < n \leq 600$, $m \leq 600$ and $k \leq 150$.

Implementation Notes

The capacities to the super-sink are *not* 1; they are infinite. This was a common source of error. It is sufficient to instead use capacity n, since at most n bots of level k can arrive at a particular sink each second.

Another common error was to treat the upgrades as optional, i.e. by placing edges from (i, l) to (j, l) even if j is an l-upgrader, but placing an additional edge of infinite capacity from (j, l) to $(j, l + 1)$ to allow any number of upgrades. This doesn't change the answer to the subtask, but it does lead to `WRONG-ANSWER` for the full problem.

Since each edge is duplicated in the residual graph, make sure to allocate space for twice as many edges as were explicitly referred to above.

Any max flow algorithm was sufficient for this problem. The first reference solution uses Dinitz, and the second uses the most basic Ford-Fulkerson (finding augmenting paths by DFS).

Reference Solutions

```
// Solution by Angus, using Dinitz's algorithm

#include <cstdio>
#include <algorithm>
#include <queue>
using namespace std;

const int V = 100000;
const int E = 200000;
const int INF = 1000000;
int start[V];
int succ[E], cap[E], to[E];

int edge_counter = 0;
void add_edge(int u, int v, int c) {
  cap[edge_counter] = c, to[edge_counter] = v;
  succ[edge_counter] = start[u];
  start[u] = edge_counter;
  ++edge_counter;
}

int n, s, t;
int lvl[V];
int nextchld[V];

bool bfs() {
  for (int i = 0; i < n; i++) lvl[i] = -1;
  queue<int> q;
  q.push(s); lvl[s] = 0;
  while (!q.empty()) {
    int u = q.front(); q.pop();
    nextchld[u] = start[u];
    for (int e = start[u]; ~e; e = succ[e]) {
      if (cap[e] > 0) {
        int nxt = to[e];
        if (lvl[nxt] != -1) continue;
        lvl[nxt] = lvl[u] + 1;
        q.push(nxt);
      }
    }
  }
  return lvl[t] != -1;
}

int aug(int u, int cflow) {
  if (u == t) return cflow;
  for (int &i = nextchld[u]; i >= 0; i = succ[i]) {
    if (cap[i] > 0) {
      int nxt = to[i];
      if (lvl[nxt] != lvl[u] + 1) continue;
      int rf = aug(nxt, min(cflow, cap[i]));
      if (rf > 0) {
        cap[i] -= rf;
        cap[i^1] += rf;
        return rf;
      }
    }
  }
  lvl[u] = -1;
  return 0;
}

int mf() {
  int tot = 0;
  while (bfs())
    for (int x = aug(s,INF); x; x = aug(s,INF)) tot+=x;
  return tot;
}

int N, M, K, x, y, z;
int is_source[V], is_sink[V], upgrade[V];
vector<int> adj[V];
```

```
int main() {
    fill(start, start + V, -1);

    scanf("%d%d%d", &N, &M, &K);
    scanf("%d%d%d", &x, &y, &z);
    for (int i = 0; i < x; i++) {
        int a;
        scanf("%d", &a);
        is_source[a] = 1;
    }
    for (int i = 0; i < y; i++) {
        int a;
        scanf("%d", &a);
        is_sink[a] = 1;
    }
    for (int i = 0; i < z; i++) {
        int p, q;
        scanf("%d%d", &p, &q);
        upgrade[p] = q;
    }
    for (int i = 0; i < M; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        adj[a].push_back(b);
    }

    for (int level = 1; level <= K; level++) {
        for (int i = 1; i <= N; i++) {
                int node = i*K+level;
                if (level == 1 && is_source[i]) {
                        add_edge(0, node, 1);
                        add_edge(node, 0, 0);
                }
                if (level == K && is_sink[i]) {
                        add_edge(node, 1, INF);
                        add_edge(1, node, 0);
                }
                if (upgrade[i] == level) {
                        add_edge(node, node+1, INF);
                        add_edge(node+1, node, 0);
                } else {
                        for (int e : adj[i]) {
                                int otherNode = e*K+level;
                                add_edge(node, otherNode, 1);
                                add_edge(otherNode, node, 0);
                        }
                }
        }
    }

    n = N*(K+1)+1;
    s = 0;
    t = 1;

    int flow = mf();

    printf("%d\n", flow);
}
```

```cpp
// Solution by Raveen, using the Ford-Fulkerson algorithm

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

const int N = 606;
const int M = 606;
const int K = 155;
const int V = N*K+2;
const int E = 2*(M*K+N);
int start[V];
int seen[V];
int succ[E], cap[E], to[E];
int n, m, k;

int edge_counter = 0;
void add_edge (int u, int v, int c) {
  cap[edge_counter] = c, to[edge_counter] = v;
  succ[edge_counter] = start[u];
  start[u] = edge_counter;
  ++edge_counter;
}

inline int inv (int e) { return e^1; }

bool augment (int u, int t, int f) {
  if (u == t) return true;
  if (seen[u]) return false;
  seen[u] = true;
  for (int e = start[u]; ~e; e = succ[e])
    if (cap[e] >= f && augment(to[e],t,f)) {
      cap[e] -= f;
      cap[inv(e)] += f;
      return true;
    }
  return false;
}

int max_flow (int s, int t) {
  int res = 0;
  fill(seen,seen+V,0);
  while (augment(s,t,1)) {
    fill(seen,seen+V,0);
    res++;
  }
  return res;
}

int upgrade[N];

int main (void) {
  fill(start,start+V,-1);

  cin >> n >> m >> k;
  int s = n*k, t = n*k+1;

  int x, y, z;
  cin >> x >> y >> z;

  for (int i = 0; i < x; i++) {
    int f;
    cin >> f;
    int v = f-1;
    add_edge(s,v,1);
    add_edge(v,s,0);
  }

  for (int i = 0; i < y; i++) {
    int g;
    cin >> g;
    int v = n*(k-1)+g-1;
```

```
    add_edge(v,t,n);
    add_edge(t,v,0);
  }

  fill(upgrade,upgrade+N,-1);
  for (int i = 0; i < z; i++) {
    int p, q;
    cin >> p >> q;
    upgrade[p-1] = q-1;
  }

  for (int j = 0; j < m; j++) {
    int a, b;
    cin >> a >> b;
    for (int l = 0; l < k; l++) {
      int u = n*l+a-1;
      int v = n*l+b-1;
      if (upgrade[b-1] == l)
        v += n;
      add_edge(u,v,1);
      add_edge(v,u,0);
    }
  }

  cout << max_flow(s,t) << '\n';
}
```