# USACO FEB11 Problem 'lostcows' Analysis

**by Jelle van den Hooff**

The Lost Cows is a complicated graph problem. The problem's roots lie in complexity theory where instead of a farm with pastures and signs we try to synchronize a finite state machine. The cute result in complexity theory is that if a synchronizing sequence exists, then such a sequence can be constructed with length O(N^3).

Here, you are asked to not only find such a sequence but also do so efficiently. The first observation to make is that keeping track of all states (which pastures are currently occupied) is infeasible since 2^N is way too large. A second idea is that no matter what signs you instruct the cows to follow, never will the number of occupied pastures increase. But it can decrease!

A way to decrease the number of occupied pastures is by getting two cows to meet up. Since it is guaranteed that a synchronizing sequence exists for all the N cows, it should certainly be possible to get two cows to end up in the same pasture.

If you only keep track of which pastures the pair occupies there are N choose 2 or N*(N-1)/2 possible states. Consider these states as vertices, and the transitions between states by following signs as edges. Since any pair of vertices can be synchronized, there exists a path from any of the state vertices to the state where both cows are at the barn.

All paths can be precomputed in time O(N^2 M) by doing a 'reverse' BFS from the goal, as N^2 M is the number of transition edges. An implementation detail is finding these N^2 M edges; a solution is to do so implicitly in the BFS. If you consider the N^2 states you might arrive in after a transition, you can then loop through all pairs incoming edges per sign.

Now we can synchronize all the cows by repeatedly merging two cows not on the same pasture; we are almost done. When merging two cows, you know where those cows will end up. But where will the other cows end up? To quickly simulate this we can precompute using dynamic programming where a cow X will end up after Y and Z are merged.

My implementation:
```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

#define foreach(i, v) for (typeof((v).begin()) i = (v).begin(); i != (v).end();
i++)

const int MAXN = 205, MAXM = 205;

int n, m;

// prev[N][M] stores all farms which lead to farm N after following sign M
vector<int> prev[MAXN][MAXM];
// bfs_queue stores states as int pairs when doing the reverse BFS from the barn
```

```
queue< pair<int, int> > bfs_queue;
// sign_to_follow[X][Y] has the sign to folllow when merging X and Y
int sign_to_follow[MAXN][MAXN];
// simulate_merge[Z][X][Y] stores where Z will end up after merging X and Y
unsigned char simulate_merge[MAXN][MAXN][MAXN];
// sign[X][Y] stores where sign Y at pasture X points to
int sign[MAXN][MAXM];

// occupied[X] is true if pasture X is occupied at the current simulation step
bool occupied[MAXN], new_occupied[MAXN];

int main() {
  freopen("lostcows.in", "r", stdin);
  freopen("lostcows.out", "w", stdout);
  ios_base::sync_with_stdio(false);

  // read input
  cin >> n >> m;
  for (int j = 0; j < m; j++)
    for (int i = 0; i < n; i++) {
      cin >> sign[i][j];
      sign[i][j]--;
      prev[sign[i][j]][j].push_back(i);
    }

  // compute sign_to_follow using a BFS
  memset(sign_to_follow, 255, sizeof(sign_to_follow));

  // mark the barn as the end point
  sign_to_follow[0][0] = -2;
  bfs_queue.push(make_pair(0, 0));

  // the barn is the end state, so the cow at pasture i will stay there
  for (int i = 0; i < n; i++)
    simulate_merge[i][0][0] = i;

  while (!bfs_queue.empty()) {
    // a and b are currently occupied
    int a = bfs_queue.front().first;
    int b = bfs_queue.front().second;
    bfs_queue.pop();

    // consider all previously followed signs
    for (int prev_sign = 0; prev_sign < m; prev_sign++) {
      foreach(prev_a_iterator, prev[a][prev_sign])
      foreach(prev_b_iterator, prev[b][prev_sign]) {
        int prev_a = *prev_a_iterator;
        int prev_b = *prev_b_iterator;
        if (prev_a > prev_b)
          swap(prev_a, prev_b);

        if (sign_to_follow[prev_a][prev_b] == -1) {
          sign_to_follow[prev_a][prev_b] = prev_sign;
          bfs_queue.push(make_pair(prev_a, prev_b));
        }
      }
    }

    // if we are not at the barn, precompute the merge simulation
    if (a != 0 || b != 0) {
      int move = sign_to_follow[a][b];
      int next_a = sign[a][move];
      int next_b = sign[b][move];
      if (next_a > next_b)
        swap(next_a, next_b);
      for (int i = 0; i < n; i++)
        simulate_merge[i][a][b] = simulate_merge[sign[i][move]][next_a][next_b];
    }
  }

  // now compute the actual synchronizing sequence
  first = true;
  fill(occupied, occupied + n, true);

  while (true) {
    // a and b will contain a pair to be merged
    int a = -1, b = -1;
    for (int i = 1; i < n; i++)
      if (occupied[i]) {
        if (a == -1)
          a = i;
        else if (b == -1)
          b = i;
      }

    // if no pastures except the barn are occupied, we are done
    if (a == -1)
      break;
    // if only one other pasture is occupied, merge that with the barn
    if (b == -1)
      b = 0;

    // simulate the merge
    fill(new_occupied, new_occupied + n, false);
    for (int i = 0; i < n; i++)
      if (occupied[i])
        new_occupied[simulate_merge[i][a][b]] = true;
    copy(new_occupied, new_occupied + n, occupied);

    // and print the sequence to merge a and b
    while (a != 0 || b != 0) {
      if (a > b)
        swap(a, b);
      int now = sign_to_follow[a][b];

      cout << (now + 1) << "\n";
      a = sign[a][now];
      b = sign[b][now];
    }
  }
}
```