Solution — Knights

1 The problem in a nutshell

Finding the maximum number of edge disjoint paths with specified starting vertices and ending edges in an undirected graph where vertex capacities are bounded.

2 Modelling

We are given an undirected graph with grid structure, whose vertices and edges represent hallway segments and intersections correspondingly. There can be either 0 or 1 knight at each vertex. The knights can move in a way such that each edge is traversed by at most 1 knight and each vertex is traversed by at most C knights. The goal is to maximize the number of knights reaching to the outside, which can be modelled as a vertex incident to all ending segments of the hallways.

Since we are looking for edge disjoint paths, flow is a natural candidate approach. However, we can only apply the maximum flow algorithm on a directed graph with a source and without vertex capacities. So we need to model the following:

The Source Create a special vertex that connects to each knight's starting vertex via a directed edge of capacity 1.

Undirected Edges For an undirected edge $\{u, v\}$, the standard approach is to model it by 2 directed edges $\{u, v\}$ and $\{v, u\}$. However, this seems to be problematic as now two knights can traverse the same edge in different directions. So we need to make the following observation: if two knights traverse edge $\{u, v\}$ in opposite directions and they both make it to the outside, they can also succeed by not traversing $\{u, v\}$ without passing through any additional edges or vertices. To see this, assume the escaping paths for the knights are $s_0, \dots, u_0, u, v, v_0, \dots, t$ and $s_1, \dots, v_1, v, u, u_1, \dots, t$, where t is the vertex modelling the outside, while s_0 and s_1 are the starting vertices for the knights. Clearly, $s_0, \dots, u_0, u, u_1, \dots, t$ and $s_1, \dots, v_1, v, v_0, \dots, t$ are also valid escaping paths.

Vertex Capacity For a vertex v with capacity c, we split it into two vertices v_{in} (receiving all incoming edges to v) and v_{out} (sending all outgoing edges from v), and add a directed edge $\{v_{in}, v_{out}\}$ with capacity c.

3 Algorithm Design

Before we start to implement our solution, let us estimate whether it is fast enough. From the problem description and modelling we learn that our graph has $V = 2 * nm + 2 = 5002$ vertices. Each v_{in} is incident to at most 6 edges, i.e. an incoming one from the source, 4 incoming ones from the neighbours, and an outgoing one to v_{out}. Each v_{out} is incident to at most 4 outgoing edges to the neighbours or the outside. Altogether the number of edges E is bounded by $10nm = 25000$. For the two algorithms introduced in the tutorial, `push_relabel_max_flow` have a complexity of $O(V^3)$, while `edmonds_karp_max_flow` runs in $O(VE^2)$ or $O(VEU)$ when the edge capacities are integers bounded by some constant U. In our case, we have $U = 1$ and $VEU \approx 10^8$, which is just below the time limit. However, note that although `push_relabel_max_flow` has a worse worst-case time complexity here, it outperforms `edmonds_karp_max_flow` for all testsets we have. Try it out!

Now we can focus on the subtask specifications.

First subtask: $C = 2$ It might seem weird that the first subtask asks for $C = 2$ instead of $C = 1$. However, since each edge can be traversed by at most 1 knight and each vertex is incident to 4 edges, there can not be more than 2 knights passing through the same vertex. So if the capacity of edges is set to 1, the vertices obtain a capacity limit of 2 automatically. As a result, one do not need to do the splitting trick to set the vertex capacity for this subtask.

Second subtask: $C = 1$ In addition to the solution for the first subtask, we now need to put a limit of 1 on the capacity of vertices, otherwise there might be more knights managing to escape successfully.

Third subtask: $C \leqslant 4$ Well, since the vertex capacity can be at most 2, we do not have any new test cases to deal with.

4 Full Solution

The implementation is basically constructing the graph and applying maximum flow. Since each vertex is described by 2 coordinates, it will be convenient to have a function that maps the coordinates to integers.

```cpp
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>

// Graph Type with nested interior edge properties for flow algorithms
typedef boost::adjacency_list_traits<boost::vecS, boost::vecS, boost::directedS> traits;
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS, boost::no_property,
    boost::property<boost::edge_capacity_t, long,
        boost::property<boost::edge_residual_capacity_t, long,
            boost::property<boost::edge_reverse_t, traits::edge_descriptor>>>> graph;

typedef traits::vertex_descriptor vertex_desc;
typedef traits::edge_descriptor edge_desc;

// Custom edge adder class, highly recommended
class edge_adder {
  graph &G;

 public:
  explicit edge_adder(graph &G) : G(G) {}

  void add_edge(int from, int to, long capacity) {
    auto c_map = boost::get(boost::edge_capacity, G);
```

```
    auto r_map = boost::get(boost::edge_reverse, G);
    const auto e = boost::add_edge(from, to, G).first;
    const auto rev_e = boost::add_edge(to, from, G).first;
    c_map[e] = capacity;
    c_map[rev_e] = 0; // reverse edge has no capacity!
    r_map[e] = rev_e;
    r_map[rev_e] = e;
  }
};

// mapping coordinates of a vertex to its index in G
int index(int x, int y, int m, int n, bool in) {
  return y + n*x + in*m*n;
}

void solve() {
  int m, n, k, c;
  std::cin >> m >> n >> k >> c;

  graph G(2*m*n);
  edge_adder adder(G);
  // Add special vertices source and sink
  const vertex_desc source = boost::add_vertex(G);
  const vertex_desc sink = boost::add_vertex(G);

  // Configure outgoing edges from the source
  for(int i=0; i<k; ++i){
    int x, y;
    std::cin >> x >> y;
    adder.add_edge(source, index(x,y,m,n,true), 1);
  }

  int dx[4] = {0, 1, 0, -1};
  int dy[4] = {1, 0, -1, 0};
  for(int x=0; x<m; ++x){
    for(int y=0; y<n; ++y){
      // Configure the capcity for vertex (x, y)
      adder.add_edge(index(x,y,m,n,true), index(x,y,m,n,false), c);
      for(int i=0; i<4; ++i){
        int newx = x + dx[i];
        int newy = y + dy[i];
        if(newx>=0 && newx<m && newy>=0 && newy<n)
          // Configure the segments (excluding ending segments)
          adder.add_edge(index(x,y,m,n,false), index(newx,newy,m,n,true), 1);
      }
      // Configure the ending segments
      if(x==0 || x==m-1)
        adder.add_edge(index(x,y,m,n,false), sink, 1);
      if(y==0 || y==n-1)
        adder.add_edge(index(x,y,m,n,false), sink, 1);
    }
  }

  //long flow = boost::push_relabel_max_flow(G, source, sink);
  long flow = boost::edmonds_karp_max_flow(G, source, sink);
  std::cout << flow << std::endl;
}

int main() {
```

```
  int t;
  std::cin >> t;
  while(t --)
    solve();
  return 0;
}
```