Algorithms Lab.

Álvaro Marco
amarco@inf.ethz.ch

June 7, 2016

Contents

1 Basic

Include the line:

```
std::ios_base::sync_with_stdio(false);
```

at the beginning of each program.

To print doubles specifying the precision use `setprecision`, e.g.:

```
cout << setprecision(7) << sum << endl;
```

For CGAL programs, add:

```
set(CMAKE_CXX_FLAGS ‘‘$CMAKE_CXX_FLAGS -std=c++11’’)
```

somewhere in the `CMakeLists.txt` file.

Whenever sets are needed consider `https://judge.inf.ethz.ch/doc/boost/libs/disjoint_sets/disjoint_sets.html`.

Figure 1 shows how the triangulation datastructure works with faces in CGAL. Check slides 19 and 20 in `https://judge.inf.ethz.ch/doc/course/cgal_proximity.pdf` for further information.

1.1 Theory

- A double can store integers until 2^{53}
- Minimum independent set: maximum number of vertices s.t. there is no edge between the vertices (NP-COMPLETE! however OK for bipartite graph)
- Maximum vertex cover: Minimum number of vertices s.t. all the edges of the graph are included. Complementary to Minimum independent set (NP-COMPLETE! however OK for bipartite graph)
- König's theorem: In bipartite graphs, the size of a minimum vertex cover equals the size of a maximum matching. $|MIS| = n - |MVC|$
- $a \leq b \equiv -a \geq -b$
- $\max(f(x)) = -\min(-f(x))$

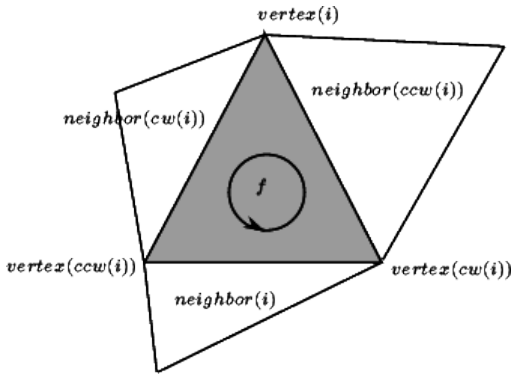

Figure 1: Vertices and neighbor faces in a CGAL triangulation.

2 Complexities

2.1 Boost Graph Library

- Connected components (`connected_components`): $O(V + E)$
- Cycle cancelling (`cycle_canceling`): $O(C \cdot (EV))$ where C is the cost of the initial flow
- Dijkstra (`dijkstra_shortest_paths`): $O(V \log V + E)$
- Kruskal MST (`kruskal_minimum_spanning_tree`): $O(E \log E)$
- Prim MST (`prim_minimum_spanning_tree`): $O(E \log V)$
- Push relabel max flow (`push_relabel_max_flow`): $O(V^3)$
- Strong components (`strong_components`): $O(V + E)$
- Succesive shortest path nonneg... (`successive_shortest_path_nonnegative_weights`): $O(|f|(E + V \log V))$

2.1.1 CGAL

Use Kernels in the following order of preference:

1. `CGAL::Exact_predicates_inexact_constructions_kernel` (constructions use `double`)

2. `CGAL::Exact_predicates_exact_constructions_kernel` (constructions use an exact number type)

3. `CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt`

Try to avoid constructions.

3 Problems

3.1 Greedy & BFS/DFS

3.1.1 Aliens

Keywords— Custom sort

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct interval { int left, right; };

bool equals(const interval &i1, const interval &i2) {
  return i1.left == i2.left && i1.right == i2.right;
}

bool cmp(const interval &i1, const interval &i2) {
  return (i1.left != i2.left)
    ? i1.left < i2.left
    : i1.right > i2.right;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  vector<interval> aliens;
  int T;
  cin >> T;
  while (T--) {
    int n, m;
    cin >> n >> m;

    aliens.clear();
    vector<interval>::iterator it;

    while (n--) {
      interval in;
      cin >> in.left >> in.right;
      if (in.left != 0 || in.right != 0)
        aliens.push_back(in);
    }
```

```
    sort(aliens.begin(), aliens.end(), cmp);
    // check if for each human there exists an alien which
        wound him
    int end_point = 0;
    for (it = aliens.begin(); it != aliens.end(); ++it) {
      if (it->left > end_point +1)
        break;
      else
        end_point = max(end_point, it->right);
    }
    if (end_point < m) {
      cout << 0 << endl;
      continue;
    }
    // counting the superiors (intervals not contained in any
        other interval)
    int superior = 0;
    int rightmost = -1;
    for (unsigned int i = 1; i < aliens.size(); ++i) {
      if (equals(aliens[i - 1], aliens[i])) {
        if (aliens[i - 1].right > rightmost)
          rightmost = aliens[i - 1].right;
        while (i < aliens.size() && equals(aliens[i - 1],
            aliens[i]))
          ++i;
      } else if (aliens[i - 1].right > rightmost) {
        rightmost = aliens[i - 1].right;
        ++superior;
      }
    }
    if (aliens.size() == 1 || aliens[aliens.size() - 1].right >
        rightmost)
      ++superior;
    cout << superior << endl;
  }
  return 0;
}
```

3.1.2 Boats

Keywords— Greedy, Intervals

```
#include <iostream>
#include <limits>
#include <set>
#include <vector>

using namespace std;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int N;
    cin >> N;
    set< pair<int, int> > boats;
    while (N--) {
      int l, p;
      cin >> l >> p;
      boats.insert(make_pair(p, l));
    }
    int current_start = numeric_limits<int>::min();
    int best_deadline = numeric_limits<int>::max();
    int num_boats = 1;
    for (auto it = boats.begin(); it != boats.end(); ++it) {
      int p = it->first;
      int l = it->second;
      if (p >= best_deadline) {
        ++num_boats;
        current_start = best_deadline;
        best_deadline = numeric_limits<int>::max();
      }
      int possible_deadline = max(current_start, p - l) + l;
      best_deadline = min(best_deadline, possible_deadline);
    }
    cout << num_boats << endl;
  }
  return 0;
}
```

3.1.3 Next path

Keywords— BFS

```cpp
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
#include <cstring>

using namespace std;

struct state { int n, d; }; // node where we are, depth

char visited[110];

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    memset(visited, 0, 110);
    vector<int> empty;
    vector< vector<int> > graph(110, empty);
    vector<int>::iterator it;

    int n, m, s, t;
    cin >> n >> m >> s >> t;
    while (m--) {
      int ts, tt;
      cin >> ts >> tt;
      graph[ts].push_back(tt);
    }
    state is;
    is.n = s; // set source
    is.d = 0; // set length of path
    int times = 0; // t reached 0 times
    bool path_found = false;
    queue<state> frontier;
    frontier.push(is);
    while (!frontier.empty()) {
      state current = frontier.front();
      if (current.n == t) {
        ++times;
        if (times == 2) {
          path_found = true;
          cout << current.d << endl;
          break;
        }
      }
      ++current.d;
      frontier.pop();
      for (it = graph[current.n].begin(); it != graph[current.n
          ].end(); ++it) {
        if (visited[*it] < 2) {
          ++visited[*it];
          state tmp;
          tmp.n = *it;
          tmp.d = current.d;
          frontier.push(tmp);
        }
      }
    }
    if (!path_found)
      cout << "no" << endl;
  }
  return 0;
}
```

3.1.4 Race tracks

Keywords— BFS

```cpp
#include <iostream>
#include <algorithm>
#include <queue>
#include <cstring>

using namespace std;

struct node {
  int x, y;   // position
  int sx, sy; // speed
  int d;      // depth
};

int X, Y;
int end_x, end_y;
bool visited[30][30][7][7];
```

```
queue<node> frontier;
int hops;

void clear(queue<node> &q) {
  queue<node> empty;
  swap(q, empty);
}

// just to stay inside the bounds of the array
int speed(int s) {
  if      (s == -1) { s = 4; }
  else if (s == -2) { s = 5; }
  else if (s == -3) { s = 6; }
  return s;
}

void bfs() {
  while (!frontier.empty()) {
    node extracted = frontier.front();
    frontier.pop();
    // insert new states here
    for (int dx = -1; dx != 2; ++dx) {
      for (int dy = -1; dy != 2; ++dy) {
        node tmp = extracted;
        tmp.sx += dx; tmp.sy += dy;
        int newx = tmp.x + tmp.sx, newy = tmp.y + tmp.sy;
        if (tmp.sx >= -3 && tmp.sx <= 3 && tmp.sy >= -3 && tmp.sy <= 3
            && newx >= 0 && newx < X && newy >= 0 && newy < Y
            && !visited[newx][newy][speed(tmp.sx)][speed(tmp.sy)]) {
          visited[newx][newy][speed(tmp.sx)][speed(tmp.sy)] = true;
          ++tmp.d;
          if (newx == end_x && newy == end_y) {
            hops = tmp.d;
            return;
          }
          tmp.x = newx; tmp.y = newy;
          frontier.push(tmp);
        }
      }
    }
  }
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    memset(visited, false, 30 * 30 * 7 * 7);
    clear(frontier);

    node start;
    start.sx = start.sy = start.d = 0;

    int P;
    cin >> X >> Y;
    cin >> start.x >> start.y >> end_x >> end_y;
    cin >> P;
    for (int p = 0; p < P; ++p) {
      int px1, py1, px2, py2;
      cin >> px1 >> py1 >> px2 >> py2;
      for (int i = px1; i <= px2; ++i)
        for (int j = py1; j <= py2; ++j)
          memset(visited[i][j], 1, 7 * 7);
    }
    visited[start.x][start.y][0][0] = true;
    frontier.push(start);
    hops = (start.x == end_x && start.y == end_y) ? 0 : dfs();
    if (hops == -1)
      cout << "No solution." << endl;
    else
      cout << "Optimal solution takes " << hops << " hops." << endl;
  }
  return 0;
}
```

3.1.5 Snippets

Keywords— priority_queue

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

```cpp
#include <limits>
#include <queue>
#include <vector>

using namespace std;

vector<int> times;
vector< vector<int> > positions;

class CompareDist {
public:
  bool operator()(pair<int, int> n1, pair<int, int> n2) {
    return n1.first > n2.first;
  }
};

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m, p;
    times.clear();
    positions.clear();
    cin >> n;
    for (int i = 0; i < n; ++i) {
      cin >> m;
      times.push_back(m);
    }
    for (int i = 0; i < n; ++i) {
      vector<int> newv;
      positions.push_back(newv);
      for (int j = 0; j < times[i]; ++j) {
        cin >> p;
        positions[i].push_back(p);
      }
    }

    int a = positions[0][0], b = positions[0][0];
    // greater<int>
    priority_queue< pair<int, int>, vector< pair<int, int> >,
        CompareDist > current_snippets;
    vector<vector<int>::iterator> iterators;
    for (int i = 0; i < n; ++i) {
      sort(positions[i].begin(), positions[i].end());
      iterators.push_back(positions[i].begin());
      current_snippets.push(make_pair(*iterators[i], i));
      a = min(a, *iterators[i]);
      b = max(b, *iterators[i]);
      ++iterators[i];
    }
    int min_snippet = b - a + 1;
    int pos_first_last = -1; // position of first iterator
        arriving at last position
    while (!current_snippets.empty()) {
      auto top_snippet = current_snippets.top();
      current_snippets.pop();
      int pos = top_snippet.first;
      a = pos;
      if (pos_first_last != -1)
        a = min(a, pos_first_last);
      min_snippet = min(min_snippet, b - a + 1);
      //cout << "[" << a << ", " << b << "]" << endl;
      int owner = top_snippet.second;
      if (iterators[owner] != positions[owner].end()) {
        current_snippets.push(make_pair(*iterators[owner],
            owner));
        b = max(b, *iterators[owner]);
        ++iterators[owner];
      } else if (pos_first_last == -1) {
        pos_first_last = pos;
      }
    }
    cout << min_snippet << endl;
  }
  return 0;
}
```

3.2 CGAL basic

3.2.1 Antenna

Keywords— ceil_to_double, Min_circle_2

```cpp
#include <CGAL/
    Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <iostream>
#include <vector>

typedef CGAL::
    Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;

double ceil_to_double(const K::FT& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n;
    std::cin >> n;
    if (n == 0)
      break;

    std::vector<K::Point_2> citizens;
    while (n--) {
      double x, y;
      std::cin >> x >> y;
      citizens.push_back(K::Point_2(x, y));
    }
    Min_circle mc(citizens.begin(), citizens.end(), true);
    Traits::Circle c = mc.circle();
    K::FT r = sqrt(c.squared_radius());
    std::cout << (long long) ceil_to_double(r) << std::endl;
  }
  return 0;
}
```

3.2.2 Almost antenna

Keywords— Min_circle_2

```cpp
#include <CGAL/
    Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <iostream>
#include <algorithm>
#include <vector>

typedef CGAL::
    Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;

double ceil_to_double(const K::FT& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n;
    std::cin >> n;
    if (n == 0)
      break;

    std::vector<K::Point_2> citizens;
    double x, y;
    while (n--) {
      std::cin >> x >> y;
      citizens.push_back(K::Point_2(x, y));
    }
    bool first = true;
    K::FT min_r;
```

```
    Min_circle mc_all(citizens.begin(), citizens.end(), true);
    for (auto it = mc_all.support_points_begin();
        it != mc_all.support_points_end(); ++it) {
      auto to_remove = std::find(citizens.begin(), citizens.end
          (), *it);
      citizens.erase(to_remove);
      Min_circle mc(citizens.begin(), citizens.end(), true);
      Traits::Circle c = mc.circle();
      auto r = sqrt(c.squared_radius());
      if (first) {
        min_r = r;
        first = false;
      } else if (r < min_r) {
        min_r = r;
      }
      citizens.push_back(*it);
    }
    std::cout << (long long) ceil_to_double(min_r) << std::endl
        ;
  }
  return 0;
}
```

3.2.3 Hiking maps

Keywords— all_of, left_turn

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>
#include <limits>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

K::Point_2 q0, q1, q2, q3, q4, q5;

bool inside_map(K::Point_2 &p) {
  return (CGAL::left_turn(q0, q1, p) || CGAL::collinear(q0, q1,
      p))
      && (CGAL::left_turn(q2, q3, p) || CGAL::collinear(q2, q3,
          p))
      && (CGAL::left_turn(q4, q5, p) || CGAL::collinear(q4, q5,
          p));
}

// WARNING!! modifies the first vector
void vsum(std::vector<int> &a, std::vector<int> &b) {
  for (int i = 0; i < a.size(); ++i)
    a[i] += b[i];
}

void vsub(std::vector<int> &a, std::vector<int> &b) {
  for (int i = 0; i < a.size(); ++i)
    a[i] -= b[i];
}

bool all_contained(std::vector<int> &contained) {
  return std::all_of(contained.begin(), contained.end(), [](int
      i){return i>0;});
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  std::cin >> T;
  while (T--) {
    int m, n;
    std::cin >> m >> n;
    std::vector<K::Point_2> legs;
    legs.reserve(m);
    for (int i = 0; i < m; ++i) {
      int x, y;
      std::cin >> x >> y;
      legs.push_back(K::Point_2(x, y));
    }
    std::vector< std::vector<int> > cont_leg;
    for (int i = 0; i < n; ++i) {
      int x0, y0, x1, y1, x2, y2, x3, y3, x4, y4, x5, y5;
      std::cin >> x0 >> y0 >> x1 >> y1 >> x2 >> y2 >> x3 >> y3
          >> x4 >> y4
          >> x5 >> y5;
      q0 = K::Point_2(x0, y0);
      q1 = K::Point_2(x1, y1);
      q2 = K::Point_2(x2, y2);
      q3 = K::Point_2(x3, y3);
      q4 = K::Point_2(x4, y4);
```

```
      q5 = K::Point_2(x5, y5);

      // orient the points to test with left_turn after
      if (CGAL::right_turn(q0, q1, q2))
        std::swap(q0, q1);
      if (CGAL::right_turn(q2, q3, q4))
        std::swap(q2, q3);
      if (CGAL::right_turn(q4, q5, q0))
        std::swap(q4, q5);

      std::vector<int> contained(m - 1, 0);
      for (int li = 0; li < legs.size() - 1; ++li) {
        if (inside_map(legs[li]) && inside_map(legs[li + 1]))
          contained[li] = 1;
      }
      cont_leg.push_back(contained);
    }
    int i = 0;
    std::vector<int> contained(m - 1, 0);
    int min_size = std::numeric_limits<int>::max();
    for (int j = i; j < n; ++j) {
      vsum(contained, cont_leg[j]);
      while (all_contained(contained)) {
        min_size = std::min(min_size, j - i + 1);
        vsub(contained, cont_leg[i]);
        ++i;
      }
    }
    std::cout << min_size << std::endl;
  }
  return 0;
}
```

3.2.4 Hit

Keywords— do_intersect, Ray_2, Segment_2

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <iostream>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 P;
typedef K::Segment_2 S;

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n;
    std::cin >> n;
    if (n == 0)
      break;

    std::vector<S> segments;
    double x, y, a, b;
    std::cin >> x >> y >> a >> b;
    P p = P(x, y);
    P q = P(a, b);
    auto phileas_ray = K::Ray_2(p, q);
    while (n--) {
      double r, s, t, u;
      std::cin >> r >> s >> t >> u;
      p = P(r, s);
      q = P(t, u);
      S seg = S(p, q);
      segments.push_back(seg);
    }
    bool intersect = false;
    for (auto it = segments.begin(); it != segments.end(); ++it
        ) {
      if (CGAL::do_intersect(phileas_ray, *it)) {
        intersect = true;
        break;
      }
    }
    std::cout << (intersect ? "yes" : "no") << std::endl;
  }
  return 0;
}
```

3.2.5 First hit

Keywords— floor_to_double, intersection

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <iostream>
```

```cpp
#include <vector>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef K::Point_2 P;
typedef K::Segment_2 S;

double floor_to_double(const K::FT& x) {
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n;
    std::cin >> n;
    if (n == 0)
      break;

    std::vector<S> segments;
    double x, y, a, b;
    std::cin >> x >> y >> a >> b;
    p_src = P(x, y);
    q = P(a, b);
    auto phileas_ray = K::Ray_2(p_src, q);
    while (n--) {
      double r, s, t, u;
      std::cin >> r >> s >> t >> u;
      S seg = S(P(r, s), P(t, u));
      segments.push_back(seg);
    }
    bool one_found = false;
    P min_dist_p;
    K::FT min_dist;
    for (auto it = segments.begin(); it != segments.end(); ++it
        ) {
      if (CGAL::do_intersect(phileas_ray, *it)) {
        auto o = CGAL::intersection(phileas_ray, *it);
        K::FT dist;
        P ipoint;
        if (P* op = boost::get<P>(&*o)) {
          dist = CGAL::squared_distance(p_src, *op);
          ipoint = *op;
        } else if (S* os = boost::get<S>(&*o)) {
          if (CGAL::squared_distance(p_src, os->source())
              < CGAL::squared_distance(p_src, os->target())) {
            dist = CGAL::squared_distance(p_src, os->source());
            ipoint = os->source();
          } else {
            dist = CGAL::squared_distance(p_src, os->target());
            ipoint = os->target();
          }
        }
        if (!one_found) {
          min_dist = dist;
          min_dist_p = ipoint;
          one_found = true;
        } else if (dist < min_dist) {
          min_dist = dist;
          min_dist_p = ipoint;
        }
      }
    }
    if (one_found)
      std::cout << (long long) floor_to_double(min_dist_p.x())
          << " "
                << (long long) floor_to_double(min_dist_p.y())
                    << std::endl;
    else
      std::cout << "no" << std::endl;
  }
  return 0;
}
```

3.3 Graphs: Dijkstra & MST

3.3.1 Ant challenge

Keywords— Dijkstra, Kruskal

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/tuple/tuple.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, undirectedS, no_property,
    property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    vector<Graph> species;
    vector<WeightMap> wms;
    vector<int> hives;
    int t, e, s, a, b;
    cin >> t >> e >> s >> a >> b;
    for (int ss = 0; ss < s; ++ss) {
      Graph newg(t);
      species.push_back(newg);
      wms.push_back(get(edge_weight, newg));
    }
    while (e--) {
      int t1, t2;
      cin >> t1 >> t2;
      for (int ss = 0; ss < s; ++ss) {
        Edge e;
        int w;
        cin >> w;
        tie(e, tuples::ignore) = add_edge(t1, t2, species[ss]);
        wms[ss][e] = w;
      }
    }
    for (int ss = 0; ss < s; ++ss) {
      int h;
      cin >> h;
      hives.push_back(h);
    }
    Graph finalg(t);
    WeightMap finalwm;
    for (int ss = 0; ss < s; ++ss) {
      vector<Edge> mst;
      kruskal_minimum_spanning_tree(species[ss], back_inserter(
          mst));
      for (auto ite = mst.begin(); ite != mst.end(); ++ite) {
        Edge e;
        tie(e, tuples::ignore) = add_edge(source(*ite, species[
            ss]),
                                          target(*ite, species[
                                              ss]),
                                          finalg);
        finalwm[e] = wms[ss][*ite];
      }
    }
    vector<int> dist(t);
    vector<int> pred(t);
    dijkstra_shortest_paths(finalg, a,
                            predecessor_map(
                                make_iterator_property_map(pred
                                .begin(),
                            get(vertex_index, finalg)))
                              .distance_map(
                                 make_iterator_property_map(
                                 dist.begin(),
                                get(vertex_index, finalg))
                              )
                           );
    cout << dist[b] << endl;
  }
  return 0;
}
```

3.3.2 First steps BGL

Keywords— Dijkstra, Kruskal

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/tuple/tuple.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, undirectedS, no_property,
    property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m;
    cin >> n >> m;
    Graph G(n);
    WeightMap wm = get(edge_weight, G);
    while (m--) {
      int u, v, c;
      Edge e;
      cin >> u >> v >> c;
      tie(e, tuples::ignore) = add_edge(v, u, G);
      wm[e] = c;
    }
    // Sum of weight of MST
    int w = 0;
    vector<Edge> mst;
    kruskal_minimum_spanning_tree(G, back_inserter(mst));
    for (auto ite = mst.begin(); ite != mst.end(); ++ite)
      w += wm[*ite];
    // distance from node 0 to node furthest from it
    int d = -1;
    vector<int> dist(n);
    vector<int> pred(n);
    dijkstra_shortest_paths(G, 0, predecessor_map(
        make_iterator_property_map(pred.begin(),
                                   get(vertex_index, G)))
                              .distance_map(
                                  make_iterator_property_map(
                                  dist.begin(),
                                 get(vertex_index, G))
                              )
                            );
    graph_traits<Graph>::vertex_iterator vi, vend;
    for (tie(vi, vend) = vertices(G); vi != vend; ++vi)
      d = max(d, dist[*vi]);
    cout << w << " " << d << endl;
  }
  return 0;
}
```

3.3.3 Important bridges

Keywords— Custom sort, Kruskal

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/tuple/tuple.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, undirectedS, no_property,
    property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
```

```
  while (T--) {
    int n, m;
    cin >> n >> m;
    Graph G(n);
    WeightMap wm = get(edge_weight, G);
    while (m--) {
      int e1, e2;
      Edge e;
      cin >> e1 >> e2;
      tie(e, tuples::ignore) = add_edge(e1, e2, G);
      wm[e] = 1;
    }
    vector<Edge> mst;
    kruskal_minimum_spanning_tree(G, back_inserter(mst));
    int num_critical = 0;
    vector< pair<int, int> > critical_bridges;
    for (auto ite = mst.begin(); ite != mst.end(); ++ite) {
      vector<int> component(num_vertices(G));
      int src = source(*ite, G);
      int tgt = target(*ite, G);
      remove_edge(src, tgt, G);
      int num = connected_components(G, &component[0]);
      add_edge(src, tgt, G);
      if (num != 2) { // WARNING this should be 1! It's only 2
          because of by default vertex
        critical_bridges.push_back(make_pair(min(src, tgt), max
            (src, tgt)));
        ++num_critical;
      }
    }
    cout << num_critical << endl;
    sort(critical_bridges.begin(), critical_bridges.end(),
         [](const std::pair<int,int> &left, const std::pair<int
             ,int> &right) {
           return left.first != right.first ?
                  left.first < right.first :
                  left.second < right.second;
    });
    for (auto it = critical_bridges.begin(); it !=
        critical_bridges.end(); ++it)
      cout << it->first << " " << it->second << endl;
  }
  return 0;
}
```

3.3.4 Return of the Jedi

Keywords— Kruskal, Second MST

Check https://judge.inf.ethz.ch/doc/course/return_of_the_jedi.pdf

```
#include <iostream>
#include <algorithm>
#include <limits>
#include <stack>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <boost/tuple/tuple.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, undirectedS, no_property,
    property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

// WARNING! costs is an in-out parameter
void dfs(int src, vector<bool> &visited, Graph &T, WeightMap &
    wm,
     vector<int> &costs, int max_edge) {
  costs[src] = max_edge;
  OutEdgeIt ei, oeiend;
  for (tie(ei, oeiend) = out_edges(src, T); ei != oeiend; ++ei)
       {
    int edge_w = max(max_edge, wm[*ei]);
    Vertex v = target(*ei, T);
    if (!visited[v]) {
      visited[v] = true;
      dfs(v, visited, T, wm, costs, edge_w);
    }
  }
```

```
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, i_tatoo;
    cin >> n >> i_tatoo;

    Graph G(n);
    WeightMap wm = get(edge_weight, G);
    for (int j = 0; j < n-1; ++j) {
      for (int k = j+1; k < n; ++k) {
        int w;
        cin >> w;
        Edge e;
        tie(e, tuples::ignore) = add_edge(j, k, G);
        wm[e] = w;
      }
    }
    // Build MST T
    vector<Edge> mst;
    kruskal_minimum_spanning_tree(G, back_inserter(mst));
    long long mst_cost = 0;
    Graph T(n);
    WeightMap wm_T = get(edge_weight, T);
    for (auto ite = mst.begin(); ite != mst.end(); ++ite) {
      Edge e;
      Vertex u = source(*ite, G);
      Vertex v = target(*ite, G);
      tie(e, tuples::ignore) = add_edge(u, v, T);
      int cost = wm[*ite];
      wm_T[e] = cost;
      mst_cost += cost;
    }
    // DFS on T for each vertex
    vector< vector<int> > max_edge(n); // max edge in the path from v to each
                                       // other vertex in the mst
    for (int v = 0; v < n; ++v) {
      vector<int> costs(n, 0);
      vector<bool> visited(n, false);
      visited[v] = true;
      dfs(v, visited, T, wm_T, costs, 0);
      max_edge[v] = costs;
    }
    long long second_mst = numeric_limits<long long>::max();
    EdgeIt ei, eend;
    for (tie(ei, eend) = edges(G); ei != eend; ++ei) {
      Vertex u = source(*ei, G);
      Vertex v = target(*ei, G);
      // edge(u,v,g) returns pair<edge_descriptor, bool> where the bool is
      // whether the edge exists
      if (edge(u, v, T).second) continue; // if it's in the MST skip it!
      int delta = wm[*ei] - max_edge[u][v];
      second_mst = min(second_mst, mst_cost + delta);
    }
    cout << second_mst << endl;
  }
  return 0;
}
```

3.3.5 Tracking

Keywords— Dijkstra

```
#include <iostream>
#include <algorithm>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/tuple/tuple.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

```

```cpp
// add edge from u to v and from v to u with cost c
void add_edges(Graph &G, WeightMap &wm, int u, int v, int c) {
  Edge e;
  tie(e, tuples::ignore) = add_edge(u, v, G);
  wm[e] = c;
  tie(e, tuples::ignore) = add_edge(v, u, G);
  wm[e] = c;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m, k, x, y;
    cin >> n >> m >> k >> x >> y;
    int nvert = n * (k + 1);
    Graph G(nvert);
    WeightMap wm;
    while (m--) {
      int a, b, c, d;
      cin >> a >> b >> c >> d;
      if (d) { // bridge
        for (int i = 0; i < k; ++i) {
          Edge e;
          tie(e, tuples::ignore) = add_edge(a, b + n, G);
          wm[e] = c;
          tie(e, tuples::ignore) = add_edge(b, a + n, G);
          wm[e] = c;
          a += n;
          b += n;
        }
        add_edges(G, wm, a, b, c);
      } else {
        for (int i = 0; i <= k; ++i) { // for k+1 !!!
          add_edges(G, wm, a, b, c);
          a += n;
          b += n;
        }
      }
    }
    vector<int> dist(nvert);
    vector<int> pred(nvert);
    dijkstra_shortest_paths(G, x,
                            predecessor_map(
                                make_iterator_property_map(pred
                                .begin(),
                            get(vertex_index, G)))
                              .distance_map(
                                 make_iterator_property_map(
                                 dist.begin(),
                                get(vertex_index, G))
                              )
                           );
    cout << dist[y + k*n] << endl;
  }
  return 0;
}
```

3.4 Dynamic programming

3.4.1 Bonus level

Keywords— ATTENTION! this solution gives only 80 points! (100 probably replacing the `map` by a `vector`) check 3.9.1 for 100 points

```cpp
#include <iostream>
#include <algorithm>
#include <map>
#include <vector>

using namespace std;

int n; // size of the board
int D; // number of diagonals in the board
int board[100][100];
map<tuple< int, tuple<int, int> >, int> stored;

int mario(int d, pair<int, int> pos1, pair<int, int> pos2) {
  //cout << d << " [" << pos1.first << ", " << pos1.second << "] [" << pos2.first << ", " <<pos2.second<<"]" << endl;
  if (d == D-1)
    return board[n - 1][n - 2] + board[n - 2][n - 1];

  int max_element = numeric_limits<int>::min();
  auto np1 = make_pair(pos1.first + 1, pos1.second);
  auto np2 = make_pair(pos2.first + 1, pos2.second);
  if (np1.first < n && np2.first < n && np1 < np2) {
    auto key = make_pair(d + 1, make_pair(np1.first, np2.first));
    auto it = stored.find(key);
    int res = it->second;
    if (it == stored.end())
      res = mario(d + 1, np1, np2);
    max_element = max(max_element, res);
  }
  np1 = make_pair(pos1.first + 1, pos1.second);
  np2 = make_pair(pos2.first, pos2.second + 1);
  if (np1.first < n && np2.second < n && np1 < np2) {
    auto key = make_pair(d + 1, make_pair(np1.first, np2.first));
    auto it = stored.find(key);
    int res = it->second;
    if (it == stored.end())
      res = mario(d + 1, np1, np2);
    max_element = max(max_element, res);
  }
  np1 = make_pair(pos1.first, pos1.second + 1);
  np2 = make_pair(pos2.first + 1, pos2.second);
  if (np1.second < n && np2.first < n && np1 < np2) {

    auto key = make_pair(d + 1, make_pair(np1.first, np2.first));
    auto it = stored.find(key);
    int res = it->second;
    if (it == stored.end())
      res = mario(d + 1, np1, np2);
    max_element = max(max_element, res);
  }
  np1 = make_pair(pos1.first, pos1.second + 1);
  np2 = make_pair(pos2.first, pos2.second + 1);
  if (np1.second < n && np2.second < n && np1 < np2) {
    auto np1 = make_pair(pos1.first, pos1.second + 1);
    auto np2 = make_pair(pos2.first, pos2.second + 1);
    auto key = make_pair(d + 1, make_pair(np1.first, np2.first));
    auto it = stored.find(key);
    int res = it->second;
    if (it == stored.end())
      res = mario(d + 1, np1, np2);
    max_element = max(max_element, res);
  }

  int vp1 = board[pos1.first][pos1.second];
  int vp2 = 0;
  if (pos1 != pos2)
    vp2 = board[pos2.first][pos2.second];

  auto key = make_pair(d, make_pair(pos1.first, pos2.first));
  auto value = max_element + vp1 + vp2;
  auto it = stored.find(key);
  if (it == stored.end())
    stored.insert(make_pair(key, value));
  return value;
}

```

```
int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    cin >> n;
    stored.clear();
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        int a;
        cin >> a;
        board[i][j] = a;
      }
    }
    D = 2*n - 2;
    cout << mario(1, make_pair(0, 1), make_pair(1, 0)) + board
        [0][0] + board[n - 1][n - 1] << endl;
  }
  return 0;
}
```

3.4.2 Burning coins from two sides

Keywords— Minimax

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>

#define S 1000

using namespace std;

vector<int> coins;
int stored[S][S];

int max_(int f, int l);

int min_(int f, int l) {
  if (f > l)
    return 0;
  int v1 = (stored[f + 1][l] == -1) ? max_(f + 1, l) : stored[f
      + 1][l];
  int v2 = (stored[f][l - 1] == -1) ? max_(f, l - 1) : stored[f
      ][l - 1];
  stored[f][l] = min(v1, v2);
  return stored[f][l];
}

int max_(int f, int l) {
  if (f > l)
    return 0;

  int v1 = (stored[f + 1][l] == -1) ? min_(f + 1, l) : stored[f
      + 1][l];
  int v2 = (stored[f][l - 1] == -1) ? min_(f, l - 1) : stored[f
      ][l - 1];
  stored[f][l] = max(coins[f] + v1, coins[l] + v2);
  return stored[f][l];
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n;
    cin >> n;
    coins.clear();
    memset(stored, -1, S * S * sizeof(int));
    for (int i = 0; i < n; ++i) {
      int v;
      cin >> v;
      coins.push_back(v);
    }
    cout << max_(0, coins.size() - 1) << endl;
  }
  return 0;
}
```

3.4.3 DHL

```
#include <iostream>
```

```cpp
#include <algorithm>
#include <vector>
#include <cstring>

using namespace std;

vector<int> A;
vector<int> B;

int stored[1005][1005];

int rec_try(int i, int j) {
  int asum = 0;
  for (int a = 0; a < i; ++a)
    asum += A[a];
  int bsum = 0;
  for (int b = 0; b < j; ++b)
    bsum += B[b];

  if (i == 0)
    return A[i] * bsum;
  if (j == 0)
    return asum * B[j];
  int best = asum * bsum;

  asum = 0;
  for (int a = i - 1; a >= 0; --a) {
    asum += A[a];
    int cost = asum * B[j - 1];
    if (stored[a][j - 1] == -1)
      stored[a][j - 1] = rec_try(a, j - 1);
    best = min(best, cost + stored[a][j - 1]);
  }
  bsum = 0;
  for (int b = j - 1; b >= 0; --b) {
    bsum += B[b];
    int cost = A[i - 1] * bsum;
    if (stored[i - 1][b] == -1)
      stored[i - 1][b] = rec_try(i - 1, b);
    best = min(best, cost + stored[i - 1][b]);
  }
  return best;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    memset(stored, -1, sizeof(int) * 1005 * 1005);
    A.clear();
    B.clear();
    int n;
    cin >> n;
    int tmp;
    for (int i = 0; i < n; ++i) {
      cin >> tmp;
      A.push_back(tmp - 1);
    }
    for (int i = 0; i < n; ++i) {
      cin >> tmp;
      B.push_back(tmp - 1);
    }
    cout << rec_try(n, n) << endl;
  }
  return 0;
}
```

3.4.4 Poker chips

Unfortunately only 30 points :(

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstring>

using namespace std;

const int SIZE = 1025;

vector<int> ms;
vector< vector<int> > stacks;

int dp[SIZE][SIZE];

```

```
int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    memset(dp, 0, sizeof(short) * SIZE * SIZE);
    ms.clear();
    stacks.clear();
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
      int m;
      cin >> m;
      ms.push_back(m);
      vector<int> newv;
      stacks.push_back(newv);
    }
    for (int i = 0; i < n; ++i) {
      stacks[i].push_back(-1); // insert unused element just
          for index convenience
      for (int j = 0; j < ms[i]; ++j) {
        int c;
        cin >> c;
        stacks[i].push_back(c);
      }
    }
    for (int i = 1; i <= ms[0]; ++i) {
      for (int j = 1; j <= ms[1]; ++j) {
        int match = stacks[0][i] == stacks[1][j];
        dp[i][j] = max(dp[i-1][j], max(dp[i][j-1], dp[i-1][j-1]
            + match));
      }
    }
    cout << dp[ms[0]][ms[1]] << endl;
  }
  return 0;
}
```

3.4.5 The great game

Keywords— Minimax, max_element, min_element

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>

#define S 50001

using namespace std;

int minstored[S];
int maxstored[S];

int max_(vector< vector<int> > &graph, int src, int tgt);

int min_(vector< vector<int> > &graph, int src, int tgt) {
  if (src == tgt) {
    minstored[src] = 0;
    return minstored[src];
  }

  vector<int> dist;
  for (int v: graph[src]) {
    if (maxstored[v] == -1)
      max_(graph, v, tgt);
    dist.push_back(maxstored[v]);
  }
  minstored[src] = *min_element(begin(dist), end(dist)) + 1;
  return minstored[src];
}

int max_(vector< vector<int> > &graph, int src, int tgt) {
  if (src == tgt) {
    maxstored[src] = 0;
    return maxstored[src];
  }

  vector<int> dist;
  for (int v: graph[src]) {
    if (minstored[v] == -1)
      min_(graph, v, tgt);
    dist.push_back(minstored[v]);
  }
  maxstored[src] = *max_element(begin(dist), end(dist)) + 1;
```

```
  return maxstored[src];
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m, r, b;
    cin >> n >> m >> r >> b;
    vector< vector<int> > graph(n + 1);
    for (int i = 0; i < m; ++i) {
      int u, v;
      cin >> u >> v;
      graph[u].push_back(v);
    }
    memset(minstored, -1, S * sizeof(int));
    memset(maxstored, -1, S * sizeof(int));
    int red_moves = min_(graph, r, n);
    int black_moves = min_(graph, b, n);
    cout << (black_moves < red_moves
        || (black_moves == red_moves && black_moves%2 == 0)) <<
            endl;
  }
  return 0;
}
```

3.5 Network flows

3.5.1 Coin tossing

```
#include <iostream>
#include <algorithm>
#include <map>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
```

```
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m;
    cin >> n >> m;
    vector<int> final_scores(n, 0);
    map<pair<int, int>, int> unknown_rounds;
    for (int mm = 0; mm < m; ++mm) {
      int a, b, c;
      cin >> a >> b >> c;
      switch (c) {
        case 0: {
          int tmp = max(a, b);
          a = min(a, b);
          b = tmp;
          auto p = make_pair(a, b);
          auto it = unknown_rounds.find(p);
          if (it != unknown_rounds.end()) {
            ++unknown_rounds[p];
          } else {
            unknown_rounds.insert(pair<pair<int, int>, int> (p,
                1));
          }
          break;
        }
        case 1:
          --final_scores[a];
          break;
        case 2:
          --final_scores[b];
          break;
      }
    }
    bool impossible = false;
    int s_sum = 0;
    for (int i = 0; i < n; ++i) {
      int s;
      cin >> s;
      final_scores[i] += s;
      s_sum += s;
      if (final_scores[i] < 0)
        impossible = true;
    }
    if (impossible || s_sum != m) {
      cout << "no" << endl;
      continue;
    }
    Graph G(final_scores.size() + unknown_rounds.size());
    EdgeCapacityMap capacity = get(edge_capacity, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResidualCapacityMap res_capacity = get(
        edge_residual_capacity, G);
    EdgeAdder ea(G, capacity, rev_edge);

    Vertex _source = add_vertex(G);
    Vertex sink = add_vertex(G);
    int i = 0;
    for (auto it = unknown_rounds.begin(); it != unknown_rounds
        .end(); ++it) {
      ea.addEdge(_source, i, it->second);
      ea.addEdge(i, it->first.first + unknown_rounds.size(), it
          ->second);
      ea.addEdge(i, it->first.second + unknown_rounds.size(),
          it->second);
      ++i;
    }
    i = unknown_rounds.size();
    for (auto it = final_scores.begin(); it != final_scores.end
        (); ++it) {
      ea.addEdge(i, sink, *it);
      ++i;
    }
    push_relabel_max_flow(G, _source, sink);
    EdgeIt e, eend;
    for (tie(e, eend) = edges(G); e != eend; ++e) {
      if (sink == target(*e, G) && res_capacity[*e] != 0) {
        break;
```

```
      }
    }
    cout << ((e == eend) ? "yes" : "no") << endl;
  }
  return 0;
}
```

3.5.2 Kingdom defence

Keywords— Minimum flow per edge

```
// the tricky thing about this problem is to add a lower bound for the edges in
// the flow algorithm. For this purpose for each edge with a lower bound:
//  * Add an edge from source to the dest of the original edge (weight=lower
//    bound)
//  * Add an edge from source of original edge to sink (weight= lower bound)
//  * The original edge now has weight upper_bound - lower_bound
#include <iostream>
#include <algorithm>
#include <map>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt; // Iterator

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  int T;
  cin >> T;
  while (T--) {
    int l, p; // locations and paths
    cin >> l >> p;

    Graph G(l);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    EdgeAdder ea(G, capacity, rev_edge);

    Vertex _source = add_vertex(G);
    Vertex sink = add_vertex(G);

    int total_flow = 0;
    map<pair<int, int>, int> flows;
    for (int i = 0; i < l; ++i) {
```

```
      int g, d;
      cin >> g >> d;
      total_flow += d;
      auto epair = make_pair(_source, i);
      flows.insert(pair<pair<int, int>, int> (epair, g));
      epair = make_pair(i, sink);
      flows.insert(pair<pair<int, int>, int> (epair, d));
    }
    while (p--) {
      int f, t, min_, max_;
      cin >> f >> t >> min_ >> max_;
      total_flow += min_;
      ea.addEdge(f, t, max_ - min_);

      auto epair = make_pair(_source, t);
      flows[epair] += min_;
      epair = make_pair(f, sink);
      flows[epair] += min_;
    }
    for (auto it = flows.begin(); it != flows.end(); ++it)
      ea.addEdge(it->first.first, it->first.second, it->second)
          ;

    cout << (total_flow == push_relabel_max_flow(G, _source,
        sink) ? "yes" : "no")
         << endl;
  }
  return 0;
}
```

3.5.3 Shopping trip

Keywords— Edge disjoint paths

```
#include <iostream>
#include <algorithm>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt; // Iterator

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
```

```
    int n, m, s;
    cin >> n >> m >> s;
    int tmps = s;
    Graph G(n);
    Vertex sink = add_vertex(G);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    EdgeAdder ea(G, capacity, rev_edge);
    while (tmps--) {
      int store;
      cin >> store;
      ea.addEdge(store, sink, 1);
    }
    while (m--) {
      int u, v;
      cin >> u >> v;
      ea.addEdge(u, v, 1);
      ea.addEdge(v, u, 1);
    }

    long flow = push_relabel_max_flow(G, 0, sink);
    cout << (flow == s ? "yes" : "no") << endl;
  }
  return 0;
}
```

3.5.4 Tetris

Keywords— Vertex capacities

```
#include <iostream>
#include <algorithm>
#include <limits>
#include <vector>
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt; // Iterator

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int w, n;
    cin >> w >> n;
    set< pair<int, int> > pieces;
    int full_width_pieces = 0;
```

```
    for (int i = 0; i < n; ++i) {
      int a, b;
      cin >> a >> b;
      if (a > b)
        swap(a, b);
      if (a == 0 && b == w) // special case of full width piece
        ++full_width_pieces;
      else
        pieces.insert(make_pair(a, b));
    }
    Graph G(2 * w);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResidualCapacityMap res_capacity = get(
        edge_residual_capacity, G);
    EdgeAdder ea(G, capacity, rev_edge);
    for (int i = 1; i < w; ++i) // vertex capacities
      ea.addEdge(i - 1, i + w, 1);
    for (auto p: pieces)
      ea.addEdge(p.first + w, p.second - 1, numeric_limits<int
          >::max());
    long flow = push_relabel_max_flow(G, w, w - 1);
    cout << flow + full_width_pieces << endl;
  }
  return 0;
}
```

3.5.5 The phantom menace

Keywords— Vertex capacities

```
#include <iostream>
#include <algorithm>
#include <map>
#include <queue>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m, s, d;
    cin >> n >> m >> s >> d;
    // Define graph
    Graph G(2 * n);
```

```
    EdgeCapacityMap capacity = get(edge_capacity, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResidualCapacityMap res_capacity = get(
        edge_residual_capacity, G);
    EdgeAdder ea(G, capacity, rev_edge);
    Vertex _source = add_vertex(G);
    Vertex _sink = add_vertex(G);
    for (int i = 0; i < n; ++i) // vertex capacity
      ea.addEdge(i, i + n, 1);
    for (int i = 0; i < m; ++i) {
      int u, v;
      cin >> u >> v;
      ea.addEdge(u + n, v, 1);
    }
    for (int i = 0; i < s; ++i) {
      int src;
      cin >> src;
      ea.addEdge(_source, src, 1);
    }
    for (int i = 0; i < d; ++i) {
      int tgt;
      cin >> tgt;
      ea.addEdge(tgt + n, _sink, 1);
    }
    int flow = push_relabel_max_flow(G, _source, _sink);
    cout << flow << endl;
  }
  return 0;
}
```

3.6 Linear programming

3.6.1 Diet

Keywords— floor_to_double

```
#include <iostream>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

double floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n, m;
    std::cin >> n >> m;
    if (n == 0 && m == 0)
      break;

    Program lp(CGAL::SMALLER, true, 0, false, 0); // all foods
        >= 0
    // nutrients
    for (int i = 0; i < n; ++i) {
      int min_i, max_i;
      std::cin >> min_i >> max_i;
      // ax <= b --> -ax >= -b
```

```
      lp.set_b(2*i, -min_i); // lower bound for nutrient
      lp.set_b(2*i + 1, max_i); // upper bound for nutrient
    }
    // foods
    for (int i = 0; i < m; ++i) {
      int p;
      std::cin >> p;
      lp.set_c(i, p); // min equation price multiplier (min \
          sum_i p_i*f_i)
      for (int j = 0; j < n; ++j) {
        int c;
        std::cin >> c;
        lp.set_a(i, 2*j, -c); // factors for lower bound
            nutrient
        lp.set_a(i, 2*j + 1, c); // factors for upper bound
            nutrient
      }
    }
    Solution s = CGAL::solve_nonnegative_linear_program(lp, ET
        ());
    if (s.is_optimal()) {
      std::cout << (int) floor_to_double(s.objective_value())
          << std::endl;
    } else {
      std::cout << "No such diet." << std::endl;
    }
  }
  return 0;
}
```

3.6.2 Inball

***Keywords*—**

```
#include <iostream>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n, d;
    std::cin >> n;
    if (n == 0)
      break;
    std::cin >> d;

    // Let c be the center of the ball and r the radius. The
        constraints can be
    // phrased as:
    // a^T (x + r*a / ||a||_2 ) <= b
    // The above, in words: a is normal to a^T (i.e.
        perpendicular) and
    // a / ||a||_2 is then the normal unit vector. In addition,
    // a^T a = (||a||_2)^2 . So the constraint finally is :
    // aT x + r *||a||2  <= b
    const int R = d;
    Program lp(CGAL::SMALLER, false, 0, false, 0);
    lp.set_l(R, true, 0); // the radious cannot be negative
    lp.set_c(R, -1); // min -R = max R
    for (int i = 0; i < n; ++i) {
      int sq_sum = 0; // to compute the norm
      for (int j = 0; j < d; ++j) {
        int a;
        std::cin >> a;
        lp.set_a(j, i, a);
        sq_sum += a*a;
      }
      lp.set_a(R, i, sqrt(sq_sum));

      int b;
      std::cin >> b;
      lp.set_b(i, b);
    }
```

```
    Solution s = CGAL::solve_linear_program(lp, ET());
    if (s.is_optimal()) {
      std::cout << (int) CGAL::to_double(-s.objective_value())
          << std::endl;
    } else if (s.is_unbounded()) {
      std::cout << "inf" << std::endl;
    } else {
      std::cout << "none" << std::endl;
    }
  }
  return 0;
}
```

3.6.3 Maximize it

Keywords— Quadratic programming

```
#include <iostream>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

double ceil_to_double(const CGAL::Quotient<ET>& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

double floor_to_double(const CGAL::Quotient<ET>& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int p, a, b;
    std::cin >> p;
    if (p == 0)
      break;
    std::cin >> a >> b;

    const int X = 0;
    const int Y = 1;
    const int Z = 2;
    Program final_qp;
    if (p == 1) {
      // CGAL::SMALLER makes the restrictions <=
      Program qp(CGAL::SMALLER, true, 0, false, 0); // x, y >=
          0
      qp.set_a(X, 0, 1); qp.set_a(Y, 0, 1); qp.set_b(0, 4); //
          x+y <= 4
      qp.set_a(X, 1, 4); qp.set_a(Y, 1, 2); qp.set_b(1, a * b);
           // 4x+2y <= ab
      qp.set_a(X, 2, -1); qp.set_a(Y, 2, 1); qp.set_b(2, 1); //
           -x+y <= 1
      // max f(x) = - min -f(x)
      // -by + ax^2
      qp.set_c(Y, -b); qp.set_d(X, X, 2*a);
      final_qp = qp;
    } else if (p == 2) {
      // CGAL::LARGER makes the restrictions >=
      Program qp(CGAL::LARGER, false, 0, false, 0);
      qp.set_u(X, true, 0); qp.set_u(Y, true, 0); // x, y <= 0
      qp.set_a(X, 0, 1); qp.set_a(Y, 0, 1); qp.set_b(0, -4); //
           xy >= -4
      // 4x+2y+z^2 >= -ab
      qp.set_a(X, 1, 4); qp.set_a(Y, 1, 2); qp.set_a(Z, 1, 1);
          qp.set_b(1, -(a*b));
      qp.set_a(X, 2, -1); qp.set_a(Y, 2, 1); qp.set_b(2, -1);
          // -x*y >= -1
      // min ax^2 + by + z^4
```

```
      qp.set_d(X, X, 2*a); qp.set_c(Y, b); qp.set_d(Z, Z, 2);
      final_qp = qp;
    }
    Solution s = CGAL::solve_quadratic_program(final_qp, ET());
    if (s.is_optimal() && p == 1) {
      std::cout << (int) floor_to_double(-s.objective_value())
          << std::endl;
    } else if (s.is_optimal() && p == 2) {
      std::cout << (int) ceil_to_double(s.objective_value()) <<
           std::endl;
    } else if (s.is_unbounded()) {
      std::cout << "unbounded" << std::endl;
    } else {
      std::cout << "no" << std::endl;
    }
  }
  return 0;
}
```

3.6.4 Portfolios

Keywords— Quadratic programming

```
#include <iostream>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

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

  while (true) {
    int n, m;
    std::cin >> n >> m;
    if (n == 0 && m == 0)
      break;

    const int ret_const = 0; // expected return constraint
    const int cos_const = 1; // cost constraint
    Program qp(CGAL::SMALLER, true, 0, false, 0); // all alphas
        >= 0
    for (int i = 0; i < n; ++i) {
      int c, r;
      std::cin >> c >> r;
      qp.set_a(i, ret_const, r); // component for return lower
          bound inequation
      qp.set_a(i, cos_const, c); // component for cost upper
          bound inequation
    }
    qp.set_r(ret_const, CGAL::LARGER);
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        int v;
        std::cin >> v;
        if (j <= i)
          qp.set_d(i, j, 2*v);
      }
    }
    for (int i = 0; i < m; ++i) {
      int C, R, V;
      std::cin >> C >> R >> V;
      qp.set_b(ret_const, R);
      qp.set_b(cos_const, C);

      Solution s = CGAL::solve_nonnegative_quadratic_program(qp
          , ET());
      std::cout << (s.is_optimal() && CGAL::to_double(s.
          objective_value()) <= V
          ? "Yes." : "No.") << std::endl;
    }
  }
  return 0;
}
```

3.6.5 Portfolios revisited

Keywords— Binary search, Quadratic programming

```cpp
#include <iostream>
#include <limits>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif

typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

double find_result(Solution &s, std::vector<int> &returns) {
  double result = 0;
  int idx = 0;
  for (auto it = s.variable_values_begin(); it !=
       s.variable_values_end(); ++it) {
    result += returns[idx] * CGAL::to_double(*it);
    ++idx;
  }
  return result;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n, m;
    std::cin >> n >> m;
    if (n == 0 && m == 0)
      break;

    std::vector<int> returns(n);
    const int ret_const = 0; // expected return constraint
    const int cos_const = 1; // cost constraint
    Program qp(CGAL::SMALLER, true, 0, false, 0); // all alphas
        >= 0
    for (int i = 0; i < n; ++i) {
      int c, r;
      std::cin >> c >> r;
      returns[i] = r;
      qp.set_a(i, ret_const, r); // component for return lower
          bound inequation
      qp.set_a(i, cos_const, c); // component for cost upper
          bound inequation
    }
    qp.set_r(ret_const, CGAL::LARGER);
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        int v;
        std::cin >> v;
        if (j <= i)
          qp.set_d(i, j, 2*v);
      }
    }
    for (int i = 0; i < m; ++i) {
      int C, V;
      std::cin >> C >> V;
      qp.set_b(cos_const, C);

      double best_result = std::numeric_limits<double>::min();
      int lower = 0;
      int upper = 40 * 1000000;
      while (lower + 1 != upper) {
        int middle = lower + (upper - lower) / 2;
        //std::cout << "[" << lower << ", " << upper << "]" <<
            std::endl;
        //std::cout << " " << middle << std::endl;
        qp.set_b(ret_const, middle);

        Solution s = CGAL::solve_nonnegative_quadratic_program(
            qp, ET());
        if (s.is_optimal() && CGAL::to_double(s.objective_value
            ()) <= V) {
          lower = middle;
          best_result = std::max(best_result, find_result(s,
              returns));
        } else {
          upper = middle;
```

```
        }
      }
      std::cout << best_result << std::endl;
    }
  }
  return 0;
}
```

3.7 CGAL proximity structures

3.7.1 Bistro

Keywords— Triangulation, nearest_vertex Distance to closest point in triangulation.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n, m;
    std::cin >> n;
    if (n == 0)
      break;

    std::vector<K::Point_2> restaurants;
    restaurants.reserve(n);
    while (n--) {
      int x, y;
      std::cin >> x >> y;
      restaurants.push_back(K::Point_2(x, y));
    }
    Triangulation t;
    t.insert(restaurants.begin(), restaurants.end());
    std::cin >> m;
    while (m--) {
      int x, y;
      std::cin >> x >> y;
      auto proposed_loc = K::Point_2(x, y);
      auto nearest_rest = t.nearest_vertex(proposed_loc)->point
          ();
      std::cout << (unsigned long long)
          CGAL::to_double(CGAL::squared_distance(nearest_rest,
              proposed_loc))
          << std::endl;
    }
  }
  return 0;
```

```
}
```

3.7.2 Germs

Keywords—

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>
#include <map>

typedef unsigned int uint;
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;

double ceil_to_double(const K::FT& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n;
    std::cin >> n;
    if (n == 0)
      break;

    // rectangle
    int l, b, r, t;
    std::cin >> l >> b >> r >> t;
    auto top = K::Segment_2(K::Point_2(l, t), K::Point_2(r, t))
        ;
    auto right = K::Segment_2(K::Point_2(r, t), K::Point_2(r, b
        ));
    auto bottom = K::Segment_2(K::Point_2(l, b), K::Point_2(r,
        b));
    auto left = K::Segment_2(K::Point_2(l, t), K::Point_2(l, b)
        );
    // bacteria
    std::vector<K::Point_2> bacteria;
    bacteria.reserve(n);
    std::map<K::Point_2, double> min_time;
    while (n--) {
      int x, y;
      std::cin >> x >> y;
      auto p = K::Point_2(x, y);
      bacteria.push_back(p);
      auto dist = CGAL::min(squared_distance(top, p),
                    CGAL::min(squared_distance(right, p),
                      CGAL::min(squared_distance(bottom, p),
                        squared_distance(left, p))));
      double time_to_border = ceil_to_double(sqrt(sqrt(dist) -
          0.5));
      min_time[p] = time_to_border;
    }
    Triangulation tri;
    tri.insert(bacteria.begin(), bacteria.end());
    for (auto e = tri.finite_edges_begin(); e != tri.
        finite_edges_end(); ++e) {
      auto dist = tri.segment(e).squared_length();
      double time = ceil_to_double(sqrt((sqrt(dist) - 1) / 2));
      auto p1 = tri.segment(e).point(0);
      auto p2 = tri.segment(e).point(1);
      min_time[p1] = std::min(min_time[p1], time);
      min_time[p2] = std::min(min_time[p2], time);
    }
    std::vector<double> times;
    times.reserve(bacteria.size());
    for (auto it = min_time.begin(); it != min_time.end(); ++it
        )
      times.push_back(it->second);
    std::sort(times.begin(), times.end());
    std::cout << (uint) times[0] << " "
        << (uint) times[times.size() / 2] << " "
        << times[times.size() - 1] << std::endl;
  }
  return 0;
}
```

3.7.3 Graypes

Keywords— Minimum distance between all possible pairs of elements in a triangu-

lation.

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;
typedef Triangulation::Edge_iterator Edge_iterator;

double ceil_to_double(const K::FT& x) {
  double a = std::ceil(CGAL::to_double(x));
  while (a < x) a += 1;
  while (a-1 >= x) a -= 1;
  return a;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n;
    std::cin >> n;
    if (n == 0)
      break;

    std::vector<K::Point_2> graypes;
    graypes.reserve(n);
    while (n--) {
      int x, y;
      std::cin >> x >> y;
      graypes.push_back(K::Point_2(x, y));
    }
    Triangulation t;
    t.insert(graypes.begin(), graypes.end());
    bool first = true;
    K::FT min_dist;
    for (auto e = t.finite_edges_begin(); e != t.
        finite_edges_end(); ++e) {
      auto dist = t.segment(e).squared_length();
      if (dist < min_dist || first) {
        first = false;
        min_dist = dist;
      }
    }
    std::cout << (int) ceil_to_double(sqrt(min_dist) / 2 * 100)
        << std::endl;
  }
  return 0;
}
```

3.7.4 H1N1

Keywords— Motion planning, Triangulation DFS

```cpp
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>
#include <map>
#include <queue>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;
typedef Triangulation::Face_handle Face_handle;

int main() {
  std::ios_base::sync_with_stdio(false);
  while (true) {
    int n, m;
    std::cin >> n;
    if (n == 0)
      break;

    std::vector<K::Point_2> infected;
    infected.reserve(n);
    while (n--) {
      int x, y;
      std::cin >> x >> y;
      infected.push_back(K::Point_2(x, y));
    }
    Triangulation t;
    t.insert(infected.begin(), infected.end());
    std::cin >> m;
    while (m--) {
      int x, y;
      double d;
      std::cin >> x >> y >> d;
      auto healthy = K::Point_2(x, y);
```

```
      auto nearest_infected = t.nearest_vertex(healthy)->point
          ();
      // already inside the infection area
      if (CGAL::to_double(CGAL::squared_distance(
          nearest_infected, healthy)) < d) {
        std::cout << "n";
        continue;
      }
      auto face = t.locate(healthy);
      // already outside the infection area
      if (t.is_infinite(face->vertex(0)) || t.is_infinite(face
          ->vertex(1))
          || t.is_infinite(face->vertex(2))) {
        std::cout << "y";
        continue;
      }
      // BFS on the faces of the triangulation starting at the
          one containing
      // the query point. Add neighbors iff the edge between it
           and the current
      // face has squared length >= 4d [(2*sqrt(d))^2]
      std::map<Face_handle, bool> visited;
      bool path_found = false;
      std::queue<Face_handle> q;
      q.push(face);
      while (!q.empty()) {
        auto f = q.front();
        q.pop();
        for (int i = 0; i < 3; ++i) {
          // see slide 19 of CGAL proximity structures
          int i_v1 = i % 3;
          int i_v2 = (i + 1) % 3;
          int i_neighbor = (i + 2) % 3;

          auto p1 = f->vertex(i_v1)->point();
          auto p2 = f->vertex(i_v2)->point();
          if (CGAL::to_double(CGAL::squared_distance(p1, p2))
              >= 4*d) {
            auto neighbor = f->neighbor(i_neighbor);
            if (t.is_infinite(neighbor->vertex(0))
                || t.is_infinite(neighbor->vertex(1))
                || t.is_infinite(neighbor->vertex(2))) {
              path_found = true;
              break;
            }
            if (!visited[neighbor]) {
              visited[neighbor] = true;
              q.push(neighbor);
            }
          }
        }
        if (path_found)
          break;
      }
      if (path_found)
        std::cout << "y";
      else
        std::cout << "n";
    }
    std::cout << std::endl;
  }
  return 0;
}
```

3.7.5 Light the stage

Keywords— Triangulation

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>
#include <limits>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  std::cin >> T;
  while (T--) {
    int m, n;
    std::cin >> m >> n;

    std::vector<K::Point_2> participants;
```

```
    std::vector<int> participant_radius;
    std::vector<K::Point_2> lamps;
    std::vector<int> time_dead(m, std::numeric_limits<int>::max
        ());
    for (int i = 0; i < m; ++i) {
      int x, y, r;
      std::cin >> x >> y >> r;
      participants.push_back(K::Point_2(x, y));
      participant_radius.push_back(r);
    }
    int h;
    std::cin >> h;
    for (int i = 0; i < n; ++i) {
      int x, y;
      std::cin >> x >> y;
      lamps.push_back(K::Point_2(x, y));
    }
    Triangulation t;
    t.insert(lamps.begin(), lamps.end());
    for (int i = 0; i < m; ++i) {
      // check if intersection area > 0
      K::Point_2 nearest_lamp = t.nearest_vertex(participants[i
          ])->point();
      double closest_dist = CGAL::squared_distance(nearest_lamp
          ,
                                                   participants
                                                       [i]);
      double radius_intersect = h + participant_radius[i];
      double radius_intersect2 = radius_intersect *
          radius_intersect;
      if (radius_intersect2 <= closest_dist) // the closest is
          already too far
        continue;
      for (int j = 0; j < n; ++j) {
        double dist = CGAL::squared_distance(participants[i],
            lamps[j]);
        if (radius_intersect2 > dist) {
          time_dead[i] = j;
          break;
        }
      }
    }
    int winner_time = *std::max_element(time_dead.begin(),
        time_dead.end());
    for (int i = 0; i < m; ++i)
      if (time_dead[i] == winner_time) std::cout << i << " ";
    std::cout << std::endl;
  }
  return 0;
}
```

3.8 Minimum cut, Bipartite matching

3.8.1 Algocoon

Keywords— Minimum cut

Check https://judge.inf.ethz.ch/doc/course/algocoon_group.pdf

```
#include <iostream>
#include <algorithm>
#include <queue>
#include <set>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m;
    cin >> n >> m;
    Graph G(n);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResidualCapacityMap res_capacity = get(
        edge_residual_capacity, G);
    EdgeAdder ea(G, capacity, rev_edge);
    while (m--) {
      int a, b, c;
      cin >> a >> b >> c;
      ea.addEdge(a, b, c);
    }

    int best_source = -1;
    int best_sink = -1;
    int min_flow = numeric_limits<int>::max();
    set< pair<int, int> > explored_pairs;
    for (int i = 1; i < n; ++i) {
      int flow = push_relabel_max_flow(G, 0, i);
      if (flow < min_flow) {
        min_flow = flow;
        best_source = 0;
        best_sink = i;
      }
      flow = push_relabel_max_flow(G, i, 0);
```

```
      if (flow < min_flow) {
        min_flow = flow;
        best_source = i;
        best_sink = 0;
      }
    }

    push_relabel_max_flow(G, best_source, best_sink);
    vector<int> vis(n, false);
    vis[best_source] = true;
    std::queue<int> Q;
    Q.push(best_source);
    while (!Q.empty()) {
      const int u = Q.front();
      Q.pop();
      OutEdgeIt ebeg, eend;
      for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++ebeg) {
        const int v = target(*ebeg, G);
        if (res_capacity[*ebeg] == 0 || vis[v]) continue;
        vis[v] = true;
        Q.push(v);
      }
    }

    cout << min_flow << endl;
    cout << count(vis.begin(), vis.end(), true);
    for (int i = 0; i < n; ++i) {
      if (vis[i]) cout << " " << i;
    }
    cout << endl;
  }
  return 0;
}
```

3.8.2 Knights

Keywords— Minimum vertex cover

```
#include <iostream>
#include <algorithm>
#include <limits>
#include <map>
#include <queue>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};
```

```
43
44 void connect(map<pair<int, int>, Vertex> &mapping, EdgeAdder &
       ea, int u, int i, int j) {
45   auto it = mapping.find(make_pair(i, j));
46   if (it != mapping.end()) {
47     Vertex v = it->second;
48     ea.addEdge(u, v, 1);
49   }
50 }
51
52 int main() {
53   int T;
54   cin >> T;
55   while (T--) {
56     int n;
57     cin >> n;
58     map<pair<int, int>, Vertex> mapping;
59     int n_fields = 0;
60     for (int i = 0; i < n; ++i) {
61       for (int j = 0; j < n; ++j) {
62         bool field;
63         cin >> field;
64         if (field) {
65           mapping.insert(make_pair(make_pair(i, j), n_fields));
66           ++n_fields;
67         }
68       }
69     }
70     vector<Vertex> black; // black vertices
71     vector<Vertex> white; // white vertices
72     Graph G(n_fields);
73     EdgeCapacityMap capacity = get(edge_capacity, G);
74     ReverseEdgeMap rev_edge = get(edge_reverse, G);
75     ResidualCapacityMap res_capacity = get(
         edge_residual_capacity, G);
76     EdgeAdder ea(G, capacity, rev_edge);
77     Vertex _source = add_vertex(G);
78     Vertex _sink = add_vertex(G);
79     for (auto it = mapping.begin(); it != mapping.end(); ++it)
         {
80       auto pos = it->first;
81       int i = pos.first;
82       int j = pos.second;
83       Vertex u = it->second;
84       if ((i + j)%2 == 0) { // black
85         black.push_back(u);
86         ea.addEdge(_source, u, 1);
87
88         connect(mapping, ea, u, i - 1, j - 2);
89         connect(mapping, ea, u, i - 2, j - 1);
90         connect(mapping, ea, u, i + 1, j - 2);
91         connect(mapping, ea, u, i + 2, j - 1);
92         connect(mapping, ea, u, i - 1, j + 2);
93         connect(mapping, ea, u, i - 2, j + 1);
94         connect(mapping, ea, u, i + 1, j + 2);
95         connect(mapping, ea, u, i + 2, j + 1);
96       } else { // white
97         white.push_back(u);
98         ea.addEdge(u, _sink, 1);
99       }
100    }
101    push_relabel_max_flow(G, _source, _sink);
102    vector<int> vis(n_fields + 2, false);
103    vis[_source] = true;
104    std::queue<int> Q;
105    Q.push(_source);
106    while (!Q.empty()) {
107      const int u = Q.front();
108      Q.pop();
109      OutEdgeIt ebeg, eend;
110      for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++
           ebeg) {
111        const int v = target(*ebeg, G);
112        if (res_capacity[*ebeg] == 0 || vis[v]) continue;
113        vis[v] = true;
114        Q.push(v);
115      }
116    }
117    int vertex_cover = 0;
118    for (auto it = black.begin(); it != black.end(); ++it)
119      if (!vis[*it]) ++vertex_cover;
120    for (auto it = white.begin(); it != white.end(); ++it)
121      if (vis[*it]) ++vertex_cover;
122    cout << n_fields - vertex_cover << endl;
123  }
124  return 0;
```

```
}
```

3.8.3 New hope

Keywords— Bipartite graph, is_bipartite, Minimum vertex cover

50 points. It only works when there is a unique stormtrooper per command center.

```
#include <iostream>
#include <algorithm>
#include <queue>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/bipartite.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
      property<edge_reverse_t, Traits::edge_descriptor> > > >
        Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::vertex_iterator VertexIt;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;
// for bipartition
typedef std::vector <default_color_type> partition_t;
typedef typename property_map <Graph, vertex_index_t>::type
    index_map_t;
typedef iterator_property_map <partition_t::iterator,
    index_map_t> partition_map_t;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  int T;
  cin >> T;
  while (T--) {
    int k, s, m;
    cin >> k >> s >> m;
    int num_vert = k * s;
    Graph G(num_vert);
    for (int i = 0; i < m; ++i) {
      int u, v, h;
      cin >> u >> v >> h;
      for (int j = 0; j < h; ++j) {
        int x, y;
        cin >> x >> y;
        add_edge(u * s + x, v * s + y, G);
      }
    }
    // build bipartite graph
    partition_t partition(num_vertices (G));
    partition_map_t partition_map(partition.begin(), get(
        vertex_index, G));
    bool is_bip = is_bipartite(G, get(vertex_index, G),
        partition_map);
    Graph bipG(num_vert);
```

```
    EdgeCapacityMap capacity = get(edge_capacity, bipG);
    ReverseEdgeMap rev_edge = get(edge_reverse, bipG);
    ResidualCapacityMap res_capacity = get(
        edge_residual_capacity, bipG);
    EdgeAdder ea(bipG, capacity, rev_edge);
    Vertex _source = add_vertex(bipG);
    Vertex _sink = add_vertex(bipG);
    if (!is_bip) continue;
    vector<Vertex> white;
    vector<Vertex> black;
    VertexIt vit, vend;
    for (boost::tie(vit, vend) = vertices(G); vit != vend; ++
        vit) {
      auto color = get(partition_map, *vit);
      if (color == color_traits<default_color_type>::white()) {
        ea.addEdge(_source, *vit, 1);
        white.push_back(*vit);
      } else {
        ea.addEdge(*vit, _sink, 1);
        black.push_back(*vit);
      }
    }
    for (auto wv: white) {
      OutEdgeIt ebeg, eend;
      for (tie(ebeg, eend) = out_edges(wv, G); ebeg != eend; ++
          ebeg) {
        const int v = target(*ebeg, G);
        ea.addEdge(wv, v, 1);
      }
    }
    for (auto bv: black) {
      OutEdgeIt ebeg, eend;
      for (tie(ebeg, eend) = out_edges(bv, G); ebeg != eend; ++
          ebeg) {
        const int v = target(*ebeg, bipG);
        ea.addEdge(v, bv, 1);
      }
    }
    push_relabel_max_flow(bipG, _source, _sink);
    vector<int> vis(num_vert + 2, false);
    vis[_source] = true;
    std::queue<int> Q;
    Q.push(_source);
    while (!Q.empty()) {
      const int u = Q.front();
      Q.pop();
      OutEdgeIt ebeg, eend;
      for (tie(ebeg, eend) = out_edges(u, bipG); ebeg != eend;
          ++ebeg) {
        const int v = target(*ebeg, bipG);
        if (res_capacity[*ebeg] == 0 || vis[v]) continue;
        vis[v] = true;
        Q.push(v);
      }
    }
    int vertex_cover = 0;
    for (auto it = white.begin(); it != white.end(); ++it)
      if (!vis[*it]) ++vertex_cover;
    for (auto it = black.begin(); it != black.end(); ++it)
      if (vis[*it]) ++vertex_cover;
    cout << num_vert - vertex_cover << endl;
  }
  return 0;
}
```

3.8.4 Satellites

Keywords— Minimum vertex cover, No-source no-sink

```
#include <iostream>
#include <algorithm>
#include <queue>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/edmonds_karp_max_flow.hpp>
#include <boost/tuple/tuple.hpp>

using namespace std;
using namespace boost;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
    property<edge_residual_capacity_t, long,
```

```
      property<edge_reverse_t, Traits::edge_descriptor> > > >
          Graph;
typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity,
      ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), rev_edge(rev_edge) {}

  void addEdge(int u, int v, long c) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    capacity[reverseE] = 0;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  ReverseEdgeMap &rev_edge;
};

int main() {
  int T;
  cin >> T;
  while (T--) {
    int g, s, l;
    cin >> g >> s >> l;
    Graph G(g + s);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResidualCapacityMap res_capacity = get(
        edge_residual_capacity, G);
    EdgeAdder ea(G, capacity, rev_edge);
    Vertex _source = add_vertex(G);
    Vertex _sink = add_vertex(G);
    while (l--) {
      int ground, satellite;
      cin >> ground >> satellite;
      ea.addEdge(ground, g + satellite, 1);
    }
    for (int i = 0; i < g; ++i) {
      ea.addEdge(_source, i, 1);
    }
    for (int i = g; i < g + s; ++i) {
      ea.addEdge(i, _sink, 1);
    }

    push_relabel_max_flow(G, _source, _sink);
    vector<int> vis(g+s+2, false);
    vis[_source] = true;
    std::queue<int> Q;
    Q.push(_source);
    while (!Q.empty()) {
      const int u = Q.front();
      Q.pop();
      OutEdgeIt ebeg, eend;
      for (tie(ebeg, eend) = out_edges(u, G); ebeg != eend; ++
          ebeg) {
        const int v = target(*ebeg, G);
        if (res_capacity[*ebeg] == 0 || vis[v]) continue;
        vis[v] = true;
        Q.push(v);
      }
    }
    // minimum vertex cover is formed by the non visited
        vertices of ground
    // stations (left side of bipartite graph) and visited
        vertices of
    // (right side of bipartite graph) satellites
    int g_prime = 0, s_prime = 0;
    vector<int> result;
    for (int i = 0; i < g; ++i) {
      if (!vis[i]) {
        ++g_prime;
        result.push_back(i);
```

```
      }
    }
    for (int i = g; i < g + s; ++i) {
      if (vis[i]) {
        ++s_prime;
        result.push_back(i - g);
      }
    }
    cout << g_prime << " " << s_prime << endl;
    bool first = true;
    for (auto it = result.begin(); it != result.end(); ++it) {
      if (!first)
        cout << " ";
      cout << *it;
      first = false;
    }
    cout << endl;
  }
  return 0;
}
```

3.9 Min cost max flow

3.9.1 Bonus level

***Keywords*—**

Check https://judge.inf.ethz.ch/doc/course/solution_bonus_level.pdf

```
#include <iostream>
#include <cstdlib>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/
    successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;

typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type EdgeWeightMap
    ;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity, EdgeWeightMap
      &weight, ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), weight(weight), rev_edge(
        rev_edge) {}

  void addEdge(int u, int v, long c, long w) {
```

```
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    weight[e] = w;
    capacity[reverseE] = 0;
    weight[reverseE] = -w;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  EdgeWeightMap &weight;
  ReverseEdgeMap  &rev_edge;
};

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n;
    cin >> n;

    const int EXTRA_COST = 100;
    Graph G(3 * n * n);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    EdgeWeightMap weight = get(edge_weight, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResCapacityMap res_capacity = get(edge_residual_capacity, G
        );
    EdgeAdder ea(G, capacity, weight, rev_edge);
    Vertex _source = add_vertex(G);
    Vertex _sink = add_vertex(G);

    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        int a;
        cin >> a;
        int current = 3 * j + 3 * n * i;
        ea.addEdge(current, current + 1, 1, EXTRA_COST - a);
        ea.addEdge(current, current + 2, 1, EXTRA_COST);
        if (j != n-1) {
          ea.addEdge(current + 1, current + 3, 1, 0);
          ea.addEdge(current + 2, current + 3, 1, 0);
        }
        if (i != n-1) {
          ea.addEdge(current + 1, current + 3 * n, 1, 0);
          ea.addEdge(current + 2, current + 3 * n, 1, 0);
        }
      }
    }
    ea.addEdge(_source, 0, 2, 0);
    ea.addEdge(n * n * 3 - 2, _sink, 1, 0);
    ea.addEdge(n * n * 3 - 1, _sink, 1, 0);

    successive_shortest_path_nonnegative_weights(G, _source,
        _sink);
    int cost = find_flow_cost(G);
    cout << -cost + 2 * (2 * n - 1) * EXTRA_COST << endl;
  }
}
```

3.9.2 Canteen

Keywords—

Check https://judge.inf.ethz.ch/doc/course/canteen.pdf

```
#include <iostream>
#include <cstdlib>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/cycle_canceling.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/
    successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
```

```
                property <edge_weight_t, long> > > > > Graph;

typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type EdgeWeightMap
    ;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity, EdgeWeightMap
      &weight, ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), weight(weight), rev_edge(
        rev_edge) {}

  void addEdge(int u, int v, long c, long w) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    weight[e] = w;
    capacity[reverseE] = 0;
    weight[reverseE] = -w;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  EdgeWeightMap &weight;
  ReverseEdgeMap  &rev_edge;
};

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

  int T;
  cin >> T;
  while (T--) {
    int n;
    cin >> n;

    const int EXTRA_COST = 20;
    Graph G(n);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    EdgeWeightMap weight = get(edge_weight, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResCapacityMap res_capacity = get(edge_residual_capacity, G
        );
    EdgeAdder ea(G, capacity, weight, rev_edge);
    Vertex _source = add_vertex(G);
    Vertex _sink = add_vertex(G);

    for (int i = 0; i < n; ++i) {
      int a, c;
      cin >> a >> c;
      ea.addEdge(_source, i, a, c + EXTRA_COST);
    }
    int total_students = 0;
    for (int i = 0; i < n; ++i) {
      int s, p;
      cin >> s >> p;
      total_students += s;
      ea.addEdge(i, _sink, s, -p + EXTRA_COST);
    }
    for (int i = 0; i < n-1; ++i) {
      int v, e;
      cin >> v >> e;
      ea.addEdge(i, i + 1, v, e);
    }

    //int flow = push_relabel_max_flow(G, _source, _sink);
    //cycle_canceling(G);
    successive_shortest_path_nonnegative_weights(G, _source,
        _sink);
    int flow = 0;
    OutEdgeIt e, eend;
    for (tie(e, eend) = out_edges(vertex(_source, G), G); e !=
        eend; ++e) {
      flow += capacity[*e] - res_capacity[*e];
    }

```

```
    int cost = find_flow_cost(G);
    cout << (flow == total_students ? "possible" : "impossible")
        << " " << flow
        << " " << -cost + flow * 2 * EXTRA_COST << endl;
  }
}
```

3.9.3 Carsharing

Keywords— equal_range

Check https://judge.inf.ethz.ch/doc/course/carsharing.pdf

```
#include <iostream>
#include <limits>
#include <cstdlib>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/cycle_canceling.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/
    successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;

typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type EdgeWeightMap
    ;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::edge_iterator EdgeIt;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity, EdgeWeightMap
      &weight, ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), weight(weight), rev_edge(
        rev_edge) {}

  void addEdge(int u, int v, long c, long w) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    weight[e] = w;
    capacity[reverseE] = 0;
    weight[reverseE] = -w;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  EdgeWeightMap &weight;
  ReverseEdgeMap  &rev_edge;
};

struct Request {
  int s, t; // source, target
  int d, a; // departure, arrival
  int p;    // profit
  Request(int ps, int pt, int pd, int pa, int pp)
      : s(ps), t(pt), d(pd), a(pa), p(pp) {}
};

int vertices_between(int dep_t, int arr_t, vector<int> &times)
    {
  int pos1 = distance(times.begin(), equal_range(times.begin(),
                      times.end(), dep_t).first);
  int pos2 = distance(times.begin(), equal_range(times.begin(),
                      times.end(), arr_t).first);
  return pos2 - pos1;
}

```

```
int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int N, S;
    cin >> N >> S;
    vector<int> sources(S);
    for (int i = 0; i < S; ++i)
      cin >> sources[i];
    vector<Request> requests;
    requests.reserve(N);
    vector< set<int> > station_times(S);
    set<int> set_times;
    set< pair<int, int> > vertices;
    int t_0 = numeric_limits<int>::max();
    int t_max = 0;
    for (int i = 0; i < N; ++i) {
      int s, t, d, a, p;
      cin >> s >> t >> d >> a >> p;
      t_0 = min(t_0, d);
      t_max = max(t_max, a);
      --s; --t;
      requests.push_back(Request(s, t, d, a, p));
      station_times[s].insert(d);
      station_times[t].insert(a);
      vertices.insert(make_pair(s, d));
      vertices.insert(make_pair(t, a));
      set_times.insert(d);
      set_times.insert(a);
    }
    vector<int> all_times(set_times.begin(), set_times.end());
    vector< vector<int> > station_times_v(S);
    for (int i = 0; i < S; ++i)
      station_times_v[i] = vector<int> (station_times[i].begin
          (), station_times[i].end());
    // define the graph
    const int MAX_PROFIT = 100;
    Graph G(vertices.size());
    EdgeCapacityMap capacity = get(edge_capacity, G);
    EdgeWeightMap weight = get(edge_weight, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResCapacityMap res_capacity = get(edge_residual_capacity, G
        );
    EdgeAdder ea(G, capacity, weight, rev_edge);
    Vertex _source = add_vertex(G);
    Vertex _sink = add_vertex(G);
    // offsets
    vector<int> offsets(S);
    int offset = 0;
    for (int i = 0; i < S; ++i) {
      offsets[i] = offset;
      offset += station_times[i].size();
    }
    // source -> first time (or source -> sink if no times)
    for (int i = 0; i < S; ++i) {
      if (station_times_v[i].size() == 0) continue;
      int between = vertices_between(t_0, station_times_v[i].
          front(), all_times);
      ea.addEdge(_source, offsets[i], sources[i], MAX_PROFIT *
          between);
    }
    // connection between times
    for (int i = 0; i < S; ++i) {
      if (station_times_v[i].size() == 0) continue;
      int between = vertices_between(station_times_v[i].back(),
           t_max, all_times);
      ea.addEdge(offsets[i] + station_times[i].size() - 1,
          _sink,
                 numeric_limits<int>::max(), MAX_PROFIT *
                     between);
      for (int j = 0; j < station_times[i].size() - 1; ++j) {
        between = vertices_between(station_times_v[i][j],
            station_times_v[i][j + 1],
                                   all_times);
        ea.addEdge(offsets[i] + j, offsets[i] + j + 1,
            numeric_limits<int>::max(),
                   MAX_PROFIT * between);
      }
    }
    // request connections
    for (auto req: requests) {
      auto src_times = station_times_v[req.s];
      int pos1 = distance(src_times.begin(), equal_range(
          src_times.begin(),
                          src_times.end(), req.d).first);
```

```
      int u = offsets[req.s] + pos1;
      auto tgt_times = station_times_v[req.t];
      int pos2 = distance(tgt_times.begin(), equal_range(
          tgt_times.begin(),
                          tgt_times.end(), req.a).first);
      int v = offsets[req.t] + pos2;
      int between = vertices_between(req.d, req.a, all_times);
      ea.addEdge(u, v, 1, MAX_PROFIT * between - req.p);
    }
    //int flow = push_relabel_max_flow(G, _source, _sink);
    //cycle_canceling(G);
    successive_shortest_path_nonnegative_weights(G, _source,
        _sink);
    int flow = 0;
    OutEdgeIt e, eend;
    for (tie(e, eend) = out_edges(vertex(_source, G), G); e !=
        eend; ++e)
      flow += capacity[*e] - res_capacity[*e];
    int cost = find_flow_cost(G);
    int max_dist = vertices_between(t_0, t_max, all_times);
    cout << flow * max_dist * MAX_PROFIT - cost << endl;
  }
  return 0;
}
```

3.9.4 Real estate market

Keywords— Integer programming

Check https://judge.inf.ethz.ch/doc/course/real_estate_handout.pdf

```
#include <iostream>
#include <cstdlib>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/cycle_canceling.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/
    successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;

typedef property_map<Graph, edge_capacity_t>::type
    EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type EdgeWeightMap
    ;
typedef property_map<Graph, edge_residual_capacity_t>::type
    ResCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type
    ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator OutEdgeIt;

struct EdgeAdder {
  EdgeAdder(Graph & G, EdgeCapacityMap &capacity, EdgeWeightMap
      &weight, ReverseEdgeMap &rev_edge)
    : G(G), capacity(capacity), weight(weight), rev_edge(
        rev_edge) {}

  void addEdge(int u, int v, long c, long w) {
    Edge e, reverseE;
    tie(e, tuples::ignore) = add_edge(u, v, G);
    tie(reverseE, tuples::ignore) = add_edge(v, u, G);
    capacity[e] = c;
    weight[e] = w;
    capacity[reverseE] = 0;
    weight[reverseE] = -w;
    rev_edge[e] = reverseE;
    rev_edge[reverseE] = e;
  }
  Graph &G;
  EdgeCapacityMap &capacity;
  EdgeWeightMap &weight;
  ReverseEdgeMap  &rev_edge;
};

int main() {
```

```
  std::ios_base::sync_with_stdio(false);

  int T;
  cin >> T;
  while (T--) {
    int N, M, S;
    cin >> N >> M >> S;

    const int BUYER = 0;
    const int LAND = N;
    const int STATE = N + M;
    const int DELTA = 100;
    Graph G(N + M + S);
    EdgeCapacityMap capacity = get(edge_capacity, G);
    EdgeWeightMap weight = get(edge_weight, G);
    ReverseEdgeMap rev_edge = get(edge_reverse, G);
    ResCapacityMap res_capacity = get(edge_residual_capacity, G
        );
    EdgeAdder ea(G, capacity, weight, rev_edge);
    Vertex _source = add_vertex(G);
    Vertex _sink = add_vertex(G);

    for (int i = 0; i < S; ++i) {
      int l;
      cin >> l;
      ea.addEdge(STATE + i, _sink, l, 0);
    }
    for (int i = 0; i < M; ++i) {
      int s;
      cin >> s;
      ea.addEdge(LAND + i, STATE + s - 1, 1, 0);
    }
    for (int i = 0; i < N; ++i) {
      ea.addEdge(_source, BUYER + i, 1, 0);
      for (int j = 0; j < M; ++j) {
        int b;
        cin >> b;
        ea.addEdge(BUYER + i, LAND + j, 1, DELTA - b);
      }
    }

    successive_shortest_path_nonnegative_weights(G, _source,
        _sink);
    int flow = 0;
    OutEdgeIt e, eend;
    for (tie(e, eend) = out_edges(vertex(_source, G), G); e !=
        eend; ++e) {
      flow += capacity[*e] - res_capacity[*e];
    }
    int cost = find_flow_cost(G);

    cout << flow << " " << flow * DELTA - cost << endl;
  }
}
```

3.10 Miscellaneous

3.10.1 Buddy selection

Keywords— Max cardinality matching, Set intersection

```cpp
#include <iostream>
#include <algorithm>
#include <set>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/tuple/tuple.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, undirectedS, no_property,
    property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef property_map<Graph, edge_weight_t>::type WeightMap;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, f, c;
    cin >> n >> c >> f;
    Graph G(n);
    vector< set<string> > characteristics;
    WeightMap wm = get(edge_weight, G);
    for (int u = 0; u < n; ++u) {
      string characteristic;
      set<string> newset;
      characteristics.push_back(newset);
      for (int i = 0; i < c; ++i) {
        cin >> characteristic;
        characteristics[u].insert(characteristic);
      }
      for (int v = 0; v < (int) characteristics.size(); ++v) {
        if (u == v) continue;

        set<string> intersect;
        set_intersection(characteristics[u].begin(),
            characteristics[u].end(),
                         characteristics[v].begin(),
                            characteristics[v].end(),
                         inserter(intersect, intersect.begin())
                            );
        if ((int) intersect.size() > f) {
          Edge e;
          tie(e, tuples::ignore) = add_edge(u, v, G);
          wm[e] = (int) intersect.size();
        }
      }
    }
    vector<graph_traits<Graph>::vertex_descriptor> mate(n);
    edmonds_maximum_cardinality_matching(G, &mate[0]);
    graph_traits<Graph>::vertex_iterator vi, vi_end;
    int connections = 0;
    for(tie(vi, vi_end) = vertices(G); vi != vi_end; ++vi)
      if (mate[*vi] != graph_traits<Graph>::null_vertex() && *
          vi < mate[*vi])
        ++connections;

    cout << ((connections == n/2) ? "not optimal" : "optimal")
        << endl;
  }
  return 0;
}
```

3.10.2 Divisor distance

Keywords— Prime numbers

```cpp
#include <iostream>
#include <algorithm>
#include <bitset>
#include <vector>

using namespace std;

#define HIGHER_BOUND 10000010
bitset<HIGHER_BOUND> is_prime;

// NOT USED to solve the problem!! just here for reference :)
```

```cpp
bool is_prime(int n) {
  long long i = 2;
  // 1 is not prime
  if (n <= 1) return false;
  // If the number is even
  if       (n == 2) return true;
  else if (n%i == 0) return false;
  // If the number is odd
  ++i;
  long long root = sqrt(n);
  while (root >= i) {
    if (n%i == 0)
      return false;
    i += 2;
  }
  return true;
}

void prime_sieve() {
  is_prime.set(); // all true;
  is_prime.reset(0); // 0 not prime
  is_prime.reset(1); // 1 not prime
  for (int prime = 2; prime * prime < HIGHER_BOUND; ++prime) {
    if (is_prime[prime]) {
      // all multiples of the prime number are not prime
      for (int multiple = prime + prime; multiple <
          HIGHER_BOUND; multiple += prime)
        is_prime.reset(multiple);
    }
  }
}

// return next smallest prime that divides n
int next_smallest_prime(int n, int current) {
  if (is_prime[n])
    return 1;
  for (int prime = current; prime < n; ++prime)
    if (is_prime[prime] && n%prime == 0)
      return prime;
  return 1;
}

// returns a vector with all the divisors of v
vector<int> divisors(int v) {
  vector<int> dv;
  dv.push_back(v);
  if (v == 1)
    return dv;
  int prime = 2;
  while (!is_prime[v]) {
    prime = next_smallest_prime(v, prime);
    int mdiv = v / prime;
    dv.push_back(mdiv);
    v = mdiv;
  }
  dv.push_back(1);
  return dv;
}

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

  prime_sieve();
  int T;
  cin >> T;
  while (T--) {
    int n, c;
    cin >> n >> c;
    while (c--) {
      int v1, v2;
      cin >> v1 >> v2;
      vector<int> dv1 = divisors(v1);
      vector<int> dv2 = divisors(v2);
      int i = 0;
      int j = 0;
      while (dv1[i] != dv2[j]) {
        if (dv1[i] > dv2[j])
          ++i;
        else
          ++j;
      }
      cout << i + j << endl;
    }
  }
  return 0;
}
```

3.10.3 Monkey island

Keywords— Strongly connected components

```
#include <iostream>
#include <algorithm>
#include <limits>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/strong_components.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, directedS> Graph;
typedef graph_traits<Graph>::edge_descriptor Edge;
typedef graph_traits<Graph>::out_edge_iterator
    out_edge_iterator;

int which_component(vector< set<int> > &components, int elem) {
  for (int i = 0; i < components.size(); ++i) {
    if (components[i].find(elem) != components[i].end())
      return i;
  }
  return -1;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    int n, m;
    cin >> n >> m;
    Graph G(n);
    vector<int> costs;
    for (int i = 0; i < m; ++i) {
      int u, v;
      cin >> u >> v;
      add_edge(u-1, v-1, G);
    }
    for (int i = 0; i < n; ++i) {
      int c;
      cin >> c;
      costs.push_back(c);
    }
    vector<int> scc(n);
    int nscc = strong_components(G,
        make_iterator_property_map(scc.begin(), get(
            vertex_index, G)));
    vector< set<int> > components(nscc);
    for (int i = 0; i < scc.size(); ++i) {
      components[scc[i]].insert(i);
    }
    vector<int> parent_scc(nscc, -1);
    for (int i = 0; i < nscc; ++i) {
      for (auto it = components[i].begin(); it != components[i
          ].end(); ++it) {
        out_edge_iterator oei, oeiend;
        for (tie(oei, oeiend) = out_edges(*it, G); oei !=
            oeiend; ++oei) {
          int t = target(*oei, G);
          if (components[i].find(t) == components[i].end())
            parent_scc[which_component(components, t)] = i;
        }
      }
    }
    // for the SCC with no parents (parent=-1) sum the cost
    int final_cost = 0;
    for (int i = 0; i < nscc; ++i) {
      if(parent_scc[i] == -1) {
        int cheapest = numeric_limits<int>::max();
        for (auto it = components[i].begin(); it != components[
            i].end(); ++it)
          cheapest = min(cheapest, costs[*it]);
        final_cost += cheapest;
      }
    }
    cout << final_cost << endl;
  }
  return 0;
}
```

3.11 Combinations

3.11.1 Clues

Keywords— connected_components, Edge_circulator, is_bipartite, Vertex_circulator

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/bipartite.hpp>
#include <boost/graph/connected_components.hpp>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;
typedef Triangulation::Face_handle Face_handle;
typedef Triangulation::Vertex_handle Vertex_handle;

typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::
    undirectedS> Graph;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  std::cin >> T;
  while (T--) {
    int n, m, r;
    std::cin >> n >> m >> r;
    double r2 = ((double) r) * ((double) r);
    std::vector<K::Point_2> stations;
    stations.reserve(n);
    std::map<K::Point_2, int> vertex_ids;
    for (int i = 0; i < n; ++i) {
      int x, y;
      std::cin >> x >> y;
      auto p = K::Point_2(x, y);
      stations.push_back(p);
      vertex_ids.insert(std::make_pair(p, i));
    }
    // Define triangulation
    Triangulation t;
    t.insert(stations.begin(), stations.end());
    // Define graph
    Graph G(n);
    for (auto vit = t.finite_vertices_begin(); vit != t.
        finite_vertices_end(); ++vit) {
      auto vc = t.incident_vertices(vit); // Vertex_circulator
      do {
        if (CGAL::squared_distance(vit->point(), vc->point())
            <= r2) {
          int u = vertex_ids.find(vit->point())->second;
          int v = vertex_ids.find(vc->point())->second;
          if (u > v)
            add_edge(u, v, G);
        }
      } while (++vc != t.incident_vertices(vit));
    }
    bool interferences = !boost::is_bipartite(G);
    if (!interferences) { // maybe not all needed edges were
        included in the graph
      for (auto vit = t.finite_vertices_begin(); vit != t.
          finite_vertices_end();
          ++vit) {
        auto ec = t.incident_edges(vit); // Edge_circulator
        std::vector<K::Point_2> edge_points;
        do {
          if (t.is_infinite(ec)) continue;
          auto f = ec->first;
          auto p = f->vertex(f->ccw(ec->second))->point();
          for (auto ep: edge_points) {
            if (CGAL::squared_distance(p, ep) <= r2) {
              int u = vertex_ids.find(p)->second;
              int v = vertex_ids.find(ep)->second;
              add_edge(u, v, G);
            }
          }
          edge_points.push_back(p);
        } while (++ec != t.incident_edges(vit));
      }
      interferences = !boost::is_bipartite(G);
    }
    std::vector<int> component(boost::num_vertices(G));
    connected_components(G, &component[0]);
    for (int i = 0; i < m; ++i) {
      int ax, ay, bx, by;
      std::cin >> ax >> ay >> bx >> by;
```

```
      if (interferences) {
        std::cout << "n";
        continue;
      }
      auto holmes = K::Point_2(ax, ay);
      auto watson = K::Point_2(bx, by);
      if (CGAL::squared_distance(holmes, watson) <= r2) {
        std::cout << "y";
        continue;
      }
      auto nearest_h = t.nearest_vertex(holmes)->point();
      int comp_h = component[vertex_ids.find(nearest_h)->second
          ];
      auto nearest_w = t.nearest_vertex(watson)->point();
      int comp_w = component[vertex_ids.find(nearest_w)->second
          ];
      if (CGAL::squared_distance(holmes, nearest_h) <= r2
          && CGAL::squared_distance(watson, nearest_w) <= r2
          && comp_h == comp_w) {
        std::cout << "y";
        continue;
      }
      std::cout << "n";
    }
    std::cout << std::endl;
  }
  return 0;
}
```

3.11.2 Sith

Keywords— Binary search, connected_components, max_element, Triangulation

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>
#include <iostream>
#include <map>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;

typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::
    undirectedS> Graph;

bool conquer(std::vector<K::Point_2> &planets,
    std::map<K::Point_2, int> &vertex_ids, int k, double r2) {
  Graph G;
  Triangulation t;
  t.insert(planets.begin() + k, planets.end());
  for (auto eit = t.finite_edges_begin(); eit != t.
      finite_edges_end(); ++eit) {
    auto f = eit->first;
    auto p1 = f->vertex(f->cw(eit->second))->point();
    auto p2 = f->vertex(f->ccw(eit->second))->point();
    if (CGAL::squared_distance(p1, p2) <= r2)
      add_edge(vertex_ids[p1], vertex_ids[p2], G);
  }
  if (boost::num_vertices(G) == 0) return false;
  std::vector<int> component(boost::num_vertices(G));
  int ncomp = connected_components(G, &component[0]);
  std::vector<int> n_elements(ncomp, 0);
  for (int i = 0; i < component.size(); ++i)
    ++n_elements[component[i]];
  return *max_element(n_elements.begin(), n_elements.end()) >=
      k;
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  std::cin >> T;
  while (T--) {
    int n, r;
    std::cin >> n >> r;
    double r2 = ((double) r) * ((double) r);
    std::vector<K::Point_2> planets;
    planets.reserve(n);
    std::map<K::Point_2, int> vertex_ids;
    for (int i = 0; i < n; ++i) {
      int x, y;
      std::cin >> x >> y;
      auto p = K::Point_2(x, y);
      planets.push_back(p);
      vertex_ids.insert(std::make_pair(p, i));
```

```
    }
    int upper = n / 2;
    int lower = 1;
    int k = 1;
    while (lower <= upper) {
      int middle = (upper - lower) / 2 + lower;
      //std::cout << "["<< lower <<", "<<upper<<"] " << middle
          << std::endl;
      if (conquer(planets, vertex_ids, middle, r2)) {
        k = middle;
        lower = middle + 1;
      } else {
        upper = middle - 1;
      }
    }
    std::cout << k << std::endl;
  }
  return 0;
}
```

3.11.3 Stamps

Keywords— Linear programming, do_intersect

Check https://judge.inf.ethz.ch/doc/course/stamps.pdf

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <iostream>
#include <vector>

typedef CGAL::Gmpq ET;
typedef CGAL::Quadratic_program<double> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  std::cin >> T;
  while (T--) {
    int l, s, w;
    std::cin >> l >> s >> w;

    Program lp(CGAL::SMALLER, true, 1, true, 4096); // 2^12 =
        4096
    std::vector<K::Point_2> lamps;
    lamps.reserve(l);
    std::vector<K::Point_2> stamps;
    stamps.reserve(s);
    std::vector<K::Segment_2> walls;
    walls.reserve(w);
    for (int i = 0; i < l; ++i) {
      int x, y;
      std::cin >> x >> y;
      lamps.push_back(K::Point_2(x, y));
    }
    for (int j = 0; j < s; ++j) {
      int x, y, M;
      std::cin >> x >> y >> M;
      stamps.push_back(K::Point_2(x, y));
      lp.set_b(2 * j, -1); // \sum{ p_i * r } >= 1
      lp.set_b(2 * j + 1, M); // \sum{ p_i * r } <= M
    }
    for (int k = 0; k < w; ++k) {
      int a, b, c, d;
      std::cin >> a >> b >> c >> d;
      walls.push_back(K::Segment_2(K::Point_2(a, b), K::Point_2
          (c, d)));
    }
    for (int j = 0; j < s; ++j) {
      for (int i = 0; i < l; ++i) {
        auto tmp_seg = K::Segment_2(lamps[i], stamps[j]);
        bool no_wall_between = true;
        for (int k = 0; k < w; ++k) {
          if (CGAL::do_intersect(tmp_seg, walls[k])) {
            no_wall_between = false;
            break;
          }
        }
        if (no_wall_between) {
          double dist = CGAL::squared_distance(lamps[i], stamps
              [j]);
          lp.set_a(i, 2 * j, -1 / dist);
```

```
          lp.set_a(i, 2 * j + 1, 1 / dist);
        }
      }
    }
    Solution sol = CGAL::solve_linear_program(lp, ET());
    std::cout << (sol.is_infeasible() ? "no" : "yes") << std::
        endl;
  }
  return 0;
}
```

3.11.4 The empire strikes back

Keywords— Linear programming, Triangulation

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <iostream>
#include <vector>
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>

typedef CGAL::Gmpq ET;
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Delaunay_triangulation_2<K> Triangulation;
typedef CGAL::Quadratic_program<ET> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  std::cin >> T;
  while (T--) {
    int a, s, b, e;
    std::cin >> a >> s >> b >> e;
    Program lp(CGAL::LARGER, true, 0, false, 0); // all
        energies >= 0
    // asteroids
    std::vector<K::Point_2> asteroids(a);
    for (int i = 0; i < a; ++i) {
      int x, y, d;
      std::cin >> x >> y >> d;
      lp.set_b(i, d);
      asteroids[i] = K::Point_2(x, y);
    }
    // laser shots
    std::vector<K::Point_2> shots(s);
    for (int var = 0; var < s; ++var) {
      int x, y;
      std::cin >> x >> y;
      lp.set_c(var, 1);
      shots[var] = K::Point_2(x, y);
    }
    // bounty hunters
    std::vector<K::Point_2> hunters(b);
    for (int i = 0; i < b; ++i) {
      int x, y;
      std::cin >> x >> y;
      hunters[i] = K::Point_2(x, y);
    }
    Triangulation t;
    t.insert(hunters.begin(), hunters.end());
    for (int var = 0; var < s; ++var) {
      ET r2 = -1;
      if (b > 0) { // if there are hunters, find the closest
          one!
        auto nearest_hunter = t.nearest_vertex(shots[var])->
            point();
        r2 = CGAL::squared_distance(shots[var], nearest_hunter)
            ;
      }
      for (int i = 0; i < a; ++i) {
        ET dist = CGAL::squared_distance(shots[var], asteroids[
            i]);
        if (r2 == -1 || dist <= r2)
          lp.set_a(var, i, ET(1) / CGAL::max(ET(1), dist));
      }
    }
    Solution sol = CGAL::solve_nonnegative_linear_program(lp,
        ET());
    std::cout << (sol.is_optimal() && sol.objective_value() <=
        e ? "y" : "n")
        << std::endl;
}
```

```
65    return 0;
66  }
```

Index