# 3725

## Description

At there seemed to be two ways of tackling this problem:

* Using the Floyd-Warshall Algorithm to find the shortest paths between each pizza delievery, and then iterating through all permutations of pizza delivery order.
* Using a state-based Dijkstra’s shortest path search, expanding every path.

An attempt was made to complete the second option, but this ultimately failed, as it was timing out from too many nodes being expanded, due to the lack of visited list to keep track of visited nodes. A visited list could not be implemented easily if the deliveries that have been made were stored in a STD collection.

To solve the excessive node expansion, the Floyd-Warshall algorithm was applied to the graph, so that, the Dijkstra path search algorithm, could expand direct routes to remaining deliveries. This only improved the situation a little.

This meant that a visited list had to be implemented. The state information had to be stored in a integer, so it can be used as an array index for the visited list. This lead to the idea of using the bits of an integer to be the flag that indicates whether a delivery has been made. The number of 2048 allows 11 bits to be set (10 for the delivery and 1 for the Pizzeria).

Using bitwise operation, this number can be manipulated to store and retrieve the necessary information from each bit.

Finally, with the combination of the Floyd-Warshall and integer based flags, the Dijkstra’s algorithm has been optimised to not expand any visited states; a state being any node with equivalent current delivery location and list of undelivered location. Also, any paths to locations that have already been delivered to, will not be expanded.

## Implemented Solution

#include <cstdio>

#include <vector>

#include <set>

#include <algorithm>

#include <queue>

#include <functional>

#include <iostream>

using namespace std;

const int MAXN = 11;

vector<vector<int> > g(MAXN, vector<int>(MAXN));

vector<bool> visited(MAXN \* 2048);

int N;

void floyd() {

for (int i = 0; i < N + 1; i++)

for (int u = 0; u < N + 1; u++)

for (int v = 0; v < N + 1; v++)

g[u][v] = min(g[u][v], g[u][i] + g[i][v]);

}

long long solve() {

priority\_queue < pair<long long, pair<int, int> >, vector <pair<long long, pair<int, int> > >, greater < pair<long long, pair<int, int > > > > pq;

for (pq.push(make\_pair(0, make\_pair(0, 0))); !pq.empty(); pq.pop()) {

long long d = pq.top().first;

int u = pq.top().second.first;

int vm = pq.top().second.second;

if (vm == (1 << N + 1) - 1 && u == 0) return d;

if (visited[u \* 2048 + vm]) continue;

visited[u \* 2048 +vm] = true;

for (int i = 0; i <= N; ++i) {

if (i == u || (1 << i) & vm) continue;

pq.push(make\_pair(d + g[u][i], make\_pair( i, vm | (1 << i) )));

}

}

return -1;

}

int main() {

while (scanf("%d", &N) && N) {

for (int i = 0; i < N + 1; ++i) {

for (int j = 0; j < N + 1; ++j) {

scanf("%d", &g[i][j]);

}

}

fill(visited.begin(), visited.end(), false);

floyd();

printf("%lld\n", solve());

}

return 0;

}

# 3728

## Description

The solution simply does the following:

* Uses vectors to represent boxes and pieces.
* With each piece, it simulates it ‘falling’ into the box and checks if it fits in the box
* If it doesn’t fit, then print out the height of the pieces and make a new box
* Repeat for each test case

The first problem that needed to be solved was to find a way to represent the box and pieces so that collision could be checked for. The solution was to have an array of integers with each integer representing horizontal line section of a box or a piece. Each bit of an integer would represent width-wise coordinate of a piece. 1 represents it is part of the piece and 0 for otherwise. A simple bitwise AND can check if a box row overlaps with a piece row.

The check function takes a box and a piece and ‘drops’ the piece into the box. It checks for when the piece stops, but bitwise ANDing each line, and seeing if it equates to anything but 0, which mean that part of the piece overlaps with pre-existing pieces. If the piece passes, then the box is updated with the piece and its final location before the first ‘collision’

The only problem that was encountered is was that the pieces needed to be reversed, as they were being described top down and not bottom.

## Implemented Solution

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int n, w, b;

bool check(vector<int> & box, const vector<int> & p) {

int i;

for (i = box.size() - p.size(); i >= 0; --i) {

for (int j = 0; j < p.size(); ++j) {

if (box[i + j] & p[p.size() - 1 - j]) goto finish;

}

}

finish:

if (i++ == box.size() - p.size()) {

return false;

}

for (int j = 0; j < p.size(); ++j) {

box[i + j] = p[p.size() - 1 - j];

}

return true;

}

int height(const vector<int> box) {

int res = 0;

for (auto x : box) {

if (x) res++;

}

return res;

}

int main() {

while (cin >> n >> w >> b && n != 0) {

vector<int> result;

vector<int> box(b, 0);

for (int i = 0; i < n; ++i) {

int h;

cin >> h;

vector<int> curr(h);

for (int j = 0; j < h; ++j) {

for (int k = 0; k < w; ++k) {

char c;

cin >> c;

if (c == 'X') curr[j] |= 1 << k;

}

}

if (!check(box, curr)) {

cout << height(box) << " ";

fill(box.begin(), box.end(), 0);

check(box, curr);

}

}

cout << height(box) << "\n";

}

return 0;

}

# 3729

## Description

The implemented solution does the following for each test case:

* Finds all the lines which bridge two points that do not have a point ‘above’ it, by iterating throught all combinations
* Iterates though all combinations of these lines, finding the one with minimal distance to all points

For isolating the lines, it checks each line with all the points using the check function to see if a point lies above the line. A problem occurred here with the derived formula because of a double negation that wasn’t applied which produced the wrong result. *l.b \* p.y - l.a \* p.x - l.c > 0;* was used instead of the correct *l.b \* p.y + l.a \* p.x + l.c > 0;*

Afterwards, the solve function uses the recursive F function to iterate through all combinations of the edges. Vertical distances are cached in D array, in an attempt to increase performance.

This rather brute force solution only works on smaller test cases such as the given sample, and times out for larger ones. A faster algorithm is needed to solve the problem.

## Implemented Solution

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

#include <cmath>

using namespace std;

struct pt {

double x, y;

pt() : x(0), y(0) {};

pt(const double &x, const double &y) : x(x), y(y) {};

};

struct line {

double a, b, c;

line() : a(0), b(0), c(0) {};

line(const pt &p1, const pt &p2) {

a = p1.y - p2.y;

b = p2.x - p1.x;

c = p1.x \* p2.y - p2.x \* p1.y;

}

};

int n, K, L;

vector<pt> pts(100);

vector<line> lines(4950);

vector<int> v(100);

vector<int> D(10000, -1);

bool check(const pt &p, const line &l) {

return l.b \* p.y + l.a \* p.x + l.c > 0;

}

double dist(const pt &p, const line &l) {

return abs(l.b \* p.y + l.a \* p.x + l.c) / l.b;

}

void find\_lines() {

for (int i = 0; i < n; ++i) {

for (int j = i + 1; j < n; ++j) {

lines[L] = line(pts[i], pts[j]);

// check line

for (int k = 0; k < n; ++k) {

if (check(pts[k], lines[L])) {

goto b;

}

}

L++;

b:

continue;

}

}

}

double F(vector<int> &v, const int &x) {

if (x == K) {

double maxd = -1;

for (int i = 0; i < n; ++i) {

double mind = -1;

for (int l = 0; l < K; ++ l) {

double d;

if (D[i \* 100 + v[l]] == -1) D[i \* 100 + v[l]] = dist(pts[i], lines[v[l]]);

d = D[i \* 100 + v[l]];

mind = mind == -1 ? d : min(mind, d);

}

maxd = max(mind, maxd);

}

return maxd;

}

else {

double mind = -1;

int vn = x == 0 ? 0 : v[x - 1] + 1;

if (L - vn < K - x) return -1;

for (int i = vn; i < L; i++) {

v[x] = i;

double maxd = F(v, x + 1);

mind = (mind == -1 || maxd == -1) ? max(mind, maxd) : min(mind, maxd);

}

return mind;

}

}

double solve() {

fill(D.begin(), D.end(), -1);

return F(v, 0);

}

int main() {

ios::sync\_with\_stdio(false);

while (cin >> n >> K && n != 0) {

L = 0;

for (int i = 0; i < n; ++i) {

cin >> pts[i].x >> pts[i].y;

}

find\_lines();

cout << setiosflags(ios::fixed) << setprecision(3) << solve() << "\n";

}

return 0;

## }