Luis Enrique Franco Marín

410921353

Week 11: Solving Maximum Matching Problem

Allow for n agents and n tasks. Any agent can be assigned to any task, which comes at a cost that varies depending on the agent-task assignment. It is necessary to complete all jobs by allocating exactly one agent to each task and exactly one task to each agent in order to reduce the overall cost of the assignment.

An optimal assignment for the resulting cost matrix is also an optimal assignment for the original cost matrix if a number is added to or subtracted from all of the entries of any one row or column of a cost matrix.

Using the preceding theorem, we decrease our initial weight matrix to zeros. We try to assign jobs to agents in such a way that each agent has only one task to do and the penalty is zero in each case.

Problem:

The input file contains several sets of inputs. The description of each set is given below:

The first line of input contains $0<n\leq m\leq 20$. n lines follow, each containing m positive real numbers: the travel time for cruiser m to reach bank n. Input is terminated by a case where m=n=0. This case should not be processed.

For each set of input output, a single number: the minimum average travel time, accurate to 2 fractional digits.

Code:

```cpp
#include <bits/stdc++.h>
#define eps 1e-6

using namespace std;

double W[105][105];
int N, M;
int mx[105], my[105];
double lx[105], ly[105];
int x[105], y[105];
int hungary(int nd) {
    int i;
    x[nd] = 1;
    for(i = 1; i <= M; i++) {
        if(y[i] == 0 && fabs(W[nd][i]-lx[nd]-ly[i]) < eps) {
            y[i] = 1;
            if(my[i] == 0 || hungary(my[i])) {
                my[i] = nd;
                return 1;
            }
        }
    }
    return 0;
}
double minTime() {
    int i, j, k;
    double d;
    //sets the arrays to
    memset(mx, 0, sizeof(mx));
    memset(my, 0, sizeof(my));
    memset(lx, 0, sizeof(lx));
    memset(ly, 0, sizeof(ly));
    for(i = 1; i <= N; i++) //O(n) times
        for(j = 1, lx[i] = W[i][j]; j <= M; j++)//nested O(m)
            lx[i] = lx[i] > W[i][j] ? lx[i] : W[i][j];
    for(i = 1; i <= N; i++) {//O(n)
        while(1) {
            //sets them to 0
            memset(x, 0, sizeof(x));
            memset(y, 0, sizeof(y));
            if(hungary(i))  break;//which can go for O(m^2)
            d = 0xfffffff;//-1
            for(j = 1; j <= N; j++) { //nested O(n)
                if(x[j]) {
                    for(k = 1; k <= M; k++)
                        if(!y[k])
                        d = d < lx[j]+ly[k]-W[j][k] ? d : lx[j]+ly[k]-W[j][k];
                }
            }
            if(d == 0xfffffff)  break; //-1
            for(j = 1; j <= N; j++) //second O(n)
                if(x[j])    lx[j] -= d;
            for(j = 1; j <= M; j++)//O(m)
                if(y[j])    ly[j] += d;
        }
    }
    double res = 0;
    for(i = 1; i <= M; i++) {
        if(my[i])
            res += W[my[i]][i];
    }
    return res;
}
```

```
int main() {
    int n, m;
    while ((cin>>n>>m)&&n&&m){
        if(n == 0 && m == 0)
            break;
        N = n, M = m;
        int i, j;
        for(i = 1; i <= n; i++) {
            for(j = 1; j <= m; j++) {
                cin >> W[i][j];
                W[i][j] *= -1;
            }
        }
        cout << setprecision(2) << fixed << -minTime()/n+eps<< endl;
    }
    return 0;
}
```

Discussion:

The algorithm utilized to solve the problem is the Hungarian algorithm. It functions by creating a matrix in which the relations between two sets are stored. These relations can be interpreted as bipartite graph. The algorithm searches for the lowest number in a row and then subtract this number to all the elements in that row, then does the same for the columns. If the number of rows and columns containing 0 is equal to n then the positions in which the 0's reside correspond to one of the solutions. If the numbers don't match the algorithm continues by looking for the lowest number in the rows that don't contain a solution, subtract the number to all elements in those rows and add it to the column with a solution. This process is repeated until the numbers match. For this several arrays are used but the biggest is size O(n*m), the process goes over the total matrix O(n*(m+n)) while also doing the operation that might take up to m^2, which trumps m+n.

Time: O(n*m^2)

Space: O(n*m)

Result: