Last update: June 8, 2022 Translated From: e-maxx.ru

Games on arbitrary graphs

Let a game be played by two players on an arbitrary graph G. I.e. the current state of the game is a certain vertex. The players perform moves by turns, and move from the current vertex to an adjacent vertex using a connecting edge. Depending on the game, the person that is unable to move will either lose or win the game.

We consider the most general case, the case of an arbitrary directed graph with cycles. It is our task to determine, given an initial state, who will win the game if both players play with optimal strategies or determine that the result of the game will be a draw.

We will solve this problem very efficiently. We will find the solution for all possible starting vertices of the graph in linear time with respect to the number of edges: $O(m)$.

Description of the algorithm

We will call a vertex a winning vertex, if the player starting at this state will win the game, if they play optimally (regardless of what turns the other player makes). Similarly, we will call a vertex a losing vertex, if the player starting at this vertex will lose the game, if the opponent plays optimally.

For some of the vertices of the graph, we already know in advance that they are winning or losing vertices: namely all vertices that have no outgoing edges.

Also we have the following **rules**:

- if a vertex has an outgoing edge that leads to a losing vertex, then the vertex itself is a winning vertex.
- if all outgoing edges of a certain vertex lead to winning vertices, then the vertex itself is a losing vertex.
- if at some point there are still undefined vertices, and neither will fit the first or the second rule, then each of these vertices, when used as a starting vertex, will lead to a draw if both player play optimally.

Thus, we can define an algorithm which runs in $O(nm)$ time immediately. We go through all vertices and try to apply the first or second rule, and repeat.

However, we can accelerate this procedure, and get the complexity down to $O(m)$.

We will go over all the vertices, for which we initially know if they are winning or losing states. For each of them, we start a depth first search. This DFS will move back over the reversed edges. First of all, it will not enter vertices which already are defined as winning or losing vertices. And further, if the search goes from a losing vertex to an undefined vertex, then we mark this one as a winning vertex, and continue the DFS using this new vertex. If we go from a winning vertex to an undefined vertex, then we must check whether all edges from this one leads to winning vertices. We can perform this test in $O(1)$ by storing the number of edges that lead to a winning vertex for each vertex. So if we go from a winning vertex to an undefined one, then we increase the counter, and check if this number is equal to the number of outgoing edges. If this is the case, we can mark this vertex as a losing vertex, and continue the DFS from this vertex. Otherwise we don't know yet, if this vertex is a winning or losing vertex, and therefore it doesn't make sense to keep continuing the DFS using it.

In total we visit every winning and every losing vertex exactly once (undefined vertices are not visited), and we go over each edge also at most one time. Hence the complexity is $O(m)$.

Implementation

Here is the implementation of such a DFS. We assume that the variable `adj_rev` stores the adjacency list for the graph in **reversed** form, i.e. instead of storing the edge (i, j) of the graph, we store (j, i). Also for each vertex we assume that the outgoing degree is already computed.

```
vector<vector<int>> adj_rev;

vector<bool> winning;
vector<bool> losing;
vector<bool> visited;
vector<int> degree;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj_rev[v]) {
        if (!visited[u]) {
            if (losing[v])
                winning[u] = true;
            else if (--degree[u] == 0)
                losing[u] = true;
            else
                continue;
            dfs(u);
        }
    }
}
```

Example: "Policeman and thief"

Here is a concrete example of such a game.

There is $m \times n$ board. Some of the cells cannot be entered. The initial coordinates of the police officer and of the thief are known. One of the cells is the exit. If the policeman and the thief are located at the same cell at any moment, the policeman wins. If the thief is at the exit cell (without the policeman also being on the cell), then the thief wins. The policeman can walk in all 8 directions, the thief only in 4 (along the coordinate axis). Both the policeman and the thief will take turns moving. However they also can skip a turn if they want to. The first move is made by the policeman.

We will now **construct the graph**. For this we must formalize the rules of the game. The current state of the game is determined by the coordinates of the police offices P, the coordinates of the thief T, and also by whose turn it is, let's call this variable P_{turn} (which is true when it is the policeman's turn). Therefore a vertex of the graph is determined by the triple (P, T, P_{turn}) The graph then can be easily constructed, simply by following the rules of the game.

Next we need to determine which vertices are winning and which are losing ones initially. There is a **subtle point** here. The winning / losing vertices depend, in addition to the coordinates, also on P_{turn} - whose turn it. If it is the policeman's turn, then the vertex is a winning vertex, if the coordinates of the policeman and the thief coincide, and the vertex is a losing one if it is not a winning one and the thief is on the exit vertex. If it is the thief's turn, then a vertex is a losing vertex, if the coordinates of the two players coincide, and it is a winning vertex if it is not a losing one, and the thief is at the exit vertex.

The only point before implementing is not, that you need to decide if you want to build the graph **explicitly** or just construct it **on the fly**. On one hand, building the graph explicitly will be a lot easier and there is less chance of making mistakes. On the other hand, it will increase the amount of code and the running time will be slower than if you build the graph on the fly.

The following implementation will construct the graph explicitly:

```cpp
struct State {
    int P, T;
    bool Pstep;
};

vector<State> adj_rev[100][100][2]; // [P][T][Pstep]
bool winning[100][100][2];
bool losing[100][100][2];
bool visited[100][100][2];
int degree[100][100][2];
```

```cpp
void dfs(State v) {
    visited[v.P][v.T][v.Pstep] = true;
    for (State u : adj_rev[v.P][v.T][v.Pstep]) {
        if (!visited[u.P][u.T][u.Pstep]) {
            if (losing[v.P][v.T][v.Pstep])
                winning[u.P][u.T][u.Pstep] = true;
            else if (--degree[u.P][u.T][u.Pstep] == 0)
                losing[u.P][u.T][u.Pstep] = true;
            else
                continue;
            dfs(u);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> a(n);
    for (int i = 0; i < n; i++)
        cin >> a[i];

    for (int P = 0; P < n*m; P++) {
        for (int T = 0; T < n*m; T++) {
            for (int Pstep = 0; Pstep <= 1; Pstep++) {
                int Px = P/m, Py = P%m, Tx = T/m, Ty = T%m;
                if (a[Px][Py]=='*' || a[Tx][Ty]=='*')
                    continue;

                bool& win = winning[P][T][Pstep];
                bool& lose = losing[P][T][Pstep];
                if (Pstep) {
                    win = Px==Tx && Py==Ty;
                    lose = !win && a[Tx][Ty] == 'E';
                } else {
                    lose = Px==Tx && Py==Ty;
                    win = !lose && a[Tx][Ty] == 'E';
                }
                if (win || lose)
                    continue;

                State st = {P,T,!Pstep};
                adj_rev[P][T][Pstep].push_back(st);
                st.Pstep = Pstep;
                degree[P][T][Pstep]++;

                const int dx[] = {-1, 0, 1, 0, -1, -1, 1, 1};
                const int dy[] = {0, 1, 0, -1, -1, 1, -1, 1};
                for (int d = 0; d < (Pstep ? 8 : 4); d++) {
                    int PPx = Px, PPy = Py, TTx = Tx, TTy = Ty;
                    if (Pstep) {
                        PPx += dx[d];
                        PPy += dy[d];
                    } else {
                        TTx += dx[d];
                        TTy += dy[d];
                    }

                    if (PPx >= 0 && PPx < n && PPy >= 0 && PPy < m && a[PPx][PPy]
!= '*' &&
                        TTx >= 0 && TTx < n && TTy >= 0 && TTy < m && a[TTx][TTy]
!= '*')
                    {
                        adj_rev[PPx*m+PPy][TTx*m+TTy][!Pstep].push_back(st);
                        ++degree[P][T][Pstep];
                    }
                }
            }
        }
    }

    for (int P = 0; P < n*m; P++) {
        for (int T = 0; T < n*m; T++) {
            for (int Pstep = 0; Pstep <= 1; Pstep++) {
                if ((winning[P][T][Pstep] || losing[P][T][Pstep]) && !visited[P]
[T][Pstep])
                    dfs({P, T, (bool)Pstep});
            }
        }
    }

    int P_st, T_st;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (a[i][j] == 'P')
                P_st = i*m+j;
            else if (a[i][j] == 'T')
                T_st = i*m+j;
        }
    }

    if (winning[P_st][T_st][true]) {
        cout << "Police catches the thief"  << endl;
    } else if (losing[P_st][T_st][true]) {
        cout << "The thief escapes" << endl;
    } else {
        cout << "Draw" << endl;
    }
}
```

Contributors:

jakobkogler (94.17%) AbhijeetKrishnan (3.14%) adamant-pwn (2.69%)