Code Jam 2008 - Round 3

Analysis: Portal

The challenge in this problem was mainly coding a bug-free solution. It was pretty obvious that solving it involved using a shortest path algorithm. And as you can see, even some people in the top 20 skipped this problem to go for other ones, which were more difficult to figure out, but easier to code.

Let's look at **bmerry**'s solution, because it is very readable, and he was the winner of this round. Then I will continue with some other, more efficient solutions.

A state of the game corresponds to the position of the player in the maze and the positions of the two portals, if they exist. This solution considers the map of the maze indexed from 1 so the values (0, 0) for the coordinates of the portals mean that the portals do not exist.

At each step, the player can create a new portal, move one step North, South, East or West or, if he is currently near a portal, and another portal exists, travel from one portal to the other one. The first type of move can be made instantaneously while the second and third types take one turn.

One optimization step is to find, for each cell, the positions of the portals that can be created from that cell, since you don't want to take O(R + C) every time you need to find the possible moves from a state.

If each move took exactly one turn, then the classic breadth-first search algorithm could provide us with the answer, but in this graph with two types of weights for the edges it seems like we need Dijkstra's shortest path algorithm. In fact, we can still use an algorithm very similar to breadth first search. The tweak is that instead of adding a new state of the same cost with the current state at the end of the state queue, we add it at the beginning. This way the states will be expanded in the order of their costs, which is exactly what Dijkstra's algorithm does. The complexity of this algorithm is $O((R*C)^3)$ instead of $O((R*C)^3 \log (R*C))$, which is the cost of Dijkstra's shortest path algorithm.

Code

Here is **bmerry**'s code, modified slightly and with some additional comments.

```
struct state {
    int r;
    int c;
    // portal rows
    int pr[2];
    // portal columns
    int pc[2];
};

#define ADDR(state) state.r][state.c] \
                  [state.pr[0]][state.pc[0]] \
                  [state.pr[1]][state.pc[1]

static unsigned char prio[16][16][16][16][16][16];

static const int dr[4] = {-1, 0, 1, 0};
static const int dc[4] = {0, -1, 0, 1};
```

```
int main() {
    int cases;
    cin >> cases;
    for (int cas = 0; cas < cases; cas++) {
        cin >> R >> C;
        grid.clear();
        grid.resize(R + 2);

        state start;
        memset(&start, 0, sizeof(start));
        grid[0] = string(C + 2, '#');
        grid[R + 1] = grid[0];
        for (int i = 1; i <= R; i++) {
            string line;
            cin >> line;
            grid[i] = "#" + line + "#";
            if (grid[i].find("O") != string::npos) {
                start.r = i;
                start.c = grid[i].find("O");
                grid[i][start.c] = '.';
            }
        }

        memset(prio, 255, sizeof(prio));
        prio[ADDR(start)] = 0;
        deque<state> q;
        q.push_back(start);
        int ans = -1;
        while (!q.empty()) {
            state cur = q.front();
            unsigned char pri = prio[ADDR(cur)];
            if (grid[cur.r][cur.c] == 'X') {
                ans = pri;
                break;
            }
            q.pop_front();

            for (int d = 0; d < 4; d++) {
                int hr = cur.r;
                int hc = cur.c;
                do {
                    hr += dr[d];
                    hc += dc[d];
                } while (grid[hr][hc] != '#');
                hr -= dr[d];
                hc -= dc[d];

                // adding a new portal
                for (int p = 0; p < 2; p++) {
                    state nxt = cur;
                    nxt.pr[p] = hr;
                    nxt.pc[p] = hc;
                    if (prio[ADDR(nxt)] > pri) {
                        prio[ADDR(nxt)] = pri;
                        // We push the state at the
                        // front since adding a portal
```

```
                        // is instantaneous.
                        q.push_front(nxt);
                    }
                }
            }

            for (int d = 0; d < 4; d++) {
                state nxt = cur;
                nxt.r += dr[d];
                nxt.c += dc[d];
                if (grid[nxt.r][nxt.c] != '#') {
                    if (prio[ADDR(nxt)] > pri + 1) {
                        prio[ADDR(nxt)] = pri + 1;
                        q.push_back(nxt);
                    }
                }
            }
            if (cur.pr[0] > 0 && cur.pr[1] > 0)
                for (int p = 0; p < 2; p++)
                    if (cur.pr[p] == cur.r &&
                       cur.pc[p] == cur.c) {
                        state nxt = cur;
                        nxt.r = cur.pr[1 - p];
                        nxt.c = cur.pc[1 - p];
                        if (prio[ADDR(nxt)] > pri + 1) {
                            prio[ADDR(nxt)] = pri + 1;
                            q.push_back(nxt);
                        }
                    }
        }

        printf("Case #%d: ", cas + 1);
        if (ans == -1)
            printf("THE CAKE IS A LIE\n");
        else
            printf("%d\n", ans);
    }
    return 0;
}
```

The restrictions on the inputs in the problem are small enough so that this solution passes all the tests. There are other, more efficient solutions which we thought of.

Other solutions

It does not make sense to create the starting portal before actually being able to jump through it. This way we can improve the previous solution and get the complexity down to $O((R*C)^2)$.

As we have just seen, premature portal creation is the root of complexity, so let's think now of the destination portal. Once we have created a destination portal, we need to move as quickly as possible to the nearest wall, create a starting portal, walk through it and arrive at the destination. So what we can do is just keep R*C states and move from one to another by either going North, West, South or East or do a teleport move which takes a few turns. We can compute in O(R*C) how much time each teleport move takes by doing a breadth first search starting from all the cells in the maze where a portal can be created. Now we can use Dijkstra's algorithm to find the shortest path. The final algorithm can have either O(R*C log (R*C))

complexity, or if we use the fact that the teleport edges have the cost at most R * C, by using a array of lists in Dijkstra's algorithm instead of a priority queue, we can get the complexity of the solution down to O(R * C).

More Information

Dijkstra's algorithm - Breadth First Search