CSES - Grid Paths

Author: Vivian Han

Language: C++ Edit This Page

Prerequisites

- Bronze - Complete Search with Recursion

View Problem Statement

TABLE OF CONTENTS

Explanation

Adapted from /CPH.pdf#page=61

Warning: Run times

Please note: the running times and number of recursive calls listed are for a separate but very similar problem (# paths from upper left to *lower right* corner on a 7×7 grid). They may not be entirely accurate, but they illustrate the effect of each optimization. The data is also C++ specific, so Java and Python users will experience significantly different runtimes. The consequences of this discrepancy which will be addressed later in the explanation.

Basic Algorithm

The first version of the algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths from the upper-left corner to the lower-right corner and count the number of such paths.

- Running time: 483 seconds
- Number of recursive calls: 76 billion

Optimization 1

If the path reaches the lower-right square before it has visited all other squares of the grid, it is clear that it will not be possible to complete the solution. Using this observation, we can terminate the search immediately if we reach the lower-right square too early.

- Running time: 119 seconds
- Number of recursive calls: 20 billion

Optimization 2

If the path touches a wall and can turn either left or right, the grid splits into two parts that contain unvisited squares. In this case, we cannot visit all squares anymore, so we can terminate the search.

- Running time: 1.8 seconds
- Number of recursive calls: 221 million

Optimization 3

The idea of Optimization 2 can be generalized: if the path cannot continue forward but can turn either left or right, the grid splits into two parts that both contain unvisited squares. It is clear that we cannot visit all squares anymore, so we can terminate the search.

- Running time: 0.6 seconds
- Number of recursive calls: 69 million

Now is a good moment to stop optimizing the algorithm and see what we have achieved. The running time of the original algorithm was 483 seconds, and now after the optimizations, the

running time is only 0.6 seconds. Thus, the algorithm became nearly 1000 times faster after the optimizations.

> In backtracking, the search tree is usually large and even simple observations can effectively prune the search. Especially useful are optimizations that occur during the first steps of the algorithm, i.e., at the top of the search tree.

For C++ users, this will be enough pruning to make the program run in time under worst case conditions. However, for Java and Python users, this is still too slow (Java takes around 1.2 seconds for the worst case). Thus, we will need to further optimize our search.

Optimization 4

If the path creates a dead end that is not the bottom left corner, either the path will fail to visit all squares (the path may stop at the dead end or pass over it, sealing a square off) or the path will end in the wrong location. Thus, we want to avoid creating dead ends. For example, if the square to the left of our current location is blocked on three sides (including our current location), then the next step must be to the left in order to avoid creating a dead end. After this optimization, the program runs in under 1 second.

Implementation

Copy CPP

```cpp
#include <iostream>
using namespace std;

const int DIR_LEN = 4;
int dr[DIR_LEN] = {-1, 0, 1, 0};
int dc[DIR_LEN] = {0, 1, 0, -1};
const int PATH_LEN = 48;  // length of all possible paths
int p[PATH_LEN];
const int GRID_SIZE = 9;
// added border to all four sides so a 7x7 becomes a 9x9
bool onPath[GRID_SIZE][GRID_SIZE];

int tryPath(int pathIdx, int curR, int curC) {
    // Optimization 3
    if ((onPath[curR][curC - 1] && onPath[curR][curC + 1])
        && (!onPath[curR - 1][curC] && !onPath[curR + 1][curC])) return 0;
    if ((onPath[curR - 1][curC] && onPath[curR + 1][curC])
        && (!onPath[curR][curC - 1] && !onPath[curR][curC + 1])) return 0;
```

```
19
20      if (curR == 7 && curC == 1) {  // reached endpoint before visiting all
21          if (pathIdx == PATH_LEN) return 1;
22          return 0;
23      }
24
25      if (pathIdx == PATH_LEN) return 0;
26
27      int ret = 0;
28      onPath[curR][curC] = true;
29
30      // turn already determined:
31      if (p[pathIdx] < 4) {
32          int nxtR = curR + dr[p[pathIdx]];
33          int nxtC = curC + dc[p[pathIdx]];
34          if (!onPath[nxtR][nxtC]) ret += tryPath(pathIdx + 1, nxtR, nxtC);
35      }
36      // see Java solution for optimization 4 implementation
37      else {  // iterate through all four possible turns
38          for (int i = 0; i < DIR_LEN; i++) {
39              int nxtR = curR + dr[i];
40              int nxtC = curC + dc[i];
41              if (onPath[nxtR][nxtC]) continue;
42              ret += tryPath(pathIdx + 1, nxtR, nxtC);
43          }
44      }
45      // reset and return
46      onPath[curR][curC] = false;
47      return ret;
48  }
49
50  int main() {
51      string line;
52      getline(cin, line);
53
54      // convert path to ints
55      for (int i = 0; i < PATH_LEN; i++) {
56          char cur = line[i];
57
58          if (cur == 'U') p[i] = 0;
59          else if (cur == 'R') p[i] = 1;
60          else if (cur == 'D') p[i] = 2;
61          else if (cur == 'L') p[i] = 3;
62          else p[i] = 4; //cur == '?'
63      }
64
```

```
    // set borders of grid
    for (int i = 0; i < GRID_SIZE; i++) {
        onPath[0][i] = true;
        onPath[8][i] = true;
        onPath[i][0] = true;
        onPath[i][8] = true;
    }
    // initialize the inside of the grid to be completely empty
    for (int i = 1; i <= 7; i++) {
        for (int j = 1; j <= 7; j++) {
            onPath[i][j] = false;
        }
    }

    int startIdx = 0;
    int startR = 1;
    int startC = 1;  // always start path at (1, 1)
    int ans = tryPath(startIdx, startR, startC);
    cout << ans << endl;
}
```