138 10

294. Flip Game II

Description (/problems/flip-game-ii/description/) Hints (/problems/flip-game-ii/hints/) Submissions (/problems/flip-game-ii/subm

Notes

Theory matters - from Backtracking(128ms) to DP (0ms)

28.8K
VIEWS

212

Last Edit: Mar 21, 2018, 4:17 PM

(/stellari) stellari (/stellari) ★ 2201

(/problems/flip-game-ii/discuss) > Theory matters - from Backtracking(128ms) to DP (0ms)

Share Subscribe Report

At first glance, backtracking seems to be the only feasible solution to this problem. We can basically try every possible move for the first player (Let’s call him 1P from now on), and recursively check if the second player 2P has any chance to win. If 2P is guaranteed to lose, then we know the current move 1P takes must be the winning move. The naive implementation is actually very simple:

```
int len;
string ss;
bool canWin(string s) {
    len = s.size();
    ss = s;
    return canWin();
}
bool canWin() {
    for (int is = 0; is <= len-2; ++is) {
        if (ss[is] == '+' && ss[is+1] == '+') {
            ss[is] = '-'; ss[is+1] = '-';
            bool wins = !canWin();
            ss[is] = '+'; ss[is+1] = '+';
            if (wins) return true;
        }
    }
    return false;
}
```

Now let’s check the time complexity: Suppose originally the board of size N contains only ‘+’ signs, then roughly we have:

```
T(N) = (N-2) * T(N-2) = (N-2) * (N-4) * T(N-4) ... = (N-2) * (N-4) * (N-6) * ... ~ (
```

This algorithm can be optimized with memoization. For instance:
https://discuss.leetcode.com/topic/27291/memoization-3150ms-130ms-44ms-python
(https://discuss.leetcode.com/topic/27291/memoization-3150ms-130ms-44ms-python)

Can we even do better than that? Sure! Below I'll show the time complexity can be reduced to O(N^2) using Dynamic Programming, but the improved method requires some non-trivial understanding of the game theory, and therefore is not expected in a real interview. If you are not interested, please simply skip the rest of the article:

Notes

> Concept 1 (**Impartial Game**): Given a particular arrangement of the game board, if either player have exactly the same set of moves should he move first, and both players have exactly the same winning condition, then this game is called **impartial game**. For example, chess is not impartial because the players can control only their own pieces, and the ± flip game, on the other hand, is impartial.

_

> Concept 2 (**Normal Play vs Misere Play**): If the winning condition of the game is that the **opponent has no valid moves**, then this game is said to follow the **normal play convention**; if, alternatively, the winning condition is that the **player himself has no valid moves,** then the game is a **Misere** game. Our ± flip has apprently normal play.

Now we understand the the flip game is an impartial game under normal play. Luckily, this type of game has been extensively studied. Note that our following discussion only applies to normal impartial games.

In order to simplify the solution, we still need to understand one more concept:

> Concept 3 (**Sprague-Grundy Function**): Suppose x represents a particular arrangement of board, and x_0, x_1, x_2, … ,x_k represent the board after a valid move, then we define the Sprague-Grundy function as:

```
g(x) = FirstMissingNumber(g(x_0), g(x_1), g(x_2), ... , g(x_k)).
```

> where FirstMissingNumber(y) stands for the smallest positive number that is not in set y. For instance, if g(x_0) = 0, g(x_1) = 0, g(x_k) = 2, then g(x) = FMV({0, 0, 2}) = 1.

Why do we need this bizarre looking S-G function? Because we can instantly decide whether 1P has a winning move simply by looking at its value. I don’t want to write a book out of it, so for now, please simply take the following theorem for granted:

> Theorem 1: If g(x) != 0, then 1P must have a guaranteed winning move from board state x. Otherwise, no matter how 1P moves, 2P must then have a winning move.

So our task now is to calculate g(board). But how to do that? Let’s first of all find a way to numerically describe the board. Since we can only flip ++ to --, then apparently, we only need to write down the lengths of consecutive ++'s of length >= 2 to define a board. For instance, +±-±+++±±---- can be represented as (2, 4).

(2, 4) has two separate ‘+’ subsequences. Any operation made on one subsequence does not interfere with the state of the other. Therefore, we say (2, 4) consists of two **subgames**: (2) and (4).

Okay now we are only one more theorem away from the solution. This is the last theorem. I promise:

> Theorem 2 (**Sprague-Grundy Theorem**): The S-G function of game x = (s1, s2, …, sk) equals the XOR of all its subgames s1, s2, …, sk. e.g. g((s1, s2, s3)) = g(s1) XOR g(s2) XOR g(s3).

With the S-G theorem, we can now compute any arbitrary g(x). If x contains only one number N (there is only one ‘+’ subsequence), then

```
g(x) = FMV(g(0, N-2), g(1, N-3), g(2, N-4), ... , g(N/2-1, N-N/2-2));
     = FMV(g(0)^g(N-2), g(1)^g(N-3), g(2)^g(N-4)), ... g(N/2-1, N-N/2-2));
```

Now we have the whole algorithm:

```
Convert the board to numerical representation: x = (s1, s2, ..., sk)
Calculate g(0) to g(max(si)) using DP.
if g(s1)^g(s2)^...^g(sk) != 0 return true, otherwise return false.
```

Calculating g(N) takes O(N) time (N/2 XOR operations plus the O(N) First Missing Number algorithm). And we must calculate from g(1) all the way to g(N). So overall, the algorithm has an O(N^2) time complexity.

Naturally, the code is bit more complicated than the backtracking version. But it reduces the running time from ~128ms to less than 1ms. The huge improvement is definitely worth all the hassle we went through:

```
int firstMissingNumber(unordered_set<int> lut) {
    int m = lut.size();
    for (int i = 0; i < m; ++i) {
        if (lut.count(i) == 0) return i;
    }
    return m;
}

bool canWin(string s) {
    int curlen = 0, maxlen = 0;
    vector<int> board_init_state;
    for (int i = 0; i < s.size(); ++i) {
        if (s[i] == '+') curlen++;              // Find the length of all continuous
        if (i+1 == s.size() || s[i] == '-') {
            if (curlen >= 2) board_init_state.push_back(curlen);    // only length
            maxlen = max(maxlen, curlen);       // Also get the maximum continuous
            curlen = 0;
        }
    }          // For instance ++--+--++++-+ will be represented as (2, 4)
    vector<int> g(maxlen+1, 0);    // Sprague-Grundy function of 0 ~ maxlen
    for (int len = 2; len <= maxlen; ++len) {
        unordered_set<int> gsub;    // the S-G value of all subgame states
        for (int len_first_game = 0; len_first_game < len/2; ++len_first_game) {
            int len_second_game = len - len_first_game - 2;
            // Theorem 2: g[game] = g[subgame1]^g[subgame2]^g[subgame3]...;
            gsub.insert(g[len_first_game] ^ g[len_second_game]);
        }
        g[len] = firstMissingNumber(gsub);
    }

    int g_final = 0;
    for (auto& s: board_init_state) g_final ^= g[s];
    return g_final != 0;    // Theorem 1: First player must win iff g(current_state
 }
```

Comments: 32

Sort By

Type comment here... (Markdown is supported)

Preview Post

StefanPochmann (/stefanpochmann) 22367 Oct 16, 2015, 9:54 AM

stellari is back! :-)
Nice article. I had started with that approach but stopped when I realized that simple backtracking was enough. Anyway, here's a **Python** implementation now:

```
def canWin(self, s):
```

Read More

15 Share Reply

SHOW 2 REPLIES

Notes

ofLucas (/oflucas) ★ 728 Apr 14, 2016, 7:04 PM

In addition to the best-voted backtracking solution, I added a hashmap to memorize repeating solution. Below is my code. I also demonstrate to you how memorization saves repeating work by printing result with “==>”. Performance is 10ms 98.5%

Read More

6 Share Reply

jinwu (/jinwu) ★ 1197 Oct 16, 2015, 2:16 PM

Great idea and amazing code! Thank both of you guys for sharing! I don't remember I ever learnt this in the classroom. Where did you learn this? Here is my java version based on Stefan's code:

```
public boolean canWin(String s) {
        s = s.replace('-', ' ');
```

Read More

6 Share Reply

SHOW 5 REPLIES

stellari (/stellari) ★ 2201 Oct 16, 2015, 5:43 PM

@jianchao, In my understanding, the S-G function basically says: If ANY subsequent state of state x is a losing position (g == 0), then x itself must be a winning position (g != 0); if ALL subsequent states are winning positions (g!=0), then x must be a losing position (g ==0). The above 2 situations can be unified by a FirstMissingNumber operation. For any normal impartial game, simply find out all states x_0, x_1, … x_k reachable from a state x, then use Concept 3 and Theorem 2 to calculate S-G function.

Read More

3 Share Reply

stellari (/stellari) ★ 2201 Oct 17, 2015, 6:29 PM

@pointbreak Exactly. These are the possible board states after one valid move. BTW we actually don't need to calculate g(0) and g(1) because it's obvious we can't flip 0 or 1 plus signs, but keeping them there makes the implementation more consistent and easy.

1 Share Reply

jianchao.li.fighter (/jianchaolifighter) ★ 5881 Oct 16, 2015, 10:57 AM

Hi, stellari. Thanks for your efforts to share such nice game theories and writing such a nice and detailed article. But I still have some questions regarding the S-G functions: I want to know what does it represent? How could we come up with this function from a game like the Filp Game? I can understand what your algorithm does but I am confused with why it is supposed to do that.

1 Share Reply

1337c0d3r (/1337c0d3r) ★ 2113 Oct 16, 2015, 2:10 AM

Awesome analysis! Thanks for sharing this amazing theory with us, @stellari.

1 Share Reply

Notes

swifttime (/swifttime) ★ 0 Mar 23, 2018, 8:51 AM

If i have 9 +'s as input : + + + + + + + + +
player 1 using '-'
player 2 using 'o'
one possible win path for player 1 will be : - - + o o + - - +, which player 1 did 2 times and player 1 did 1 times, and player 1 won the game.

Read More

0 Share Reply

MockingJay15 (/mockingjay15) ★ 0 Jan 26, 2018, 1:35 PM

very nice solution stellari!

0 Share Reply

sstcurry (/sstcurry) ★ 2 Oct 31, 2017, 8:02 AM

@jkarimi I thought the same as yours.

```
T(N) = (N-1) * T(N-2) = (N-1) * (N-3) * T(N-4) = (N-1) * (N-3) * (N-5) .... * T(0) = O(N!)
```

0 Share Reply

‹ 1 2 3 4 ›