max-n-depth

April 21, 2020

[1]:
```
import pandas as pd
```

1 Determining the Search Depth for maxN

This script is the successor to max-n-performance.pdf. See that script first, then come back here.

Since we now know, what influences the calculation time of the maxN algorithm, it is time to draw conclusions.

The goal is, that no execution of the maxN algorithm takes longer than 1 second. All calculations are done on a Macbook Pro 2018 16GB RAM, so a quite powerful machine. The maxN algorithm should be executed on mobile devices. Within the app, the computer opponent's turn is always delayed by 1-2 seconds, even though the computer turn might already be calculated. The reason is, that the user expierence is a lot better, when the computer 'takes a little time' and not just jumps to the next move. Therefore, a limit for the computation of 1 second was chosen.

The two main factors that influence the performance is of course the search depth and the number of moves all pawns on the board could do. The higher the search depth the better and stronger our computer opponent. The higher the number of moves the pawns could do, the longer the calculation takes. So, we try to find the highest possible search depth, depending also on the number of moves the pawns could do, where the execution of maxN still takes less than 1 second.

1.1 Preparation

In order to do that, we take again the randomly generated games, that we used previously.

Now we do the following:

1. calculate the number of moves the pawns could do for every game state in our data set
2. order the games states from the lowest number of moves to the highest

This is done by a node-script, located at `src/data-generation/random-games-by-moves.ts`.

The result is stored in `data/random-games-by-moves.json`

Let's have a look:

[2]:
```
pd.read_json('../data/random-games-by-moves.json')
```

[2]:
```
                                                    gs  numOfMoves
0    {'limits': {'minRow': 4, 'maxRow': 6, 'minCol'…           2
1    {'limits': {'minRow': 1, 'maxRow': 3, 'minCol'…           2
2    {'limits': {'minRow': 5, 'maxRow': 7, 'minCol'…           2
3    {'limits': {'minRow': 1, 'maxRow': 3, 'minCol'…           2
4    {'limits': {'minRow': 4, 'maxRow': 6, 'minCol'…           2
..                                                 …         …
964  {'limits': {'minRow': 0, 'maxRow': 7, 'minCol'…         121
965  {'limits': {'minRow': 0, 'maxRow': 7, 'minCol'…         122
966  {'limits': {'minRow': 0, 'maxRow': 7, 'minCol'…         123
967  {'limits': {'minRow': 0, 'maxRow': 7, 'minCol'…         124
968  {'limits': {'minRow': 0, 'maxRow': 7, 'minCol'…         125

[969 rows x 2 columns]
```

So, we see the respective game state plus the number of moves for all pawns on the board and they are ordered by the number of moves.

1.2 Finding the Limits for maxN

Now, we will loop through this data set again and again. With each iteration we will increase the search depth. If a computation takes longer than 1 second, we store the number of moves and start the next iteration.

The higher the search depth gets, the sooner it will exceed the limit of 1 second, even for game states with a lower number of moves.

We will test search depth from 1-30 and store for each depth the number of moves where the computation took too long.

All this is done by another node-script at `src/data-generation/max-n-depth.ts`.

The results are stored in the file `data/max-n-depth.csv`. Let's have a look:

[3]:
```
limits = pd.read_csv('../data/max-n-depth.csv')
limits
```

[3]:
```
    depth  numOfMovesWhenTooSlow
0       1                     -1
1       2                     -1
2       3                     93
3       4                     39
4       5                     24
5       6                     15
6       7                     11
7       8                     11
8       9                     11
9      10                     11
10     11                     11
```

11	12	10
12	13	8
13	14	4
14	15	4
15	16	4
16	17	4
17	18	4
18	19	4
19	20	4
20	21	4
21	22	4
22	23	4
23	24	4
24	25	4
25	26	4
26	27	4
27	28	4
28	29	4
29	30	4

For each search depth from 1 to 30 we get the number of moves of the game state, where the execution took too long.

If none of the game states took too long to compute for a given depth, the entry is -1. So, we can see that a search depth of at least 2 is always possible.

The rest is only acceptable, when the number of moves is below the given limit. So, a search depth of 3 can only be used safely, when the number of moves is less than 93 etc.

We can also see, that 13 is the highest search depth to be used. Higher search depths are only feasible, when the number of moves is less than 4 and if that happens the game is almost finished any way. So, we ignore that case and use a search depth of 13 for all game states where the number of moves is smaller than 8.

1.3 Final Result

Now, we can implement a function, that gets the current game state passed and returns the corresponding search depth. This function is of course written in TypeScript and is now part of the maxN algorithm.

```
/** returns the allowed search depth for a game state */
function calcDepth(gameState: IGameState): number {
    const numOfMoves = countMoves(gameState);

    if (numOfMoves <  8) return 13;
    if (numOfMoves < 10) return 12;
    if (numOfMoves < 11) return 11;
    if (numOfMoves < 15) return  6;
    if (numOfMoves < 24) return  5;
```

```
    if (numOfMoves < 39) return  4;
    if (numOfMoves < 93) return  3;

    return 2;
}

/** helper function to count total number of moves */
function countMoves(gameState: IGameState): number {
    let result = 0;
    for (let i = 0, ie = gameState.pawns.length; i < ie; i++) {
        result += getMoves(gameState, i).length;
    }
    return result;
}
```