branching-factor-and-depth

April 21, 2020

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

1 Calculating Complexity for Chameleon Chess

This script will calculate the game tree complexity of the game chameleon chess.

The complexity of a turn-based board game can be calculated with: b^d. b is the branching factor, which is the usual number of moves a player could perform on a turn. d is the depth, which is the usual length of a game (the total number of moves performed by all players).

For that purpose, a series of games were played by the computer player (maxN). There were games played with two, three and four players. The script to play the games is `src/branching-factor-and-depth.ts`. The results were stored to `data/branching-factor-and-depth.csv`.

In order to simulate different levels of experience in playing and to prevent all the games to be played in exactly then same manner, a random component was used. That means that there was a certain chance, that a random move was chosen, instead of one by the maxN algorithm. Some games were played with 1%, 10% and 20% chance for a random move.

Let's load the results from the games and have look on the data.

```
[8]: raw_data = pd.read_csv('../data/branching-factor-and-depth.csv')
     raw_data
```

```
[8]:   algorithm  propForRandomMove  numOfPlayers  branchingFactor  depth
     0       maxN               0.01             2        11.833333     24
     1       maxN               0.01             2        14.388889     18
     2       maxN               0.01             2        14.388889     18
     3       maxN               0.01             2        14.388889     18
     4       maxN               0.01             2        14.388889     18
     ..       …                  …             …              …     …
     85      maxN               0.20             4        13.833333     54
     86      maxN               0.20             4        15.238095     42
     87      maxN               0.20             4        18.482143     56
     88      maxN               0.20             4        17.428571     42
     89      maxN               0.20             4        13.625000     32
```

```
[90 rows x 5 columns]
```

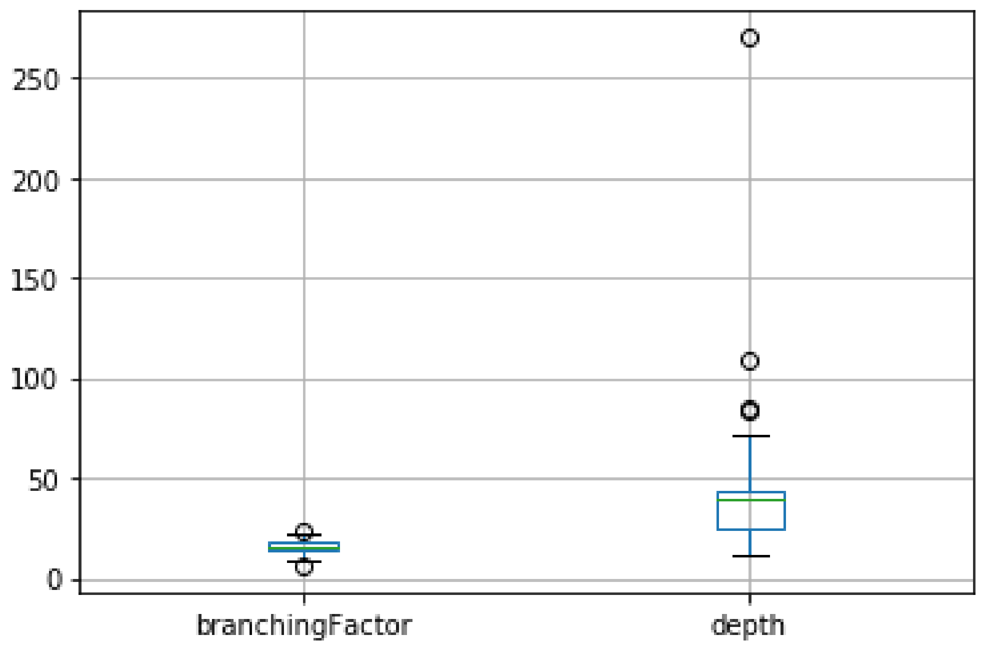
The attributes ‘algorithm’, ‘propForRandomMove’ and ‘numOfPlayers’ encode the settings of the particular game.

‘branchingFactor’ and ‘depth’ are our target attributes.

Let’s check, how our target values are distributed.

```
[3]: raw_data.loc[:,['branchingFactor','depth']].plot.box(grid='True')
```

```
[3]: <matplotlib.axes._subplots.AxesSubplot at 0x7f85b2515c90>
```

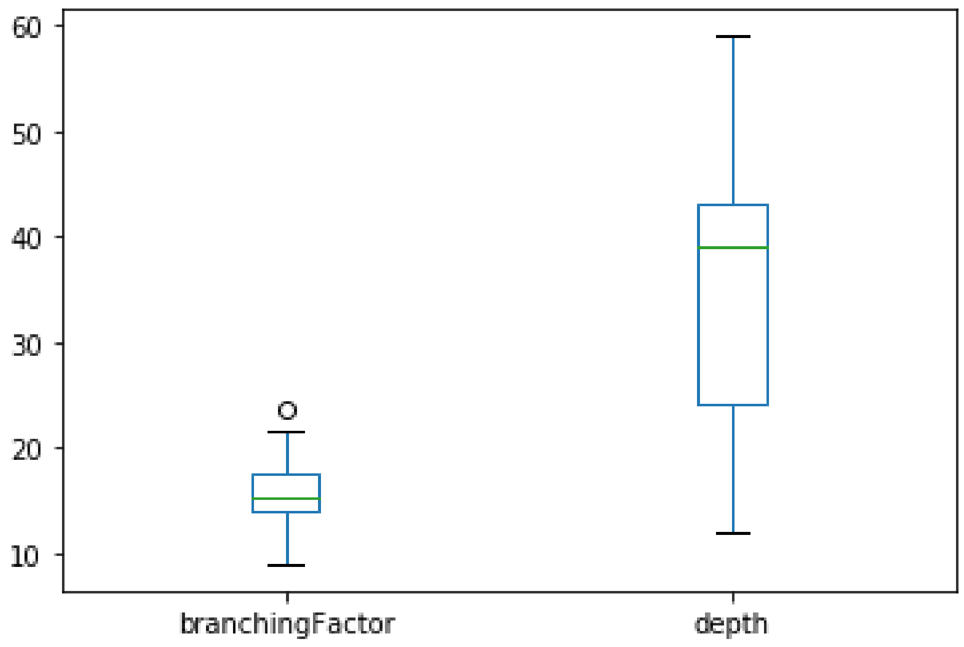


There are some outliers in the ‘depth’ property. When the game is gettings near the end, circles may occur. That means that both remaining players are basically ‘dancing around each other’ and no one gets a chance to beat the other. In reality with human players this never occurs (personal experience). The random moves should also prevent these circles. But it seems there are some cases, where a circular game took place.

Lets remove these ones from the data!

```
[4]: data = raw_data[raw_data.depth < 70]
     data.loc[:,['branchingFactor','depth']].plot.box()
```

```
[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f85b005cb50>
```



Now we have a more realistic data set.

There seems to be another outlier in the branching factor, but this is not unusual. By playing skillfully a player might have several pawns in very good positions (e.g. all pawns have the role 'queen'), thus causing the branching factor to skyrocket for this turn.

The data set is cleaned up. Let's get into the analysis.

```
[5]: data_p2 = data[data.numOfPlayers == 2]
data_p3 = data[data.numOfPlayers == 3]
data_p4 = data[data.numOfPlayers == 4]

data_p2_1 = data[(data.numOfPlayers == 2) & (data.propForRandomMove == 0.01)]
data_p3_1 = data[(data.numOfPlayers == 3) & (data.propForRandomMove == 0.01)]
data_p4_1 = data[(data.numOfPlayers == 4) & (data.propForRandomMove == 0.01)]

data_p2_10 = data[(data.numOfPlayers == 2) & (data.propForRandomMove == 0.1)]
data_p3_10 = data[(data.numOfPlayers == 3) & (data.propForRandomMove == 0.1)]
data_p4_10 = data[(data.numOfPlayers == 4) & (data.propForRandomMove == 0.1)]

data_p2_20 = data[(data.numOfPlayers == 2) & (data.propForRandomMove == 0.2)]
data_p3_20 = data[(data.numOfPlayers == 3) & (data.propForRandomMove == 0.2)]
data_p4_20 = data[(data.numOfPlayers == 4) & (data.propForRandomMove == 0.2)]
```

1.1 Analysis of Branching Factor and Depth

[6]:
```
table = [
    [
        '', data_p2_1.branchingFactor.mean(),  data_p2_1.depth.mean(),
        '', data_p2_10.branchingFactor.mean(), data_p2_10.depth.mean(),
        '', data_p2_20.branchingFactor.mean(), data_p2_20.depth.mean(),
        '', data_p2.branchingFactor.mean(),    data_p2.depth.mean(),
    ],[
        '', data_p3_1.branchingFactor.mean(),  data_p3_1.depth.mean(),
        '', data_p3_10.branchingFactor.mean(), data_p3_10.depth.mean(),
        '', data_p3_20.branchingFactor.mean(), data_p3_20.depth.mean(),
        '', data_p3.branchingFactor.mean(),    data_p3.depth.mean(),
    ],[
        '', data_p4_1.branchingFactor.mean(),  data_p4_1.depth.mean(),
        '', data_p4_10.branchingFactor.mean(), data_p4_10.depth.mean(),
        '', data_p4_20.branchingFactor.mean(), data_p4_20.depth.mean(),
        '', data_p4.branchingFactor.mean(),    data_p4.depth.mean(),
    ]
]
pd.DataFrame(table, index=['2 Players','3 Players','4 Players'],columns=['| 1%:
 ↪','b','d','| 10%:','b','d','| 20%:','b','d','| all:','b','d']).round(0)
```

[6]:
```
          | 1%:     b     d | 10%:     b     d | 20%:     b     d | all:  \
2 Players        15.0  19.0         15.0  19.0         14.0  26.0
3 Players        19.0  40.0         17.0  41.0         17.0  40.0
4 Players        14.0  40.0         14.0  47.0         16.0  46.0

              b     d
2 Players  15.0  22.0
3 Players  18.0  40.0
4 Players  15.0  44.0
```

Branching factor (b) and depth (d) stay fairly consistent even for different percentages of random moves.

However, especially the depth differs depending on the number of players. This is not surprising, because the number of players must have an effect on the overall length of a game. The more players, the more moves are made. Double the number of players also doubles the game length, which seems plausible.

Now, we can finally calculate the actual complexity. The calculation will be done for each player constellation separately.

1.2 Final Result

```
[7]: def calcComplexity(b,d):
         return b ** d

     table2 = [
         [
             calcComplexity(data_p2.branchingFactor.mean(), data_p2.depth.mean()),
             calcComplexity(data_p3.branchingFactor.mean(), data_p3.depth.mean()),
             calcComplexity(data_p4.branchingFactor.mean(), data_p4.depth.mean()),
             calcComplexity(data.branchingFactor.mean(), data.depth.mean()),
         ]
     ]
     pd.DataFrame(table2, index=['complexity'], columns=['2 Players','3 Players','4␣
      ↪Players','total'])
```

```
[7]:                2 Players     3 Players     4 Players         total
     complexity  1.404454e+25  9.725062e+49  8.057480e+51  1.129939e+42
```

Chameleon Chess has a high complexity spanning from 10^{25} for a two player game, up to almost 10^{52} in a four player game.