Large Scale Chess Game Analysis

Mathieu Acher
Associate Professor at University of Rennes 1, member of
DiverSE team at INRIA, 35042 Rennes Cedex, France

Eric Manzi
Department of Computer Science and Engineering, MIT,
MA 02139 USA, Intern in DiverSE team at INRIA

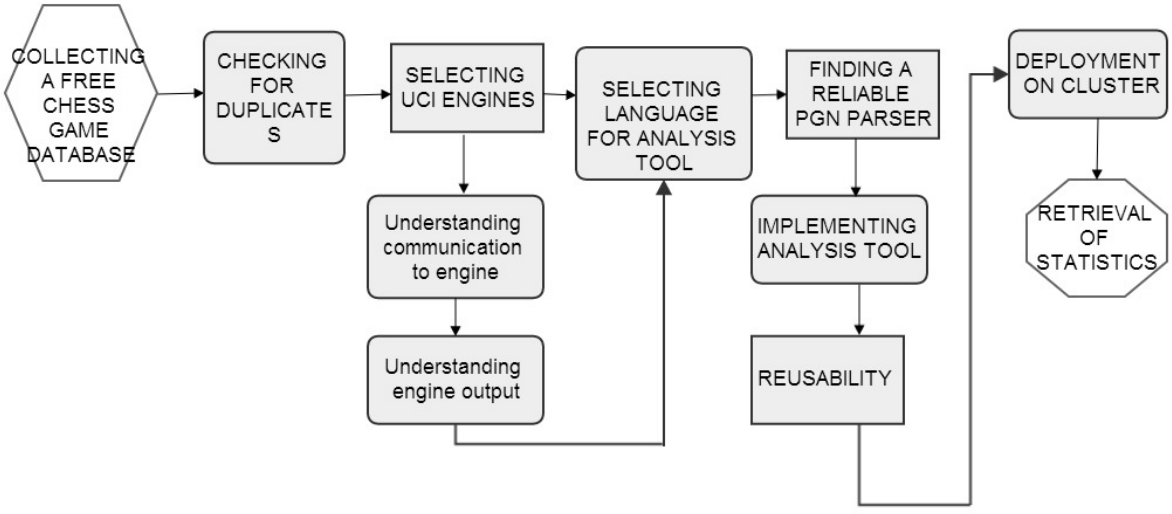
July 31, 2014

Abstract

This paper describes the building of a scalable infrastructure aimed at analyzing bulk collections of recorded chess games in PGN format. With the help of the top UCI chess engines such as Stockfish and Houdini, we analyze moves from 5,164,175 chess games. A database of analyzed games can hence be used by chess researchers to study player performances on a large scale with statistical confidence. There have been criticisms of the Elo approach, predominantly due to its inability to accurately determine relative skill of players in different leagues and in different eras. Previous attempts to measure the accuracy and stability of a more absolute rating system have all had one thing in common that they lacked: Statistical confidence. We seek to make these experiments scalable; able to handle a growing number of games.

MOTIVATION

The motivation for our work is the lack of infrastructure for executing and reproducing large-scale experiments. Several experiments have been conducted on recorded chess games but they were done on a relative small scale. For example, the alternatives to the Elo system proposed by researchers such as Regan, McHaworth, Guid and Bratko all suggest benchmarking against engines, which themselves continue to be improved in playing ability. Experiments, detailed below, have been carried out to assess the proposed systems, but few have fully addressed the issues of statistical confidence. The end solution that we envision will enable researchers to analyze millions of games in a short period of time. The rest of this paper details the progression of this project. The following is an outline of the different steps taken:



TODO (Mathieu): I think there are two distinct parts: one for motivating our work (lack of infrastructure for executing and reproducing large-scale experiments); another for describing our envisioned solution.

PROJECT TIMELINE

1 Collecting a free chess game database

TODO (Mathieu): What is currently missing is a "picture" of the different steps (the infrastructure we have in mind: chess database, PGN parser, protocols to communicate with UCI, deployments on a cluster of machine (grid/cloud), retrieval of the statistics, etc)

Evidently, the first step had to be compiling a chess game database. There exist a wide range of sources but most of these are, as you may have guessed, not free. Most of the ones that happen to be free are much smaller collections (usually about 15000 games a package), Chess.com[1] for example, and others, though available for download can only be collected one at a time, see ChessTempo[2] which has a collection of 1464928 games. I tried using a curl script:

$ curl O http://chesstempo.com/requests/download_game_pgn.php?gameids=[1-9000000]

to download all the games at chesstempo but realized soon after that this would have taken months to download every single game. So the search continued. Finally, I came across the ICOfy Base[3] which offers free chess games collections in PGN file format. [The site is dedicated] to offering chess players of all levels a free possibility to prepare for specific opponents or opening systems. We were able to obtain 5,164,175 free recorded chess games from the 19^{th} century up to May 2014.

2 Checking for duplicates

While ICOfy claims to have deleted all duplicated games in their databases, we had to check for duplicates ourselves before we could proceed. Using pgn-extract, a Portable Game Notation Manipulator for Chess Games, version 17-14 by David J. Barnes, a command-line program for manipulating chess games in PGN file format, I called the check file for duplicates function on the chess database:

$ pgn-extract dDuplicates.pgn oUnique.pgn input-file.pgn

Which outputs all the duplicates into Duplicates.pgn and creates a Unique.pgn file which has no duplicates, leaving the original input-file untouched. Detecting duplicates requires memory for storing a hash table containing information on each game, a MallocOrDie: not enough space may result. To prevent this, we used the Z flag which forces pgn-extract to store hash tables externally in a file called virtual.tmp [4]

3 Selecting ideal UCI engines for our analysis tool

The next step included selecting the right engines to carry out our analyses. It was important to find valid analysis tools to evaluate large databases such as the one we had. The three most important criteria were

1. It had to be available to everyone free-of-charge
2. Competitive playing strength
3. Not discountinued, still being maintained.

The following were among the top UCI engines listed as free by SDChess[5]: RYBKA(v.2.3.2a is free, v4 is purchase only), STRELKA,

SPIKE, FRUIT, TOGA II, GLAURUNG, STOCKFISH, KOMODO and HOUDINI.
STOCKFISH, KOMODO, RYBKA v2.3.2 and HOUDINI met our criteria.

4 Reproducing existing experiments

TODO (Mathieu): Typically a source of motivation: we should move out the material here to a dedicated section..

To understand what kind of analysis information would be required by researchers, the first author attempted to recreate experiments carried out by Guid and Bratko in Computer Analysis of World Chess Champions[6]. Guid and Bratko slightly adapted the open-source program Crafty to attempt a comparison between chess players who have never played against each other by assessing choice of moves and positions with Craftys engine. Guid and Bratko suggest, among other methods, using Blunder rate analysis to rate players. Big mistakes (blunders) are easily detected by a chess engine. Using a -1.00 (the equivalent of losing one pawn) blunder threshold, players with milestone Elos (1900, 2000, 21002800) were selected by the first author and the blunder rate was calculated as the number of blunders according to each of the four engines divided by the number of moves. . The four blunder rates were then averaged and plotted against milestone elo ratings:

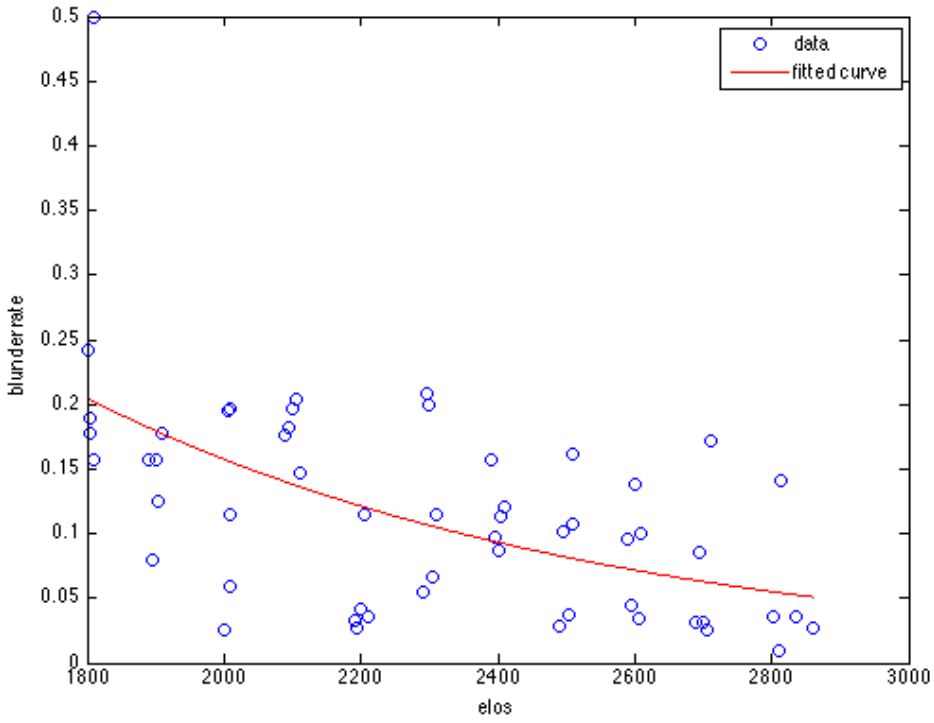


Figure 1: Blunder rate vs Elo

Conclusion: Blunder rate significantly correlates to Elo rating. In this case, Blunder = 2.163*exp(-0013*Elo) However, the deviation from the mean curve is too high. This is largely attributed to the fact that different players have different tactics/approaches and playing methods. For example, one player might have a high blunder rate but still win the game. Another possible cause could be that the Elo rating system might be intrinsically unfair in that players with decent chess skills are ranked lower because they only face opponents with much higher chess skill while poor chess players can get ranked highly because they only face weaker chess players. Guid and Bratko also approached the problem by calculating the expected average error. After five games played by players from each milestone Elo score (from 1800-2800) with the chess engines, a python regex script is used to extract the scores of

top evaluated moves and scores of the move made for each game from the analysis log. The mean error is calculated as = sum of (score of top evaluated move score of move made) / number of moves in the game. This mean error value is then plotted against the elo milestone value:

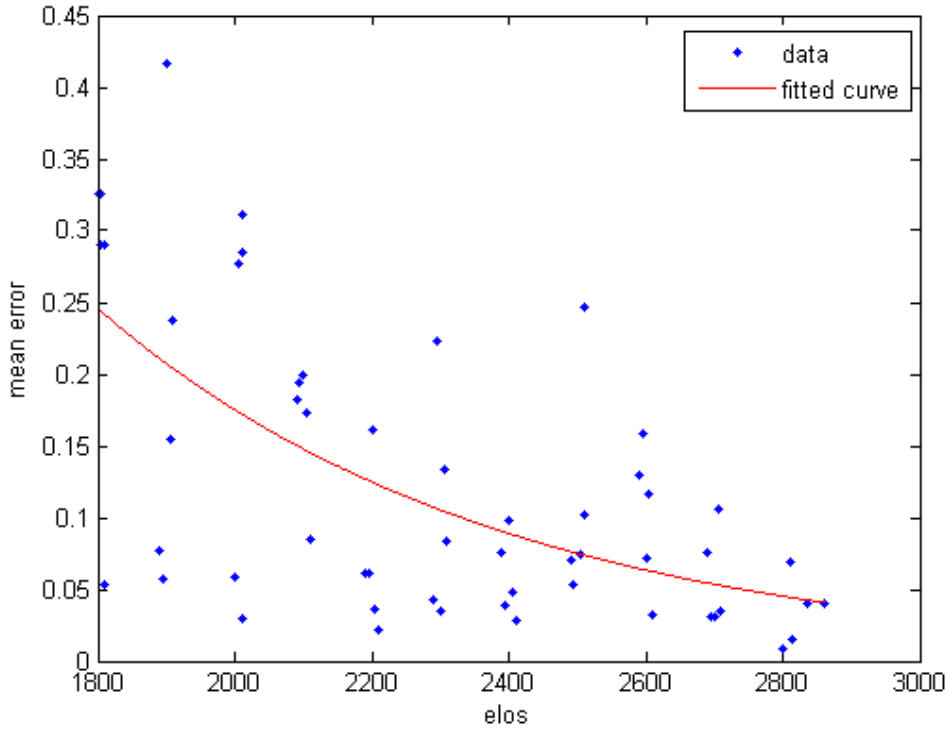


Figure 2: Mean error vs Elo

Conclusion: Mean error might strongly correlate to the Elo rating, as is suggested by the above results. However, to obtain statistically confident results, more data points are needed. This leads us to the penultimate step.

4.1 Understanding how to communicate with the engine

The UCI protocol is published by Stefan-Meyer Kahlen[7] explains that he communication with the engine is done via Standard input

and Standard output PIPEs. The relevant commands that we required for our analysis tool were as follows:

- uci: the first command that tells the engine to use uci
- setoption: this sets the parameters of the engine such as Hash Value, Nullmove value, among others
- ucinewgame: this tells the engine that the next search to be carried out is a different game
- position startpos moves sets up the position described from starting position
- go: tells the engine to start searching on the current position. This takes several options such as depth (search x plies only), nodes, movetime, infinite (search until stop command.
- Stop: stops the engine from search on given position
- Quit. Quits engine.

4.2 Understanding how to read the engine's output

- Id *name and *author. This is sent from the engine after it receives the uci command to identify itself
- Uciok. Sent after id to announce that all info has been sent.
- Readyok. Sent after engine receives isready command indicating that all input has been processed and the engine is awaiting instruction.
- Info. Engine returns all the info on the analyzed move such as depth, time, nodes, principal variations, score of the move, cpuload, etc

- And bestmove. This indicates that the engine has stopped searching and found the move best in this position.

4.3 Choosing between Java, python and C++ to implement analysis tool

I was initially under the impression that Java was the ideal language because of its simplicity in the control and execution of processes and threads, since we were going to deploy the final analysis tool on a cluster of machines. However, since the tool would only require one process instance to run, the use of Threads became redundant. This, along with my lack of conversance with C++ made python the ideal choice.

4.4 Finding a reliable PGN Parser

In order for PGN games to be analyzed by an UCI, they must first be parsed from SAN (Standard Algebraic Notation) to UCI readable format. There are a decent number of open-source PGN parsers out there but very few of them are robust and reliable. Among these few is a python-implemented PGN-parser in the pychess source code. Another free PGN parser out there is pgn-extract. The same program that was used to search for and eliminate duplicates. The pychess pgn parser is designed to support the pychess GUI. It is so intricately entangled into the game that separating the two modules would take a significant amount of time. PGN-extract emerged the better choice here for our purposes. With the following line of code, I was able to parse the collection of 5 million games into UCI compatible format:

$ pgn-extract oOutput.pgn Wuci noresults input.pgn

In this format, one can easily call the analysis tool on Output.pgn

and analyze the 240 million (give or take) moves.

4.5 Implementing the analysis tool

Using the python subprocess module, one can call any outside process such pgn-extract.exe, stockfish.exe, Houdini We designed an executable python script that loads a file of PGN games, communicates with the engines via stdout and stdin PIPEs and sends the moves to be analyzed for each position for each game. The analysis log is then stored in a log file for further analysis. The results can be nicely interpreted by chess and statistics experts.

4.6 Reusability

For the program to be reusable on a cluster, were currently in the process of making it executable through a command-line interface where parameters such as the file of PGN games, the choice of engine, the ply depth, the search length in seconds and others can be set.

Acknowledgments

Foremost we thank the programmers of pgn-extract for full scripting of a robust PGN parser, and those of Stockfish, Houdini, Komodo and Rybka for their engines. The support and advice provided by Guy McC. Haworth and Kenneth W.Regan are also highly appreciated.

References

1. http://www.chess.com/downloads/database+of+games
2. http://chesstempo.com/game-database.html

3. cofy-blog.de
4. http://www.cs.kent.ac.uk/people/staff/djb/pgn-extract/help.html#dupli
5. http://www.sdchess.ru/Engines_UCI_top.htm
6. Guid, M., Bratko, I.: Computer analysis of world chess champions. 2006. en.chessbase.com/news/2006/world_champions2006.pdf
7. Stefan-Meyer Kahlen: The UCI protocol. 2004. wbec-ridderkerk.nl/html/