The Z64C Chess Engine

Software Design Document

Filippo Costa

September 24, 2018

Contents

Preface

1 Abstract

Z64C is a free and open-source chess program written in C99. Rather than using proven techniques in the field of computer chess, such alpha-beta pruning or carefully hand-crafted evaluation function, this program uses MCTS and ANN; in this sense it is similar to the popular LC0 program. The similarities are, however, only superficial:

1. LC0 is based on the AlphaGoZero architecture from Google's Deepmind team; Z64C doesn't.
2. LC0 makes use of traditional CNN as evaluation function while Z64C has more performant heuristics.
3. More generally, Z64C happens to borrow some architecture choices from AZ but is a separate chess program; LC0 is a reverse engineering attemp.
4. LC0 heuristics are trained from millions of games played; Z64C is designed to converge extremely fast, even at the cost of a lower performance.
5. Z64C makes an attempt at being portable and light on system resources, compromising on playing strenght if necessary.

With these points, I believe I have proven my point of Z64C being a strong chess engine, original work, for constrained systems.

2 Features

3 Credits

4 Legal notes

5 Third party licenses

1 Architecture overview

1.1 Board representation

Hundreds of programmers in the history of computer chess have long debated on the optimal data structure to store chess positions in memory and there are several alternatives to choose from. The optimal choice of board representation must satisfy these requirements:

1. Compactness. Chess programs have a need to store thousands of chess positions in memory, so it is mandatory not to occupy unnecessary memory.

2. Access to useful information and speed. Even the most compact board representation of them all would be of little use if it doesn't store data in a fast way. An admittely extreme example of this is a compressed data structure.

These constraints leave bitboards as the only sane option, possibly variations of. The board representation used by Z64C is only 58 bytes wide and fits comfortably in a single cache line of 64 bytes. This ensures optimal performance and minimizes TLB misses.

* location of each piece indexed by piece ID, * game state information, * occupancy bitboard, * piece bitboard,

1.2 Move generation

1.3 Transpositions table

Z64C uses MCTS to look deeper and deeper into lines and tactics, but uses its own version of the original algorithm which tries to maximise cache usage and streams data efficiently to GPU. As we can't remove GPU/CPU communication overhead, we can only resort to smarter algorithms, such that either of the two always has something to do and never has to wait for the other. Designing a good memory layout to the search table is the first step in this direction.

Z64C search trees are a collection of memory chuncks, all of the same size. This size is page aligned for maximum performance and by default is 4kB. Each of these chuncks stores a fixed number of search nodes with the following information:

* Children's weights, so that sampling is easy and local to one node only. * Chess position using bitboards. * Stats about predictions for that particular node.

The key to making Z64C lies actually more on search than evaluation, because search takes advantage of many heuristcs that other engines don't have. So I need an algorithm that, by staying in the CPU and with no cache misses, it has a good prediction of the best nodes to search next.

When pruning the search table after making a move, Z64C should take on the opportunity to shuffle things around and improve data locality. Also, contrary to what MCTS does, it doesn't need to start samling directly on top.

I would like to experiment an open-addressing storing method for moves, such that the space used to store moves is very small. Advantages:

* Reduced storage

1.4 Z64C as a server

The UCI was designed in November 2000 and has the merit of lowering the entry barrier to graphical computer chess. It has several limitations though:

1. Internal inconsistency.
2. Remote Procedure Call via stdin.

Because of these reasons, Z64C implements a different protocol based on network RPC.

1.5 Testing metodologies

2 User manual

2.1 System requirements

2.2 Installation

2.3 `HELP topic`

The `help` command provides brief explanations to the use of Z64C.

2.4 `LOAD session-name`

The `load` command tells Z64C to retrieve for storage a past session. A session includes:
Transpositions table, settings, cluster nodes (if available).

2.5 `MOVE move [...]`

`MOVE` *move in coordinate notation* [moves ...]

2.6 `HELP topic`

2.7 `PURGE`

2.8 `QUIT`

`QUIT`
Immediately exits the program without saving progress.

2.9 `SAVE session-name`

2.10 `SETUP [fen] [move [...]]`

The `SETUP` command changes the board position.

3 Evaluation & Search

3.1 Monte Carlo Tree Search

Training will use the modular architecture of the engine and train the various neural networks with the outputs of the others.

I need to train: * Hot2board * Board2moves * Increment2board

What's the relation between the estimated score [0-1] and the move strenght [0-1]? Given one, one can find the other. So the move strenght is not relative to other moves, but only to the estimated score.

I need an optimal algorithm. Basically, it start playing moves given board positions. I add an exploration component by playing also random or unlikely moves. At every move, I get an estimated score and a move.

So my net is at the same time an actor and a critic: gives me an action (or more than one), and an estimated winning score. How do I train this? I want to bootstrap the winning score. I will train based on the advantage instead, so that it's smarter about it.

Ok, maybe I got it. Using a normal optimizer like Adam, I backtrace part of the net to maximize the estimated score, and I train the estimated score based on the temporal difference learning formula. Now I should be set, I have the training figured out, and I can devote my time to implement (and train!) Z64C v0.2.

Notes on perft: I want to develop a perft so that I don't need to worry about checks. It would be much, much faster! What's the quirk? A move is illegal if, at the next move, there is a pseudolegal move to capture the king.

I need a struct Board where I can extremely rapidly say which moves are in the tensor and if they are legal.

Cavalli: 8 Pedoni: 28 Torri: 14 Alfieri: 13 Re: 8

Architettura REDIS: Sorted set con Id dei search node e priorita'.

Non capisco se dovrei fare l'update delle statistiche... mi serve davvero? Spero davvero di no, perche' altrimenti sarei nei guai. Diventa un magnitudo di complessita' temporale piu' lento. Mi serve che la ricerca sia velocissima. La ricerca deve fare:

* Legality check (pseudo?) Come faccio a gestire una pseudo legality checking? Il problema non e' affatto lo scacco matto, ma lo stallo, che invece e' difficile da prevedere.

Z64C starts search from the top of its search table by creating a visitor. The visitor architecture is extremely easy to parallelize: one can just add more visitors and update the search table in more places at the same time. The visitor works by hammering down the search tree, finding a good minimum while mantaining exploration. The behaviour of visitors is customizable directly from C using function pointers. This allows for different playing styles or - more simply - to explore unsure nodes during training. Exploration routine:

X (*M*ultiplication). blah blah blah blah...

X1.[*D*o stuff] blah blah blah

X2.Terminate the algorithm. ▮

1. Initialization. Create a visitor at the top node. 2. Fall. Descend the tree by choosing a semirandom path. 3. Impact. Evaluate the node at the visitor and add only its children that are considered useful. 4. Pruning. Sometimes Step Impact make us understand that a whole branch of the tree is not viable. 4. Bouncing. Recursively move the visitor up the tree for a certain number of nodes, which is determined by a random value. Go back to stage 2.

Just like MCTS, this algorithm is easily parallelizable, but this one reduces the amount of nodes that need to be visited when the search table grows bigger and bigger, because it takes advantage of proximity of nodes.

The search table also needs other features:

* Pruning. I don't want the search table to just get bigger and bigger. It also must let some branches go when we realize they're not as promising as we thought they would be. * Node prediction. While the GPU is computing, the CPU is choosing the next nodes to feed to the GPU and saving them in a buffer, ready to be streamed in a batch. * Data locality and memory pools. Nodes close in the tree shuld also be close in memory, for better data locality and better chance for cache hits. * Transpositions. Transpositions are stored by Robin Hood hashing with a counter of how many times they were met during search. During pruning I know when transpositions are actually finished and I can kill them. to delete all transpositions and then * Number of children to add at new nodes. Only new children that have a certain probability to improve the current score are kept, those that fall under that are directly removed.

The problem with TTable is that it's on the RAM, so it's slow. I want Z64C to be using CPU 100long-term all tactics and lines explored, which can't possibly fit into a smaller data structure. I also need to figure out a cache-friendly version of TTable though, such that I can fetch subsets of TTable very quickly.

/sectionNeural network architecture

3.2 MCST!

4 Heuristics training

4.1 TD(λ) & TD-Leaf(λ)

* Chess 960 (5* Endgame tables * Mixed opening book and not during training, so that it can learn to use it when useful and not to when it's not.

Critic update algorithm: https://papers.nips.cc/paper/3290-temporal-difference-updating-without-a-learning-rate.pdf

Once I think about the probability distribution, I can use the extra information to: * Better drive game search: search more the more indecisive positions, don't search in clearly winning or losely positions. This favours quiet positions, just like real engines. * Better train the critic. * Can it also replace resoluteness factor? Yes. Instead of thinking about the most aggressive move, it outputs a simpler action space and I can select with the algorithm that I want.

I need to think about the learning process. I have almost everything figured out.

Temporal difference learning: Come modificatore uso: * il numero della mossa * material imbalance Vedi https://pdfs.semanticscholar.org/6e98/bbc9a3dc3ffb5e23abd748bf4e46d1 per migliorare LDleaf.

Procedura di training: Crea una nuova scacchiera dall'inizio e gioca una partita. Questo per ogni istanza, ce n'e' una per thread e sono asincrone (opzionale). Non puo' imparare durante la partita per semplicita'. Durante la partita (giocata con un numero di nodi fissi per partita) non eliminare le transpositions che non servono piu'. values con anche i vari parametri e usi TD64 per calcorare gli state values ottimali. Ci abbini le mosse migliori con una selezione bayesiana delle mosse (non sempre la migliore) a metti tutto in fila al batch per l'allenamento. Allenare il policy e' anche semplice (molto curiosamente), ma devo pensare al TD64. TD64 e' la variante di TDlambda personalizzata per Z64C. TD(64) sara' una funzione con una type signature particolare e che funziona molto bene.

Come funziona TD(64)? Ho una lista di previsioni, un risultato della partita e devo trovare dove ci sono stati errori... Ben cio'! Complicato. Anziche' distribuire equamente la ricompensa, posso toglierla da una parte. E' plausibile che ci siano degli errori in mezzo.

Faccio training sulle mosse di KingBase, e allo stesso tempo alleno con TD(64). Mantengo il learning rate lo stesso e dopo comincio il self-play.

Devo anche usare CTDleaf con lambda 0.7

* Chess 960 (5* Endgame tables * Mixed opening book and not during training, so that it can learn to use it when useful and not to when it's not.

Critic update algorithm: https://papers.nips.cc/paper/3290-temporal-difference-updating-without-a-learning-rate.pdf

Once I think about the probability distribution, I can use the extra information to: * Better drive game search: search more the more indecisive positions, don't search in clearly winning or losely positions. This favours quiet positions, just like real engines. * Better train the critic. * Can it also replace resoluteness factor? Yes. Instead of thinking about the most aggressive move, it outputs a simpler action space and I can select with the algorithm that I want.

I need to think about the learning process. I have almost everything figured out.

Temporal difference learning: Come modificatore uso: * il numero della mossa * material imbalance Vedi https://pdfs.semanticscholar.org/6e98/bbc9a3dc3ffb5e23abd748bf4 per migliorare LDleaf.

Procedura di training: Crea una nuova scacchiera dall'inizio e gioca una partita. Questo per ogni istanza, ce n'e' una per thread e sono asincrone (opzionale). Non puo' imparare durante la partita per semplicita'. Durante la partita (giocata con un numero di nodi fissi per partita) non eliminare le transpositions che non servono piu'. values con anche i vari parametri e usi TD64 per calcorare gli state values ottimali. Ci abbini le mosse migliori con una selezione bayesiana delle mosse (non sempre la migliore) a metti tutto in fila al batch per l'allenamento. Allenare il policy e' anche semplice (molto curiosamente), ma devo pensare al TD64. TD64 e' la variante di TDlambda personalizzata per Z64C. TD(64) sara' una funzione con una type signature particolare e che funziona molto bene.

Come funziona TD(64)? Ho una lista di previsioni, un risultato della partita e devo trovare dove ci sono stati errori... Ben cio'! Complicato. Anziche' distribuire equamente la ricompensa, posso toglierla da una parte. E' plausibile che ci siano degli errori in mezzo.

Faccio training sulle mosse di KingBase, e allo stesso tempo alleno con TD(64). Mantengo il learning rate lo stesso e dopo comincio il self-play.

Devo anche usare CTDleaf con lambda 0.7

* Chess 960 (5* Endgame tables * Mixed opening book and not during training, so that it can learn to use it when useful and not to when it's not.

4.2 Optimal settings during the training phase

4.3 MCTS! variations

4.4 An outline of the training process

5 On performance

Performance is central to the strenght of chess programs. Other factors being equal, faster software running faster will be able to search deeper and consider more lines, which sometimes does win a game. Let's consider the perfect evaluation function δ and its approximation ω.

5.1 Clustering

5.2 Scaling vertically

5.3 Compiler optimizations and low-level details

Compiler optimizations play a huge role in the performance of C programs. Z64C benefits from the following optimizations:
  profile guided optimization