Quiescence Search for Chess

Andrew Nakamoto
University of Washington
anak2004@cs.washington.edu

Abstract

The large branching factor of the game of chess requires aggressive search tree pruning. Quiescence search extends minimax by only exploring moves that have a high likelihood of changing the evaluation of a position to mitigate the horizon problem. We further refine this idea with the addition of the Null Move hypothesis. On a dataset of tactical positions, these improvements result in an accuracy increase over minimax from 0.510 to 0.558 with 3x runtime speedup while searching 5 plies deeper. I then attempt to optimize minimax itself further by memoizing game states.

1 Overview

This project aims to extend the Pacman projects by taking the minimax algorithm with alpha-beta pruning, writing new code to apply it to chess, and then implementing quiescence search to search deeper with less computation. Code available at `https://github.com/asn1814/chessbot573`.

1.1 Chess

Chess provides a much more challenging search environment than Pacman because the branching factor is an order of magnitude higher. A goal with this project was not to pay for compute, so everything is running on my MacBook M1 Max chip. This meant that computation was a bottleneck, so I was very focused on pruning and efficient search.

1.2 Quiescence Search

Quiescence search is an algorithm used to extend search at unstable leaf nodes in minimax game trees. It helps mitigate the horizon problem, especially for rudimentary evaluation functions. Let an unstable node be defined as a node where a single move may result in a large change in evaluation. Then ideally, we only evaluate at "quiescent" stable positions. Then quiescence search can be represented by the following pseudocode:

```
1     function quiescence_search(node, (optional) depth) is
2     if node appears stable or node is a terminal node or depth = 0 then
3         return estimated value of node
4     else
5         (recursively search node children with quiescence_search)
6         return estimated value of children
```

This project explores several variations of quiescence search in an attempt to find a computationally efficient solution that improves performance.

2 Method

2.1 Data and Performance Evaluation

Dataset As quiescence search fundamentally handles unstable positions which are measured in chess by captures and checks, I expect to see the most improvement on highly tactical positions. Therefore I selected a dataset of 2628219 tactics positions available as `tactic_evals.csv` at `https://www.kaggle.com/datasets/ronakbadhe/chess-evaluations/data`.

Each item of the dataset contains a game state, evaluation (in centipawns), and best move (as these are tactics positions, there is a demonstrable "correct move", which is not an assumption that can be made in the majority of chess positions).

Evaluation I chose to evaluate based on best-move accuracy. While centipawns would be the obvious choice for a chess evaluator and quiescence search returns a centipawn value, I focus on improving search rather than evaluation. Runtime is measured with Python's `tqdm` package. As everything is running on my MacBook, this is only really accurate to within a few dozen seconds but serves as a great approximation of computational complexity and ended up working convincingly enough.

2.2 Baselines

I use the leading[1] chess engine Stockfish 17 as a benchmark for perfect performance, and pure minimax search as implemented in the Pacman project as a baseline. While using minimax, I tested two different evaluation functions. See Appendix A for details.

3 Results

3.1 Baselines

See Table 1. Evaluated on 500 tactics positions each. Runtime as "mm:ss". Depth is counted in plies. Minimax without alpha-beta pruning included as test of correctness.

Table 1: Baseline Accuracies

Agent	Depth	Evaluator	Accuracy	Runtime
Minimax	2	Piece value summation	0.466	00:13
Minimax	2	Stockfish 17 (depth 0)	0.454	01:15
Minimax	2	Stockfish 17 (depth 1)	0.460	01:25
Minimax (no $\alpha\beta$)	3	Piece value summation	0.510	11:09
Minimax	3	Piece value summation	0.510	04:24
Stockfish 17	3	-	0.678	00:00
Stockfish 17	10	-	0.788	00:01
Stockfish 17	25	-	0.828	00:45

At low depth, the evaluation function does not have a huge impact on the minimax algorithm, so moving forward I exclusively use piece value summation. Additionally, it is clear that Stockfish's strength comes from a significantly better pruning algorithm that allows it to explore much deeper very quickly (of course the engine binary's search is also better optimized than Python code). This indicates that there is room for quiescence search to allow for deeper exploration and improved accuracy.

3.2 Brute Quiescence Search

An initially obvious definition of an "unstable" state is one where legal actions include captures or checks. At the low depths I'm working with, this will be nearly every position - it's likely a rook or

[1] AlphaZero is better, but Stockfish is open source. For this computationally constrained project, Lc0 and Komodo would not provide different results.

queen can suicide capture a pawn. Concluding that it is therefore not possible to search all the way to a stable state, a simple application of the idea behind quiescence search is to limit the actions we explore to those that cause instability. The fundamental idea is that any non-captures or non-checks will be unlikely to significantly change the evaluation function (this is exactly true in our case as the evaluation function only responds to the number of pieces and whether the king is in checkmate or not). Then this "brute quiescence search" is exactly minimax search where we only consider actions that capture a piece, deliver check, or escape check. I apply it to all leaf nodes of the minimax search.

An initial implementation did not explore moves that escape check; see Appendix B for more analysis and the accuracy differences of the two methods.

Table 2: Brute Quiescence Accuracies

Search	Depth Minimax	Depth Quiescence	Accuracy	Runtime
Minimax	3	-	0.510	04:24
Minimax + Brute quiescence	2	1	0.516	01:24
Minimax + Brute quiescence	2	2	0.532	03:33
Minimax + Brute quiescence	2	3	0.510	10:24
Minimax + Brute quiescence	1	1	0.460	00:05
Minimax + Brute quiescence	1	3	0.446	00:39
Minimax + Brute quiescence	1	5	0.410	04:31

This simple application of quiescence already outperforms the baseline of minimax at depth 3 - it can search deeper to reach higher accuracies while running faster. However, the issue mentioned above can still occur in non-check positions, which is likely what is reducing the accuracy of long chains of quiescence search.

3.3 Null Move Hypothesis Quiescence

To solve this issue, we can lower bound our evaluation as we perform quiescence by assuming that there exists some move that improves our position at least marginally [1]. This is theoretically sound in chess as long as we are not in *zugzwang*, which occurs only rarely (although it is disproportionately likely to exist in tactical positions and puzzles). Then we assume that we can achieve a position no worse than if we skipped our turn. This assumption places a lower bound our quiescence search equal to the evaluation of the current position. We modify the quiescence search discard children that produce worse results than the null move. If no children are better than the null move, we simply return the evaluation of the position. This implementation can be further optimized by checking the lower bound against beta / alpha to prune the game tree [1]. See results in Table 3.

Table 3: General Quiescence Accuracies

Search	Depth Minimax	Depth Quiescence	Accuracy	Runtime
Minimax	3	-	0.510	04:24
Minimax + General quiescence	2	1	0.524	00:57
Minimax + General quiescence	2	2	0.528	00:51
Minimax + General quiescence	2	3	0.558	01:25
Minimax + General quiescence	2	4	0.556	01:26
Minimax + General quiescence	2	5	0.558	01:39
Minimax + General quiescence	2	6	0.558	01:42
Minimax + General quiescence	1	1	0.446	00:03
Minimax + General quiescence	1	3	0.464	00:04
Minimax + General quiescence	1	5	0.474	00:05
Minimax + General quiescence	1	7	0.474	00:06

This immediately fixes the previous issue that arose when all checks and captures are bad moves, and now quiescence search consistently improves with depth instead of being damaged by long continuations. It has the additional benefit of aggressive pruning, with huge corresponding speedups. I actually didn't implement this alpha/beta pruning optimization immediately, see Appendix C to see the naive technique I wrote first and the vast improvement.

As a result of this pruning, runtime and accuracy plateaus after 2 plies of minimax with 3 plies of quiescence, likely because the vast majority of lines have been pruned away. This seems to be the limit of which tactics can be solved purely with captures and checks.

3.4 Move Ordering

We see that the main issue is that the minimax branching factor is huge and therefore quite slow. One way to attempt to fix this is to order the moves we explore so that moves likely to create alpha/beta cutoffs occur first. The canonical solution is a transposition table. This involves complex implementation and bitboards, so I tried to avoid this by hashing positions and the single best move I'd found there, checking this "principal-variation" move first.

Table 4: PV-Move Quiescence Runtimes

Depth Minimax	Depth Quiescence	Accuracy	Runtime (base)	Runtime (PV-Move)
3	-	0.510	04:24	05:11
2	3	0.558	01:25	01:34
1	5	0.474	00:05	00:06

Unfortunately this and several variations I experimented with did not work as the added hashing only increased runtime. It is likely that I am not searching at depths where the PV-move becomes super accurate. There are several other methods to help with this as well, such as internal iterative deepening and a variety of move-ordering heuristics.

4 Discussion

4.1 Further Improvements

Delta pruning An additional pruning step in quiescence search is delta pruning, where we test whether any capture has the possibility to beat alpha: if the current evaluation from the null move hypothesis, plus some large margin (more than a queen's value) then we can conclude that there is no need to explore captures. This is similar to futility pruning.

Beyond tactical positions Quiescence search is fundamentally built for tactical play rather than positional play. A better evaluation function and improved minimax search are better solutions to improve the general strength of the agent. I'd like to try implementing a transposition table next.

4.2 Fail states of quiescence search

Zugzwang *Zugzwang* refers to a situation in chess when a player is forced to make a move that worsens their position. This renders the null move hypothesis incorrect and breaks general quiescence search as a branch containing it may be pruned.

Threat move If we are in a position where the *opponent* has a threatening move, then the null hypothesis is also incorrect - instead, we must make a move that prevents the threat move from occurring. This will break the general quiescence search as the branch may be pruned instead.

5 Conclusion

From a minimax baseline with accuracy 0.510 (04:24 runtime), minimax with brute quiescence search reaches 0.532 accuracy (03:33 runtime), and minimax with generalized quiescence reaches 0.558 accuracy (01:25 runtime). From these experiments, we see that quiescence search provides an essentially free boost to performance, but the most important factor in performance remains the optimization of minimax itself.

References

Got to have some fun digging around in old stuff from the 80s and 90s. Beal's generalized algorithm seems to be the definitive work on the topic. The other three were the next most useful to read through and steal ideas from.

[1] Don Beal (1990). *A Generalized Quiescence Search Algorithm.* Artificial Intelligence, Vol. 43, No. 1, pp. 85-98. ISSN 0004-3702

[2] Adelson-Velskiy, G.M., Arlazarov, V.L., Donskoy, M.V. (1988). *Some Methods of Controlling the Tree Search in Chess Programs.* In: Levy, D. (eds) Computer Chess Compendium. Springer, New York, NY. https://doi.org/10.1007/978-1-4757-1968-0_14

[3] Tony Marsland (1992). *Computer Chess and Search.* Encyclopedia of Artificial Intelligence (2nd ed.) (ed. S.C. Shapiro) pp. 224-241. John Wiley & Sons, Inc., New York, NY. ISBN 0-471-50305-3.

[4] Beal, D. F. (1982). Benefits of minimax search. In *Advances in computer chess* (pp. 17-24). Pergamon.

Appendices

A Evaluation functions

I tested two evaluation functions to confirm that the difference between them was minimal. The first was Stockfish centipawn evaluation with search depth as a hyperparameter and a time limit of 0.01 seconds. The second was a handcrafted heuristic: the difference between the summation of piece count values of each side, with pawns worth 100 centipawns, knights worth 310, bishops worth 320, rooks worth 500, and queens worth 900. A checkmate is returned as infinite centipawns. This is the traditional evaluation method taught to chess beginners and is particularly suited for the tactics dataset because tactics emphasize material gains and losses rather than positional advantages.

B Faulty Brute Quiescence

My initial implementation of brute quiescence search only explored captures and checks. Then this "faulty brute quiescence search" is exactly minimax search where we only consider actions that capture a piece or deliver check. It turns out that this still beats the baseline, but is worse than the corrected "brute quiescence search" implementation described above that includes moves to escape check.

Runtime differences between the two algorithms were negligible.

Table 5: Faulty Brute Quiescence Accuracies

Depth Minimax	Depth Quiescence	Faulty Brute Acc.	Brute Acc.
3	-	0.510	0.510
2	1	0.510	0.516
2	2	0.530	0.532
1	1	0.434	0.460
1	3	0.400	0.446
1	5	0.396	0.410

Longer faulty quiescence searches degrade performance. By limiting the moves we search to only those that are captures or deliver check, we could find positions where the opponent was "forced" into playing a bad move. For example, a quiet position where the opponents only captures are all suicidal would get expanded without the ability for the opponent to play no move at all. The most obvious example is when the opponent is in check - we don't explore how they can simply move out of the way or block the attack, only how they can capture the attacker. Then by including moves that escape check in the search, we see immediate improvement.

C Quiescence Optimization Speedup

When implementing general quiescence search, I initially didn't prune when the null move evaluation was higher than beta (or lower than alpha for minimization nodes) or update alpha with the null move (or beta for minimization nodes). This was a huge mistake! It turns out the number of children pruned by the null move is very very large. This makes sense, because a lot of the captures and checks are random pieces capturing pawns and then being immediately recaptured for a loss of value.

Table 6: General Quiescence Runtime Optimization

Depth Mini.	Depth Quiesc.	Acc.	Unoptimized Runtime	Optimized Runtime
2	1	0.524	01:31	00:57
2	2	0.528	04:54	00:51
2	3	0.558	14:52	01:25
1	1	0.446	00:05	00:03
1	3	0.464	00:51	00:04
1	5	0.474	07:33	00:05

Of course this does not change performance and results in an truly huge reduction in computation. This indicates that the number of positions in which a capture or check are notably worse than the null move are common. Now the quiescence search can extend to much larger depths with ease.