MinMax (Alpha/Beta) for Chess

Students BROUTHEN Kamel & AKEB Abdelaziz
Professor HIDOUCI Walid-Khaled
Lab MinMax (Alpha/Beta) for Chess

1 Benchmark (Heuristics Arena)

To establish a foundational understanding of heuristic performance in chess, we conducted matchups between selected heuristics. The following heuristic indices were used:

- **Heuristic [1]:** Evaluates the number of pieces, occupation, king defense, and castling.
- **Heuristic [3]:** Considers the number of pieces and threats.
- **Heuristic [4]:** Focuses on the number of pieces and board occupation.
- **Heuristic [5]:** Dynamic evaluation: 1 for positions < 25 moves, 4 for 25–35 moves, and 3 for > 35 moves.

Games were played using: `hmin`=4, `hmax`=30, and `largeur`=12 (Only to get reasonable execution time of a full game).

W	B	Moves	Time (s)	W_{time}	B_{time}	Pruning ($\alpha + \beta$)	α	β
1	5	94	10.0	4.6	5.5	160.8K	68.4K	92.4K
1	4	137	99.8	46.8	53.0	1934.1K	1064.0K	870.1K
1	3	84	101.3	28.9	72.4	863.1K	483.0K	380.2K
3	1	171	32.1	26.3	5.9	245.3K	167.7K	77.6K
3	5	218	55.3	32.4	22.9	369.0K	149.4K	219.6K
3	4	86	74.7	57.9	16.8	587.6K	240.7K	346.8K
4	5	184	67.0	38.3	28.7	1310.9K	795.9K	515.0K
4	3	114	54.0	16.4	37.5	583.5K	261.8K	321.7K
4	1	115	65.6	38.6	26.9	1321.8K	818.1K	503.8K
5	4	61	105.0	49.0	55.9	1926.7K	1058.3K	868.4K
5	1	117	19.4	11.5	7.9	284.2K	149.4K	134.8K
5	3	85	105.7	37.5	68.3	740.9K	399.2K	341.7K

Table 1: **Heuristic Matchups — Moves, Time, and Pruning with α and β.**

W	*B*	Alternatives		Mean Depth		Max Depth		Score	
1	5	18 ± 12	26 ± 6	9 ± 3	8 ± 3	12 ± 4	11 ± 4	-9 ± 11	-16 ± 16
1	4	28 ± 7	13 ± 13	9 ± 6	9 ± 6	10 ± 8	10 ± 7	28 ± 18	23 ± 18
1	3	31 ± 14	28 ± 10	14 ± 4	14 ± 5	18 ± 5	18 ± 5	-13 ± 22	-22 ± 26
3	1	26 ± 10	14 ± 11	8 ± 4	8 ± 3	10 ± 6	10 ± 6	17 ± 20	14 ± 21
3	5	18 ± 13	26 ± 7	8 ± 3	8 ± 3	11 ± 5	11 ± 4	-2 ± 14	-4 ± 14
3	4	25 ± 10	26 ± 13	11 ± 5	12 ± 5	16 ± 7	16 ± 6	-9 ± 8	-14 ± 15
4	5	12 ± 13	27 ± 9	7 ± 5	7 ± 5	9 ± 7	9 ± 7	-17 ± 21	-18 ± 20
4	3	17 ± 17	24 ± 8	9 ± 5	9 ± 5	12 ± 7	12 ± 7	-15 ± 9	-21 ± 17
4	1	25 ± 10	17 ± 11	10 ± 5	10 ± 5	12 ± 8	13 ± 7	11 ± 18	9 ± 14
5	4	34 ± 6	25 ± 12	13 ± 6	13 ± 6	16 ± 7	17 ± 7	16 ± 19	9 ± 10
5	1	34 ± 9	17 ± 12	9 ± 3	9 ± 3	12 ± 5	13 ± 5	20 ± 22	14 ± 21
5	3	39 ± 10	22 ± 12	13 ± 4	14 ± 4	18 ± 5	19 ± 4	14 ± 20	10 ± 22

Table 2: **Heuristic Matchups — Detailed Feature Pairs.**
For each feature (Alternatives, Mean Depth, Max Depth, Score), the white (W) and black (B) values are displayed as Mean ± Std.

1.1 Interpretation

The results clearly indicate that **Heuristic [5]** dominates the matchups and wins all games. Its dynamic evaluation—adjusting based on match progress—provides a decisive advantage over the other heuristics.

2 Dynamic Hmax Based on Material Imbalance (Idea)

The original algorithm explores nodes up to a fixed `hmax` depth for unstable positions (where pieces are captured), risking inefficiency in lopsided scenarios (e.g., losing a queen for a pawn). To optimize, we propose dynamically adjusting `hmax` for each node based on the **material imbalance** between players after a move. When a node's evaluation exceeds a predefined threshold (e.g., a significant material advantage like ±5 pawn units), we **reduce `hmax` locally for that branch**. This ensures the algorithm prioritizes depth only in positions where instability matters, i.e., where material balances are close enough that further captures could plausibly alter the outcome. Conversely, in lopsided positions (e.g., a player losing a rook with no compensation), the algorithm truncates exploration earlier, preserving computational resources for more balanced branches.

Example:

- In a branch where White sacrifices a queen for a pawn (-8 material), dynamic `hmax` truncates exploration early, saving computation.
- In a balanced unstable position (e.g., trading knights), the original `hmax` is maintained for thorough analysis.

2.1 Addressing Concerns About Calculated Sacrifices

The dynamic `hmax` ensures that even in cases of intentional sacrifices (e.g., losing a queen for tactical gain), the algorithm retains sufficient depth to detect compensation (e.g., forced checkmates, positional dominance). While material loss reduces `hmax`, critical follow-up moves are still explored within the truncated depth window.

Example:

- Sacrificing a queen (-9 material) triggers a reduced `hmax`=3, but if a forced checkmate ($+M3$) is found within those 3 plies, the bot recognizes the tactical win.

- Blunders (e.g., losing a queen with no follow-up) are truncated early, saving computation.

2.2 Why It Works:

- **Short Forcing Lines:** Tactical sequences (checkmates, traps) often resolve within 3–4 plies, fitting the reduced `hmax`.
- **Smart Evaluation:** The bot prioritizes checkmates/positional gains over raw material, even in shortened branches.
- **Balanced Efficiency:** Focuses resources on uncertain positions, ignoring hopeless ones.

This balances speed and accuracy, respecting sacrifices without over-investing in dead ends.

3 MinMax (α/β) Parallelization (Implementation)

3.1 Parallelization Technique

In this subsection, we introduce a parallelization technique for the MinMax algorithm using OpenMP. The key idea is to parallelize the exploration of the first n levels of the MinMax tree. This parallelization approach offers a tradeoff: while it reduces opportunities for alpha-beta pruning, it results in a significant gain in execution time, especially when large numbers of nodes are processed. The goal is to find a "sweet spot" where the performance benefits from parallelization outweigh the loss in pruning efficiency.

We focus on a depth threshold, `PARALLEL_DEPTH`, and a minimum number of nodes, `MIN_NODES_PARALLEL`, to determine when to parallelize the evaluation. If the number of nodes exceeds `MIN_NODES_PARALLEL` and the depth is less than or equal to `PARALLEL_DEPTH`, the algorithm will parallelize the evaluation of the nodes at that depth using dynamic scheduling in OpenMP. This allows the computation to proceed in parallel, improving efficiency.

Below is the implementation of the parallelized MinMax algorithm using OpenMP:

```c
#include <omp.h>

int minmax_ab(struct config *conf, int mode, int h, int alpha, int beta, int largeur,
              int numFctEst, int npp, int *profMax) {

    const int PARALLEL_DEPTH = 3;
    const int MIN_NODES_PARALLEL = 6;
    int n, i, score, score2, npc, prof_atteinte;
    struct config T[100];

    npc = npieces(conf);
    *profMax = 0;
    if (feuille(conf, &score))
        return score;

    if (h >= hmin && (npp == npc || h == hmax))
        return Est[numFctEst](conf);

    if (mode == MAX) {
        generer_succ(conf, MAX, T, &n);
        if (largeur != +INFINI) {
            if (h <= PARALLEL_DEPTH && n >= MIN_NODES_PARALLEL) {
                #pragma omp parallel for schedule(dynamic)
                for (i = 0; i < n; i++)
                    T[i].val = Est[numFctEst](&T[i]);
            } else {
                for (i = 0; i < n; i++)
                    T[i].val = Est[numFctEst](&T[i]);
            }
            qsort(T, n, sizeof(struct config), confcmp321);
            if (largeur < n) n = largeur;
        }
        score = alpha;

        if (h <= PARALLEL_DEPTH && n >= MIN_NODES_PARALLEL) {
            #pragma omp parallel
            {
                int local_score = -INFINI;

                #pragma omp for schedule(dynamic) nowait
                for (i = 0; i < n; i++) {
                    int local_prof, temp_score;
                    int should_skip = 0;

                    #pragma omp critical
                    {
                        should_skip = (score >= beta);
                    }

                    if (should_skip) continue;

```

```
                    temp_score = minmax_ab(&T[i], MIN, h+1, score, beta,
                                           largeur, numFctEst, npc, &local_prof);
                    local_prof++;

                    if (temp_score > local_score) local_score = temp_score;

                    #pragma omp critical
                    {
                        if (local_prof > *profMax) *profMax = local_prof;
                        if (local_score > score) {
                            score = local_score;
                            if (score >= beta) {
                                #pragma omp atomic
                                nbBeta++;
                            }
                        }
                    }
                }
            }
        } else {
            for (i = 0; i < n; i++) {
                score2 = minmax_ab(&T[i], MIN, h+1, score, beta, largeur,
                                   numFctEst, npc, &prof_atteinte);
                prof_atteinte++;
                if (prof_atteinte > *profMax) *profMax = prof_atteinte;
                if (score2 > score) score = score2;
                if (score >= beta) {
                    nbBeta++;
                    return score;
                }
            }
        }
    } else {
        generer_succ(conf, MIN, T, &n);
        if (largeur != +INFINI) {
            if (h <= PARALLEL_DEPTH && n >= MIN_NODES_PARALLEL) {
                #pragma omp parallel for schedule(dynamic)
                for (i = 0; i < n; i++)
                    T[i].val = Est[numFctEst](&T[i]);
            } else {
                for (i = 0; i < n; i++)
                    T[i].val = Est[numFctEst](&T[i]);
            }
            qsort(T, n, sizeof(struct config), confcmp321);
            if (largeur < n) n = largeur;
        }
        score = beta;

        if (h <= PARALLEL_DEPTH && n >= MIN_NODES_PARALLEL) {
            #pragma omp parallel
            {
                int local_score = +INFINI;
```

```
                #pragma omp for schedule(dynamic) nowait
                for (i = 0; i < n; i++) {
                    int local_prof, temp_score;
                    int should_skip = 0;

                    #pragma omp critical
                    {
                        should_skip = (score <= alpha);
                    }

                    if (should_skip) continue;

                    temp_score = minmax_ab(&T[i], MAX, h+1, alpha, score,
                                           largeur, numFctEst, npc, &local_prof);
                    local_prof++;

                    if (temp_score < local_score) local_score = temp_score;

                    #pragma omp critical
                    {
                        if (local_prof > *profMax) *profMax = local_prof;
                        if (local_score < score) {
                            score = local_score;
                            if (score <= alpha) {
                                #pragma omp atomic
                                nbAlpha++;
                            }
                        }
                    }
                }
            }
        } else {
            for (i = 0; i < n; i++) {
                score2 = minmax_ab(&T[i], MAX, h+1, alpha, score, largeur,
                                   numFctEst, npc, &prof_atteinte);
                prof_atteinte++;
                if (prof_atteinte > *profMax) *profMax = prof_atteinte;
                if (score2 < score) score = score2;
                if (score <= alpha) {
                    nbAlpha++;
                    return score;
                }
            }
        }
    }

    if (score == +INFINI) score = +100;
    if (score == -INFINI) score = -100;
    return score;
}
```

3.2 Parallelization Results

To assess the impact of parallelization on the MinMax algorithm, we compared the performance of each heuristic against itself in both sequential and parallelized modes using different values for `PARALLEL_DEPTH`. For each configuration, we measured both the execution time and the amount of alpha-beta pruning achieved. Games were played using: `hmin`=4, `hmax`=30, and `largeur`=16-20 (Only to get reasonable execution time of a full game). Below is a table summarizing the results:

Heuristic	Parallel Depth	Execution Time (s)	Acceleration	Pruning	Pruning Loss (%)
1	seq	286.3	1.0	4013442	0.0
1	1	56.0	**5.11**	658416	**83.59**
1	2	53.6	**5.34**	659202	**83.57**
1	3	57.3	**4.99**	657668	**83.61**
1	4	50.0	**5.72**	660487	**83.54**
3	seq	314.0	1.0	1268687	0.0
3	1	83.6	**3.75**	504238	**60.25**
3	2	83.6	**3.75**	504662	**60.22**
3	3	82.5	**3.80**	505492	**60.15**
3	4	78.7	**3.98**	505228	**60.17**
4	seq	195.2	1.0	3136643	0.0
4	1	44.7	**4.36**	511640	**83.68**
4	2	45.5	**4.29**	512885	**83.64**
4	3	43.8	**4.45**	511739	**83.68**
4	4	44.1	**4.42**	510604	**83.72**
5	seq	119.4	1.0	1033656	0.0
5	1	28.8	**4.14**	328085	**68.25**
5	2	27.3	**4.37**	329092	**68.16**
5	3	27.6	**4.32**	328589	**68.21**
5	4	29.2	**4.08**	328321	**68.23**

Table 3: Performance results comparing sequential and parallelized execution for different heuristics.

While we observe a significant loss in pruning for the parallel execution, the execution time of the algorithm improves considerably. The acceleration factor shows a strong improvement, with execution times decreasing drastically for all heuristics in parallel modes compared to the sequential execution.

4 Visual Enhancement

We offer the option of displaying chess pieces as emojis in the terminal, instead of using text-based identification. This enhancement improves clarity and makes it easier to follow simulated games or play against the computer. The feature is easily configurable by setting the main variable `int affich_emoji = 1;`.

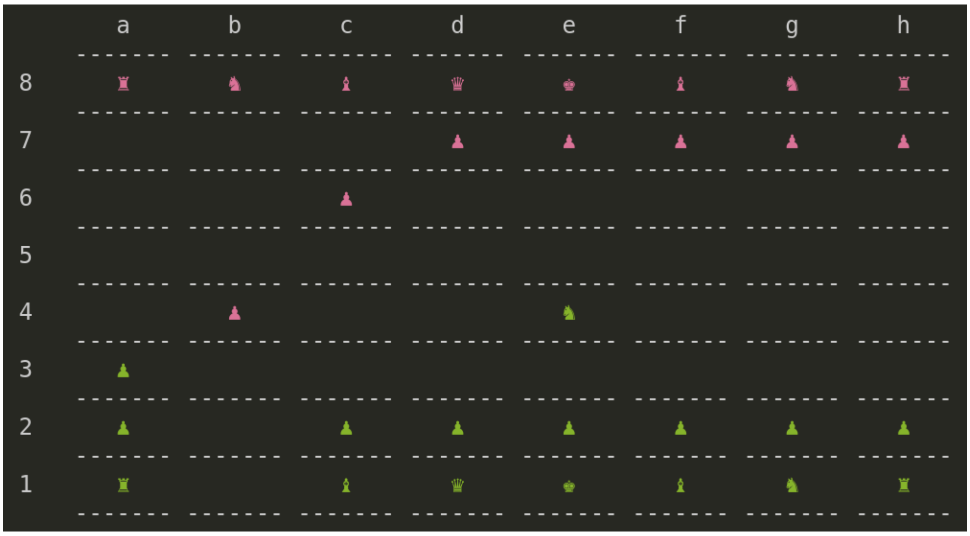

Figure 1: Chess board display in terminal with chess pieces emojis

5 Reference

The codebase for this project is available at the following GitHub repository:

`https://github.com/BrouthenKamel/chess-horizon`