1.5em 0pt

GIST AGI Spring 2025 - HW 2

Optimizing the position heuristic for the `SimpleMCTS` algorithm

20258009, Matouš Bohoněk, bohonmat@gm.gist.ac.kr

The goal of this project was to create a working intelligent system that generates an AI agent for modified `Othello` stages in JavaScript. The implementation of the final intelligent system can be found here.

> **Problem 1: The underlying algorithm**
>
> For this project, I reuse my previously implemented `SimpleMCTS` algorithm:

1 Base algorithm overview

The reason is that the algorithm is as general for pretty much any environment that we can represent as some tree search. This means that the base algorithm without any additional features already works perfectly fine on any `Othello` stage as long as we have functions giving us valid moves and a clear way to evaluate the board. This functionality is already provided in the API. As a quick refresher, here is the algorithm:

Algorithm 1 Monte Carlo Move Selection

Require: *board*, *player*, *opponent*, *timeLimit*
timePerMove ← *timeLimit*/number of *validMoves*
bestMove ← null
bestWinRate ← null
for each *move* in *validMoves* **do**
    startTime ← CurrentTime()
    MakeMove(*move*)
    wins ← 0
    games ← 0
    while CurrentTime() - *startTime* < *timePerMove* **do**
        if RandomPlayout() = WIN **then**
            wins ← *wins* + 1
        end if
        games ← *games* + 1
    end while
    RevertMove(*move*)
    winRate ← *wins*/*games*
    if *winRate* > *bestWinRate* **then**
        bestMove ← *move*
        bestWinRate ← *winRate*
    end if
end for
return *bestMove*

1.1 Other heuristics

Just like in the first project, we won't use any additional heuristics. The reason is that they take too long to compute, which drastically reduces the number of simulations we can achieve, and this outweighs the added benefit of these heuristics. Getting a weighted random move selection is pretty straightforward and computationally inexpensive.

> **Problem 2: Intelligent system**
>
> In project 1 we saw that using the positional heuristic can greatly improve the performance. This means that instead of choosing next moves in the MCTS rollout by pure random choice, using a weighted random move selection can "steer" the exploration into more promising areas. However, the modified stages raise the question of how to obtain an effective positional heuristic. Since the board can differ in size, have some cells blocked, and the rules can change, using a single weight table does not make much sense. Therefore, we want to find a way to tailor this table to the current stage:

2 Optimizing the position heuristic

How can we find a good position weight table? One naive approach would be to generate a random table, have it play against another random one, and keep the one that wins more often. But even if we limited ourselves to discrete values from some interval, this would still require a huge amount of simulation time. Standard optimization methods like hill-climbing might not work well here, since the problem has a very high dimensionality — we have many cells we can tweak, and figuring out how much to change each one is non-trivial.

One method that handles high-dimensional spaces well, can be stopped at any point, and doesn't assume much about the problem is the genetic algorithm (GA).

2.1 Genetic algorithms

Genetic algorithms work by maintaining a population of candidate solutions. In each step (called a generation), we evaluate the population and keep a portion of the best candidates — the "elite" (in this case, position weight tables). Then we generate new candidates by cross-breeding members of the elite and combining them with some randomness (mutation). We repeat this process for as long as we want. Ideally, the population improves with each generation and converges to better solutions over time.

Here's a simple GA pseudocode:

Algorithm 2 Basic Genetic Algorithm

Require: *populationSize*, *mutationRate*, *eliteSize*, *maxGenerations*

population ← RandomInitPopulation(*populationSize*) ▷ Array of pairs (individual, fitness)
for *gen* ← 1 *to maxGenerations* **do**
    population ← EvaluatePopulation(*population*)
    nextGeneration ← SelectElite(*population*, *eliteSize*)
    while Size(*nextGeneration*) < *populationSize* **do**
        *parent*1, *parent*2 ← SelectParents(*nextGeneration*)
        child ← Crossover(*parent*1, *parent*2)
        child ← Mutate(*child*, *mutationRate*)
        nextGeneration ← Append(*nextGeneration*, *child*)
    end while
    population ← *nextGeneration*
end for

There are quite a few hyperparameters to choose from. What's the ideal balance between *populationSize* and *maxGenerations*? How large should the elite group be? How often should individuals mutate? Unfortunately, there's no universal answer — each problem tends to benefit from its own unique set of hyperparameters. We'll touch on this more in a later section, where we explore some basic hyperparameter tuning (which, spoiler alert, turns out to be harder than expected).

2.2 Population evaluation

How do we evaluate an individual in our population? Usually, you'd have some kind of value function (called fitness) that takes an individual and returns a score — for example, how far a robot can walk. But in our case, it's not so clear. How do you measure the quality of a position weight table? Easy: let them compete! We can have each table play against every other one and count how many wins it gets — that's its fitness.

The downside? This process grows quadratically with population size, which makes it slow for large populations. So how can we speed it up? One idea is to evaluate each individual only against a random subset of the population. The subset size can be whatever we want — for example, 10% or 20% of the population. This doesn't change the theoretical complexity, but it helps a lot in practice by reducing the constant factor.

As mentioned before, there's no universal recipe for the best combination of population size, mutation rate, and so on. It's even harder to tune things when the whole system includes a lot of randomness (thanks to both the GA and the game outcomes). Still, I wanted at least a rough idea of what works well, so I ran some tests. For each hyperparameter setting, I evolved two tables — this way we reduce the risk that one lucky run skews the results.

Here's a table showing the tested settings and their winrates in a tournament (using the provided tournament function):

Population Size	Eval Strategy	# of Generations	Score
10	Rnd 0.1	4,383	50
10	Rnd 0.1	4,393	43.5
60	Rnd 0.1	874	76.1
60	Rnd 0.1	884	60.9
120	Rnd 0.1	25	59.8
120	Rnd 0.1	25	55.4
10	Rnd 0.2	2,259	53.3
10	Rnd 0.2	2,213	51.1
60	Rnd 0.2	57	68.5
60	Rnd 0.2	58	60.9
120	Rnd 0.2	13	68.5
120	Rnd 0.2	13	65.2
10	full	1,023	56.5
10	full	1,020	51.1
60	full	24	65.2
60	full	24	60.9
120	full	5	58.7
120	full	5	53.3

Table 1: Test results

The label "Rnd 0.1" means each individual was evaluated against a random sample of 10% of the population. To ensure fairness, each match was played twice: once with each individual playing as Black and as White. The full evaluation algorithm is shown below:

Algorithm 3 Round-Robin Fitness Evaluation

Require: Population P of size N
for $i \leftarrow 1$ **to** $N - 1$ **do**
  for $j \leftarrow i + 1$ **to** N **do** ▷ Individual i plays as BLACK against j
    $result \leftarrow$PlayGame$(P[i], P[j])$
    if $result = 1$ **then**
      $P[i]$.fitness $\leftarrow P[i]$.fitness $+ 1$
    else
      $P[j]$.fitness $\leftarrow P[j]$.fitness $+ 1$
    end if
    ▷ Individual i plays as WHITE against j
    $result \leftarrow$PlayGame$(P[j], P[i])$
    if $result = 1$ **then**
      $P[j]$.fitness $\leftarrow P[j]$.fitness $+ 1$
    else
      $P[i]$.fitness $\leftarrow P[i]$.fitness $+ 1$
    end if
  end for
end for

This evaluation setup means that once your random subset size goes above 50% of the population, you're not gaining any speed advantage — you're basically evaluating most pairs anyway. That's why I stuck to 10% and 20% for the subset sizes.

To make the results easier to digest, Figure 1 shows a boxplot of the outcomes across different settings:

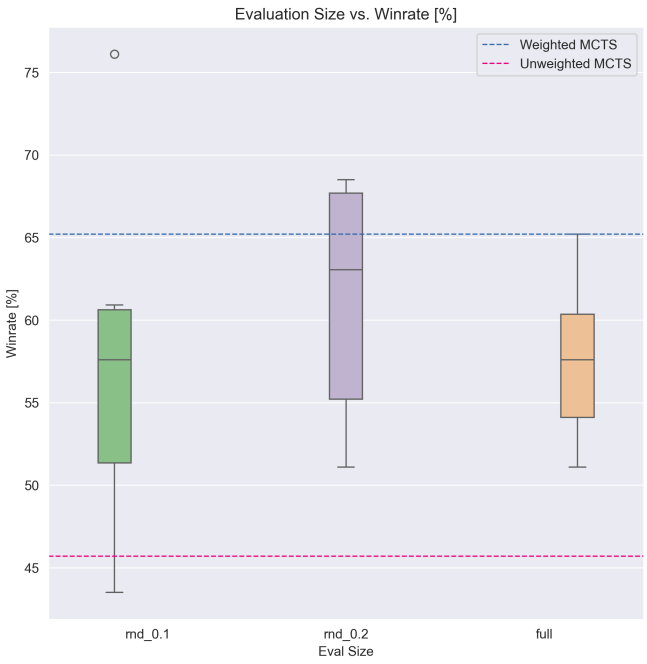

Figure 1: Results of testing different evaluation sizes

Interestingly, it looks like the sweet spot is somewhere in the middle: you're doubling the number of generations compared to full evaluation, but still testing against a decent sample of opponents.

Figure 2 breaks down the effect of different population sizes:

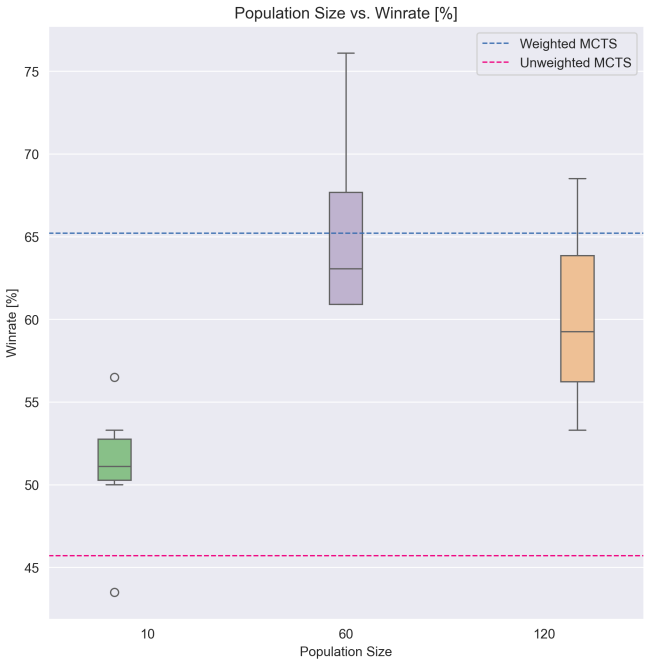

Figure 2: Results of testing different population sizes

Again, it seems like the middle ground is best — small enough to evolve quickly, but large enough to get some diversity.

For both experiments, I also compared the evolved agents to the default `SimpleMCTS`, both with and without hand-crafted position weights (the same ones we used in the first project):

$$\begin{bmatrix} 100 & 10 & 80 & 40 & 40 & 80 & 10 & 100 \\ 10 & 8 & 20 & 20 & 20 & 20 & 8 & 10 \\ 80 & 20 & 40 & 40 & 40 & 40 & 20 & 80 \\ 40 & 20 & 40 & 40 & 40 & 40 & 20 & 40 \\ 40 & 20 & 40 & 40 & 40 & 40 & 20 & 40 \\ 80 & 20 & 40 & 40 & 40 & 40 & 20 & 80 \\ 10 & 8 & 20 & 20 & 20 & 20 & 8 & 10 \\ 100 & 10 & 80 & 40 & 40 & 80 & 10 & 100 \end{bmatrix}$$

2.2.1 Strategy in population evaluation

How exactly should the individuals (i.e., position weight tables) compete against each other? Ideally, we'd want to test them in the exact setting they'll be used in later — meaning inside our `SimpleMCTS` agent. But the problem is that this setup is slow, and the time cost might outweigh the benefits. Fewer generations could hurt performance more than evaluating in a slightly simplified environment.

So, what are our alternatives? Simpler strategies like using a deterministic positional player or a weighted random player can work. Another option is to use a mix of both, where, for each pair of

compared individuals, we play one game using positional strategy and one game using the weighted random strategy.
To cover all bases, I tested four different evaluation strategies:
- Deterministic (Det)
- Weighted Random (Rnd)
- Mixed (Mix)
- `SimpleMCTS` with limited rollouts (MCTS-N)

To make the MCTS-based strategy somewhat usable, I limited the number of rollouts per move to just two. This made it barely fast enough to work with small populations. Table 2 shows the performance of each strategy:

Eval_Strategy	# of Generations	Score
Det	49	68.8
Det	50	63.7
Det	49	58.8
Mix	21	68.8
Mix	21	66.3
Mix	21	60
Rnd	44	73.8
Rnd	44	67.5
Rnd	44	63.7
MCTS4	35	60
MCTS4	34	47.5
MCTS4	34	42.5
MCTS8	5	70
MCTS8	5	62.5
MCTS8	5	32.5

Table 2: Number after MCTS means the population size, all other strategies were evaluated with population of size 60 and full evaluation

Figure 3 visualizes the results:

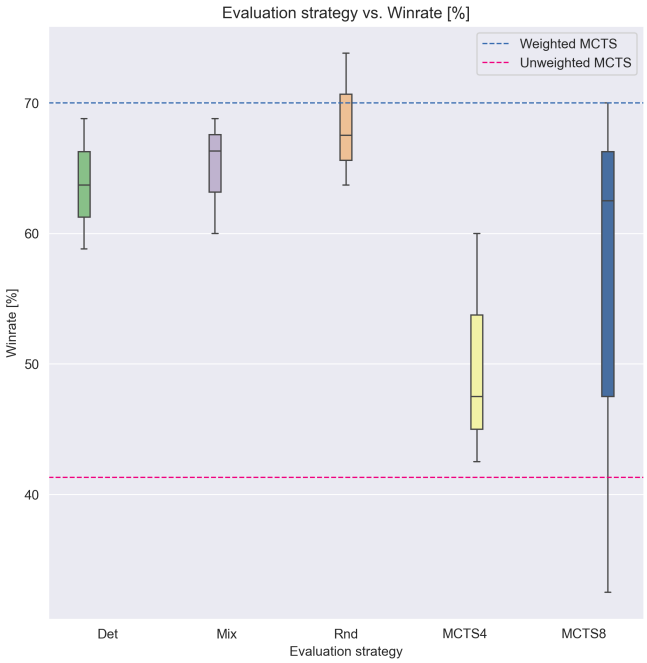

Figure 3: Results of testing different population evaluation strategies

What we see is pretty intuitive: the simple strategies — Det, Mix, and Rnd — perform similarly well. The MCTS-based strategies (MCTS4 and MCTS8) lag behind, with the exception of one lucky outlier in MCTS8. That's actually reasonable. The weighted random player is close enough in behavior to MCTS but runs much faster, so we get more generations and bigger populations — a tradeoff that clearly pays off in this context.

2.3 Elite selection

After evaluating the current generation, we want to eliminate the bottom $N\%$ of individuals. For this project, I simply chose $N = 70$ and didn't experiment with other values, as I didn't expect any particularly interesting insights from changing it. The top 30% of individuals automatically make it into the next generation. Using this elite group, we generate new individuals through crossover to refill the population.

2.4 Crossover and parent selection

New individuals are created using crossover. There are many ways to implement this, but the simplest method is to go through each cell in the position weight table and randomly decide whether to take the value from parent 1 or parent 2.

As for parent selection, I use a quick random tournament. For each parent, I randomly pick 5 candidates from the elite and select the one with the highest fitness. This ensures we're crossbreeding the best of the best.

2.5 Mutation

To avoid getting stuck in local minima — especially if the initial population is weak — we introduce random mutation. Every cell in a new individual has a small chance of being overwritten by a new random value. I use a 1% mutation rate, which is a common default in genetic algorithms.

2.6 Random value initialization and mutation

What values should each cell be initialized (and mutated) with? Because of the way we use these weights for weighted random move selection, we avoid negative values. Random values between 0 and 100 might sound reasonable, but values close to 0 would make certain moves nearly impossible. Since we don't want to restrict exploration too much, I instead use values from the interval (5, 100).

2.6.1 Symmetry

The standard `Othello` board has 64 cells, but 4 are filled at the start, leaving 60 to optimize — quite a high-dimensional problem. Can we reduce that? If we look at the board stages provided, we'll notice that every board is symmetric across quadrants. By assuming this symmetry, we only need to optimize one quadrant and mirror (or rotate to be exact) it to the rest of the board. That reduces the number of parameters by a factor of four.

3 Results

So, what does an "optimal" position weight table look like, according to the GA? It turns out to be fairly similar to the standard hand-tuned table. That suggests the algorithm learned which positions are actually important:

$$\begin{bmatrix} 99 & 8 & 84 & 68 & 71 & 91 & 9 & 99 \\ 9 & 7 & 59 & 18 & 38 & 92 & 7 & 8 \\ 91 & 92 & 96 & 9 & 38 & 96 & 59 & 84 \\ 71 & 38 & 38 & 76 & 76 & 9 & 18 & 68 \\ 68 & 18 & 9 & 76 & 76 & 38 & 38 & 71 \\ 84 & 59 & 96 & 38 & 9 & 96 & 92 & 91 \\ 8 & 7 & 92 & 38 & 18 & 59 & 7 & 9 \\ 99 & 9 & 91 & 71 & 68 & 84 & 8 & 99 \end{bmatrix}$$

The GA-optimized table beat the standard hardcoded one 5:3 on the regular 8x8 board.
On the smaller 6x6 board, it also won 5:3.
On the partial c-squares 8x8 stage, they performed evenly — not sure why. Here's the weight table optimized for that stage:

$$\begin{bmatrix} 74 & 0 & 100 & 5 & 91 & 93 & 10 & 74 \\ 10 & 9 & 7 & 9 & 19 & 85 & 9 & 0 \\ 93 & 85 & 42 & 72 & 73 & 42 & 7 & 100 \\ 91 & 19 & 73 & 18 & 18 & 72 & 9 & 5 \\ 5 & 9 & 72 & 18 & 18 & 73 & 19 & 91 \\ 100 & 7 & 42 & 73 & 72 & 42 & 85 & 93 \\ 0 & 9 & 85 & 19 & 9 & 7 & 9 & 10 \\ 74 & 10 & 93 & 91 & 5 & 100 & 0 & 74 \end{bmatrix}$$

4 Generalization to Unseen Stage Configurations

As mentioned earlier, the `SimpleMCTS` algorithm is general-purpose — it works for any `Othello` stage, or even for any environment that can be modeled as a tree. This means the Genetic Algorithm is purely a performance enhancer; it is not required for the agent to function.

5 Challenges Faced and Future Improvements

One of the main challenges was selecting optimal hyperparameters for the Genetic Algorithm. This is difficult because evaluating the performance of a given configuration is noisy: randomness from both the GA process and the `SimpleMCTS` rollouts makes it hard to get consistent results. A proper evaluation would require far more than 2 or 3 samples per setting, and ideally multiple games per evolved agent.

As discussed in the first project, both `SimpleMCTS` and the GA could benefit significantly from multithreading — which would allow for larger populations and more simulations within reasonable time limits.

6 Performance Against Built-in Strategies

Since even a vanilla `SimpleMCTS` (without any positional heuristics) can consistently beat all the built-in strategies, it's no surprise that our tuned version with GA-enhanced weights outperforms them with ease.

7 Implications for General Intelligence

While this project doesn't directly contribute to AGI, it illustrates how general-purpose optimization (like Genetic Algorithms) can be combined with adaptive decision-making strategies in complex environments. That kind of integration — of search, learning, and generalization — is essential in AGI research.

This work probably won't lead to agents that cure cancer, but it's a solid example of optimizing behavior in a system where explicit rules or objectives are hard to define.