Optimal Setup for Pool Variant: Final Project Report

Jack Weigl
weigl013@umn.edu

1 Abstract

Optimizing the initial state of a game is a niche subset of optimization problems. This is a particularly challenging problem when our objective function is unknown. We aimed to contribute to the underdeveloped framework for approaching such a problem. We explore the use of an evolutionary algorithm to optimize the initial state of a fictitious pool variant, Battle Pool. We leverage domain knowledge to discretize our state space and use Monte Carlo Tree Search to evaluate fitness. In a rugged fitness environment, we employ techniques such as a dynamic mutation standard deviation, an adaptive mutation probability, and a dynamic constraint violation penalty multiplier. Our evolutionary algorithm exhibited convergence toward feasible solutions that offer an advantage over an opponent with a random setup.

2 Introduction

"Battle Pool" is a fictitious variant of billiards. A few oddities set Battle Pool aside from well-known variants. For one, Battle Pool is played with two cue balls. For two, we divide the table into segments. One segment is for the first player, neutral territory, and the second player, respectively.

The game takes part in two phases: setup and play. In the first phase, one player is assigned seven solids, and the other player, seven stripes. Players are free to arrange their balls on their side as they see fit. In the second phase, players take turns directing attacks toward their opponent's side. Players may freely place their cue ball within the backmost portion of their side but must attack outwards. Exchanges continue until a player has no balls left on their side.

A reward for sinking a ball into a pocket is familiar in billiards. In Battle Pool, the outcome of sinking a ball into a pocket depends on the location. A corner pocket of your own side simply removes a ball from play. A corner pocket of your opponent's side removes a ball from play and grants an extra turn. A side pocket of the neutral territory grants you the opportunity to retrieve the ball, place it on your own side, and take an extra turn. Sinking balls into pockets is not necessarily the best move; keeping balls in play may be in your interest.

Additional rules of play are relevant to the balance of the game. We grant the player who is at the disadvantage of receiving the first attack an opportunity to prepare their setup based on their opponent's setup. That is, the player who attacks first must finalize their setup first. Preparing based on your opponent's side allows you to strike a balance between offensive and defensive capabilities. You may consider weaknesses in your opponent's setup and prepare a line of sight for exploitation. You may anticipate attacks based on your opponent's line of sight and avoid placements in certain areas.

Our interest lies in the setup phase of the game. The setup is a big contributor to the outcome of the game. A strong setup is an opportunity for an early lead; more effort is required for your opponent to level the playing field. We seek to approximate an optimal setup configuration. We simplify the problem by assuming each player prepares their setup in isolation.

In practice, trade-offs exist for setup configurations. Consider the scatter from your opponent's first attack. Intuitively we would like to minimize the balls lost from your side, whether that be from balls drifting away or being sunk into a pocket. But we should also consider difficulty in your own play. Defensive ball placements may block your line of sight, making it challenging to attack your opponent's side.

We frame approximating an optimal setup as an optimization problem. We lack a known objective function and must rely on black box methods. If we follow the logic that an early lead is the most important, we can potentially only consider the initial scatter. But if we wish to consider difficulty in our own play, we would like to consider our ability to attack. To build on the latter, we can consider playouts of the game.

Our optimization problem is made difficult by a continuous environment. We cannot feasibly ponder every possible action. Other considerations include the moderate dimensionality of the problem and potentially complex relationships between dimensions and evaluation methods that may prove computationally expensive. In a compromise to the above, we choose evolutionary algorithms as our optimization technique of interest. For fitness evaluation we take two approaches. A naive approach that emphasizes the initial scatter and an informed approach that considers playouts of the game.

3 Background

Our objective is to develop a framework for optimizing the initial state in a game with a continuous state space. The emphasis on the initial state is what separates this problem from the abundant literature on continuous optimization. We explore techniques used for continuous state space optimization, with particular interest in a black box objective function. Lastly, we consider approaches for bias or discretization in the evaluation.

3.1 Continuous optimization

Our goal is to find the global maxima of an objective function. We can evaluate our black box for arbitrary inputs, but it is computationally expensive. Our parameters exist in a high-dimensional space, and the relationship between them is complex. With a closed form we can calculate a gradient, but without knowing our objective function we must consider other approaches.

3.1.1 Bayesian optimization

Bayesian optimization is based on modeling our objective function with Bayesian statistics. We typically achieve this through the Gaussian process. The idea is that each sample of our objective function contributes to a multivariate distribution. We can use this distribution to predict the value of our objective function for an arbitrary input and do so with confidence bounds. The acquisition function determines where to sample next with a policy. A new sample increases the dimension of our distribution, and by repeating this process, we effectively refine our model [3].

Bayesian optimization benefits from avoiding costly evaluations of our objective function but struggles with the curse of dimensionality. Simplifying the model has demonstrated performance gain by relaxing unnecessarily assumed complexity. We do not always need to consider the relatedness of all features [5].

The acquisition function is of particular interest due to the exploration-exploitation dilemma. Our acquisition function must pick between sampling a well-explored or lesser-explored region of our input space. A common choice of acquisition function is expected improvement. For this method, we maximize the expected value of the difference between the greatest known evaluation and the predicted value of an input of interest. However, we may also consider high uncertainty for a more explorative approach [3].

3.1.2 Population-based optimization

Population-based approaches, or evolutionary algorithms, are inspired by nature's approach to optimization. While many strategies exist, the principle remains the same. We begin with a population of potential solutions. We select promising candidates with a selection policy. With our selection, we produce a new population through recombinations and mutations.

Traditionally, we use encoding to represent an individual's genotype. We encode a genotype to a bit-string and use decoding to derive their phenotype. However, representations with arbitrary variable types are now possible, rendering encoding unnecessary in many cases [2].

We are particularly interested in real numbers, since this is necessary for continuous optimization. To use real numbers, we need to define compatible operators for recombination and mutation.

Point mutations can be defined by sampling a uniform distribution with a range dependent on the current value and maximum or minimum mutation values. That is, a mutation can either increase or decrease the current value within a bound [12].

Crossovers can be defined as simply the exchange of a contiguous chunk of information between parents, but this is better suited for discrete problems. We can consider pairs of variables between related segments. In Blend Crossover (BLX), we sample a uniform distribution defined by a pair of parent values. However, we are at a disadvantage when variables are dependent. In BLXICA, we use statistical analysis against the population. We eliminate correlation with principal component analysis (PCA), followed by independent component analysis (ICA). We apply the transformations to our parents, perform BLX, then undo the transformations to find our final values. This approach has shown a promising improvement in the rate of convergence [11].

We often want a solution that satisfies a set of constraints. Our selection policy must consider the feasibility of a solution. The naive approach is to cull infeasible solutions, but the effectiveness varies from problem to problem. A popular alternative is to define a penalty function to penalize infeasible solutions. Static penalty functions are generation-agnostic; an infeasible solution in the first and last generation receives the same penalty. Alternatively, we can define a dynamic function. The dynamic penalty function relaxes the severity, which grows with the number of generations. However, we need to be careful; if the penalty grows too quickly, we may find ourselves stuck in a local maxima [10].

A concern for evolutionary algorithms is the need to frequently evaluate our objective function. Evaluations are necessary to assess the fitness of each individual of each generation [2]. If an evaluation of our objective function is particularly computationally expensive, solving complex problems becomes incredibly inefficient. We can consider addressing this problem with simplification. If we can make careful considerations to reduce the complexity of a problem, simplification has demonstrated significant performance gains [8].

3.2 Objective function evaluation

Without a closed form for our objective function, we will focus on the use of a simulation. In the case of a game, we can determine the value of our objective function with a utility function. To simulate a game, we can use a minimax technique such as alpha-beta pruning. When a large branching factor is at play, we should consider a cutoff; this requires us to define an evaluation function. This will almost certainly be necessary in a game with a continuous state space. However, it can become difficult to define a meaningful evaluation function for a complex game. To address this, Monte Carlo Tree Search (MCTS) uses playouts to estimate the value of a state. The algorithm performs iterations of selection, expansion, simulation, and backpropagation. Simulation involves a full playout of the game [4]. After several iterations, we may then sample the root of the tree for an estimate of our objective function.

Our selection policy returns to the exploration-exploitation dilemma. The most popular policy is Upper Confidence Bounds (UCB), which arises from multi-armed bandit problems. UCB is fundamental to the modern rollout-based approach to Monte-Carlo planning known as Upper Confidence Bounds Applied to Trees (UCT). UCT substantially outperforms stage-wise Monte-Carlo algorithms in the rate of convergence [7].

In a complex game environment, random actions may struggle to complete a full playout of the game. To account for this, we can consider a hybrid approach known as Monte-Carlo Tree Search Early Playout Termination (MCTS-EPT). This algorithm will terminate the playout after a set number of moves and use an evaluation function to decide the likely winner. But determining when to terminate is an important consideration. Early termination emphasizes the evaluation function, while delayed termination emphasizes MCTS. In light of this, a weak evaluation function is complemented by a prolonged playout. But even primitive evaluation functions for MCTS-EPT have still shown promising results against MCTS [9].

We can consider domain knowledge in our playouts. Playouts with informed moves are known as biased or knowledge-heavy. While it may appear advantageous to reduce randomness, this can potentially degrade the performance of MCTS. High variance aligns with a random policy, while low variance aligns with a strong bias. We need to be careful so as to maximize the benefit of our bias while mitigating or accepting reduced variance. Consider that there exists some underlying function that can describe the value of the current state. A biased policy may not accurately represent the underlying function, but so long as it aligns with the true maximum, we may benefit. However, an incorrect or overconfident bias out of alignment with the true maximum performs worse than a random policy [6].

3.3 Continuous state space bias and discretization

In a continuous state space, particularly with execution uncertainty, we look toward domain knowledge for reasonable actions. However, a discrete set of actions potentially overlooks advantageous actions outside of our domain knowledge. Domain knowledge can reduce variance similarly to the case of knowledge-heavy playouts. Kernel Regression UCT (KR-UCT) addresses this issue. First, we use the discrete selection of actions provided by domain knowledge as an initial pool for expansion in MCTS. After exhausting our initial actions, we can strike a balance between exploration and exploitation. We use kernel regression (KR) to generalize action value estimates and kernel density estimation (KDE) to assess lesser explored regions of our action space. We may select actions with a promising action value estimate using KR. We may select the most promising from lesser explored regions of our action space using KDE. This approach has not only demonstrated superior performance to UCT but also to other methods of interest, such as progressive widening [1].

4 Problem approaches

4.1 Representation

For our solution representation we consider a natural approach: a list of tuples that represent the x and y position of each ball. Since the environment is continuous, each value is a real number.

$$(x_1, y_1), (x_2, y_2), ..., (x_7, y_7)$$

We can collapse this representation into a list of 14 real numbers.

$$[x_1, y_1, x_2, y_2, ..., x_7, y_7]$$

With this representation we have defined a 14-dimensional continuous optimization problem. The following constraints apply for a solution to be considered feasible:

1. Any ball placed should be on the player's side
2. No two balls should overlap

4.2 Simulation

To assess the fitness of a solution or to evaluate our unknown objective function, we rely extensively on the use of a simulation. A simulation is necessary to account for complex physical interactions. We built our simulation using readily available libraries Pymunk and Pygame. Pygame is used exclusively for rendering the visuals, while Pymunk is used to calculate the physical interactions. Our simulation captures interactions between balls, pockets, and the table. We account for the force of friction and the elasticity of collisions.

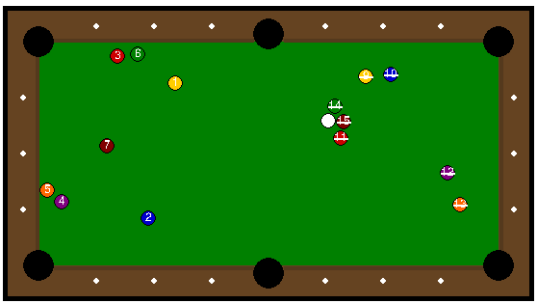


Figure 1: Simulation of pool environment.

Setting the state of the simulation requires a simulation state. A simulation state is an extension of the solution representation. Additional information for the simulation state includes the opponent's balls and whether a ball is in a pocket.

$$(\underline{origin}, \underline{direction}, power)$$

The first role of our simulation is to determine a new simulation state produced by an action. An action is represented by a tuple that includes an origin, a direction, and a power level. The origin determines the placement of the player's cue ball, while the direction and power determine the strength and direction of an impulse acting on the cue ball.

Algorithm 1 Candidate action scan

```
procedure ORIGIN-AND-DIRECTION(ball, player)
    x ← GET-POSITION(ball)
    θ ← 0
    while θ < 2π do
        v = (cos(θ), sin(θ))
        hit ← RAYCAST(x, v)
        p ← GET-POSITION(hit)
        if IN-BOUNDS(p, player) then
            return p, −v
        else
            p′, v′ ← REFLECT(p, v, player)
            if IN-BOUNDS(p′, player) then
                return p′, −v′
            end if
        end if
        θ ← θ + ϵ
    end while
    return 0, 0
end procedure
```

The second role of our simulation is to determine candidate actions with domain knowledge. Generally, you wish not to collide with balls on your own side but to collide with balls on your opponent's side. This is deduced from the winning condition of the game: you knock all the balls off your opponent's side. Consider a ball on your opponent's side; we would like to find an action that results in the cue ball colliding with this ball. We need to not only consider the direction of the attack but also the starting position inside the shooting bounds. If we considered a static position for the origin of attacks, this would dramatically change the dynamics of the game. If we always considered a straight-line attack, we would find ourselves sabotaging our own position. We need a more sophisticated approach.

In algorithm 1 we employ backwards reasoning to determine a candidate action for a ball on the opponent's side. For a given ball we emit rays from the ball starting from an angle of $\theta = 0$. For each ray, we check to see if it collides with an object within the shooting boundaries of the player. If this is the case, we found an action; we can use the location of the collision as the origin of the action and the negated ray to determine the direction of the action. Alternatively, if the collision did not occur within the player's shooting boundaries, we bounce the ray and perform the same check. If we still did not find a candidate action, we increment θ and try again.

4.3 Evolutionary algorithm

Returning to our solution representation, consider each value to be a "gene." For a stronger definition we provide a "gene space," a domain for each gene in our solution representation. We use the gene space to generate an initial population of random solutions. Note that our first constraint, for each ball to be on the player's side, is initially satisfied by virtue of the gene space. The domain of each gene is determined by the extents of the player's side.

$$\begin{bmatrix} x_{0,min} & x_{1,max} \\ y_{0,min} & y_{1,max} \\ \dots & \dots \\ x_{7,min} & x_{7,max} \\ y_{7,min} & y_{7,max} \end{bmatrix}$$

To "evolve" our population of solutions we must define a selection function, crossover function, mutation function, and fitness function.

4.3.1 Selection function

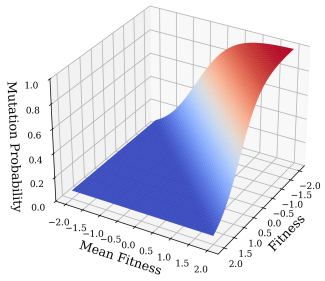
We elected for an unaltered steady-state selection. Steady-state selection pairs parents for "mating" by solution fitness. We select the two most fit solutions as the first parents, and from the remaining pool, we select the next two most fit parents, etc. We repeat the selection process until no solutions remain.

4.3.2 Crossover function

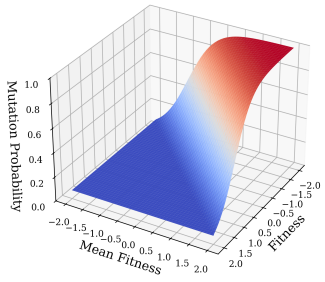
To produce our preliminary pool of offspring, we use a single-point crossover. We select a random gene as the crossover point in our solution representation. Our parents exchange segments about the crossover point to produce offspring.

4.3.3 Dynamic behavior

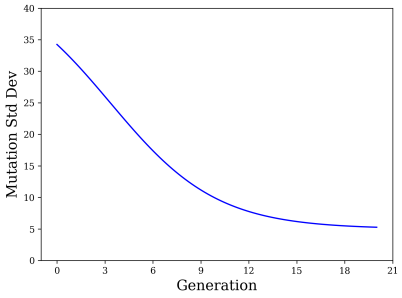
To avoid becoming stuck in local optima and to effectively address the exploration-exploitation dilemma, characteristics of our mutation function and fitness functions exhibit dynamic behavior. We define functions to model adaptive mutation probability, dynamic mutation standard deviation, and a dynamic constraint violation penalty. Note that our mutation function uses a standard deviation since the function samples a truncated normal distribution; see section 4.3.4 for details.



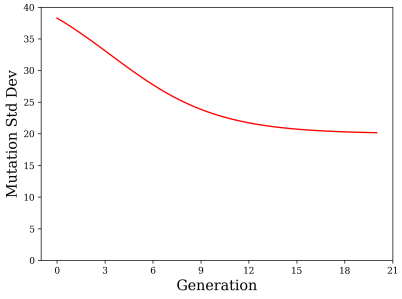
(a) Naive approach mutation probability by solution fitness and mean population fitness.



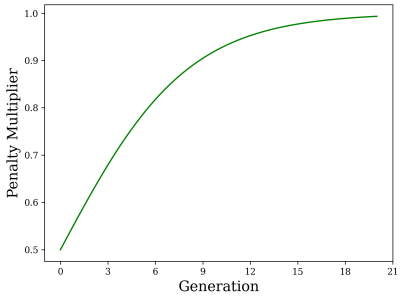
(b) Informed approach mutation probability by solution fitness and mean population fitness.



(c) Naive approach mutation standard deviation by generation.



(d) Informed approach mutation standard deviation by generation.



(e) Common penalty multiplier by generation.

Figure 2: Dynamic mutation behavior and constraint penalties.

To fine-tune desired behavior of our mutation function and constraint penalties, we make extensive use of sigmoid functions. Sigmoid functions provide flexible, smooth value transitions.

$$f(t) = \alpha \frac{e^{\epsilon t+\gamma}}{1+e^{\epsilon t+\gamma}} + \beta \tag{1}$$

Function 1 models a dynamic mutation standard deviation. A small mutation standard deviation aligns with exploitation, while a large mutation standard deviation aligns with exploration. As $\sigma \to \infty$ our truncated normal distribution converges toward a uniform distribution; effectively, each point in our gene space has the same probability. As $\sigma \to 0$ our truncated normal distribution converges toward a degenerate distribution about the mean.

t is the current generation, α is the scaling factor, β is the minimum standard deviation, ϵ is the steepness, and γ is the offset.

In figure 2c: $\alpha = 40$, $\beta = 5$, $\epsilon = -0.3$, and $\gamma = 1$. In figure 2d we adjust the scale and minimum: $\alpha = 25$ and $\beta = 20$. In practice, increasing the minimum counteracts premature convergence.

$$f(\phi, \phi_\mu) \begin{cases} \frac{e^{\epsilon|\phi-\phi_\mu|+\gamma}}{1+e^{\epsilon|\phi-\phi_\mu|+\gamma}} & \phi < \phi_\mu \\ \beta & \phi \geq \phi_\mu \end{cases} \tag{2}$$

Function 2 models an adaptive mutation probability. We favored an adaptive approach for fine control of exploration and exploitation. Underperforming solutions have a dynamic mutation probability. Overperforming solutions have a fixed mutation probability. A high mutation probability emphasizes exploration, while a low mutation probability emphasizes exploitation. We exploit promising solutions and offer poor solutions as candidates for exploration. A very poor solution is unlikely to possess desirable traits; therefore, we can be more disruptive than a moderately

poor solution, which may possess a desirable trait.

ϕ is a solution's fitness, ϕ_μ is the population's mean fitness, β is the "probability default," ϵ is the steepness, and γ is the offset.

In figure 2a: $\epsilon = 2$, $\gamma = -2.5$, and $\beta = \frac{1}{14}$. In figure 2b we adjust the steepness: $\epsilon = 2.5$. Increasing the steepness effectively increases the likelihood of a mutation for a smaller difference in fitness. In practice, a lower threshold counteracts premature convergence.

$$f(t) = \frac{e^{\epsilon t+\gamma}}{1 + e^{\epsilon t+\gamma}} \tag{3}$$

Function 3 models a dynamic constraint violation penalty multiplier. We desire severe penalties to ensure our final solution satisfies our constraints; however, harsh penalties may hinder exploration for early generations. Again, since we are most concerned with our constraints being satisfied by our final solution, we may relax our constraints for early generations. We do not ignore constraints for early generations. If we ignored our constraints, we would risk elitism persevering an infeasible solution.

t is the current generation, ϵ is the steepness, and γ is the offset.

In figure 2e, $\epsilon = 0.25$ and $\gamma = 0$. The parameters chosen are common to both problem approaches.

4.3.4 Mutation function

Algorithm 2 Offspring mutation

procedure MUTATE($offspring$, t, $genespace$)
  $\phi_\mu \leftarrow$ AVERAGE_FITNESS($offspring$)
  for $o \in offspring$ **do**
    $genes \leftarrow$ GENES(o)
    $\phi \leftarrow$ FITNESS(o)
    $i \leftarrow 0$
    while $i < |genes|$ **do**
      $p \leftarrow \begin{cases} \frac{e^{p_\epsilon|\phi-\phi_\mu|+p_\gamma}}{1+e^{p_\epsilon|\phi-\phi_\mu|+p_\gamma}} & \phi < \phi_\mu \\ p_\beta & \phi \geq \phi_\mu \end{cases}$
      if RANDOM(0,1) $\leq p$ **then**
        $\sigma \leftarrow \sigma_\alpha \frac{e^{\sigma_\epsilon t+\sigma_\gamma}}{1+e^{\sigma_\epsilon t+\sigma_\gamma}} + \sigma_\beta$
        $\mu \leftarrow genes[i]$
        $x_{min}, x_{max} = genespace[i]$
        $a_{trunc} \leftarrow (x_{min} - \mu)/\sigma$
        $b_{trunc} \leftarrow (x_{max} - \mu)/\sigma$
        $dist \leftarrow$ DIST(a_{trunc},b_{trunc}, μ, σ)
        $genes[i] \leftarrow$ RVS($dist$)
      end if
      $i \leftarrow i + 1$
    end while
  end for
end procedure

p_γ is the probability offset, p_ϵ is the probability steepness, and p_β is the probability default. σ_γ is the standard deviation offset, σ_ϵ is the standard deviation steepness, σ_β is the standard deviation minimum, and σ_α is the standard deviation scale.

Algorithm 2 is our mutation function. For a given offspring we determine our mutation probability using equation 2. With our mutation probability known, we determine on a gene-by-gene basis whether to mutate. If applicable, we apply the BLX technique by sampling a truncated normal distribution to determine the new value of the gene. The mean of our distribution is determined by the current value of the gene. The standard deviation of our distribution is determined by sampling equation 1. We determine the boundaries of our distribution using a domain specified by the gene space. These boundaries ensure we do not violate our "out of bounds" constraint; however, balls may overlap as a consequence of a mutation still.

4.3.5 Naive fitness function

Our naive fitness function emphasizes the initial scatter as critical for determining the fitness of a solution. This approach assumes a strong relationship exists between the initial scatter and winning or losing.

Algorithm 3 Naive fitness evaluation

$player \leftarrow -1$
procedure NAIVE-FITNESS($solution$)
  $penalty \leftarrow$ OVERLAP_COUNT($solution$)
  $multiplier \leftarrow \frac{e^{p_\epsilon t+p_\gamma}}{1+e^{p_\epsilon t+p_\gamma}}$
  $num\_actions \leftarrow 0$
  $score \leftarrow 0$
  $i \leftarrow 0$
  while $i < \lambda$ **do**
    $state \leftarrow$ INIT-STATE($solution$, $player$)
    $actions \leftarrow$ ACTIONS($state$, $player$)
    for $a \in actions$ **do**
      $state' \leftarrow$ MOVE($state$, $player$, a)
      $score \leftarrow score+$EVAL($state'$)
      $num\_actions \leftarrow num\_actions + 1$
    end for
    $i \leftarrow i + 1$
  end while
  $avg \leftarrow score \div num\_actions$
  return $avg - (penalty * multiplier)$
end procedure

p_γ is the penalty multiplier offset and p_ϵ is the penalty multiplier steepness. λ is the number of random opponent samples.

Algorithm 3 is our naive fitness function. For a given solution we evaluate fitness by considering all states that result from the opponent's first action. That is, we consider all initial scatters. These actions are determined by algorithm 1. In each state we calculate a score by totaling the number of balls on the player's side and subtracting the number of balls on the opponent's side. Note that

balls in neutral territory do not directly contribute. We repeat this process for a set number of opponent samples. Each opponent sample pairs the solution against a random opponent setup configuration. We calculate the average score and subtract a constraint violation penalty if any balls overlap. We apply a multiplier to our penalty using equation 3. Because the opponent samples are random, the fitness function is noisy. The fitness of a solution may vary between evaluations.

4.3.6 Informed fitness function

Our informed fitness function does not place any emphasis. We still consider that the initial scatter is of significance but also that perhaps our state evaluation cannot capture all details. That is to say, we need to consider the events that unfold over the course of playing the game.

Algorithm 4 Informed fitness evaluation

```
player ← −1
procedure INFORMED-FITNESS(solution)
    penalty ← OVERLAP_COUNT(solution)
    multiplier ← e^{p_ε t + p_γ} / (1 + e^{p_ε t + p_γ})
    score ← 0
    i ← 0
    while i < λ do
        state ← INIT-STATE(solution, player)
        tree ← MCTS(state,ρ)
        score ← score+ROOT-VAL(tree)
        i ← i + 1
    end while
    avg ← score ÷ (λ ∗ ρ)
    return avg − (penalty ∗ multiplier)
end procedure
```

p_γ is the penalty multiplier offset and p_ϵ is the penalty multiplier steepness. λ is the number of random opponent samples and ρ is the number of Monte Carlo iterations per sample.

Algorithm 4 is our informed fitness function. For a given solution we evaluate fitness by considering the root node's averaged value after several iterations of MCTS-EPT. The MCTS implementation back-propagates the state evaluation upon reaching a terminal condition. A terminal condition encompasses a terminal state or a depth limit. We use the same method for state evaluation as algorithm 3. Likewise, we use algorithm 1 for knowledge-heavy payouts, we restart MCTS for a set number of random opponent samples, and we apply a multiplier to our penalty using equation 3. The informed approach is noisy as well due to the random opponent samples.

We elected to use early playout termination with a depth limit of 4 since Battle Pool is a complex game. Physical interactions in a continuous state space do not inspire confidence. With knowledge-heavy playouts, we lack stochasticity and risk reaching a cyclical position in the game. We must also weigh the computational complexity of evaluation.

4.3.7 Elitism

After crossovers and mutations take place, our offspring replace the previous population. However, to avoid potentially losing high-fitness solutions, we employ elitism with n elites. The n highest fitness individuals from the previous generation are carried over to the next untouched. To maintain a constant population size, we eliminate the n lowest fitness individuals of the new generation.

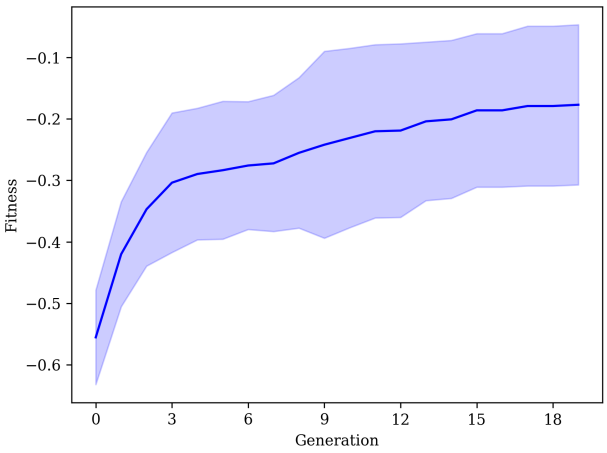
5 Results

To evaluate performance, we performed 10 iterations of the naive approach and the informed approach, respectively. Each iteration spanned 20 generations with a population of 80 solutions. We created separate instances of our evolutionary algorithm for both approaches. While the primary difference between these instances is the fitness function, we performed fine-tuning to mitigate premature convergence for both approaches. Our fine-tuning is reflected in the parameter choice for our dynamic functions; see figure 2 for details. Another distinction is that the naive approach uses three elites while the informed approach uses a single elite. In early trials the naive approach struggled with exploitation. Increasing the number of elites seeks to improve exploitation by keeping a larger record of potential local optima.

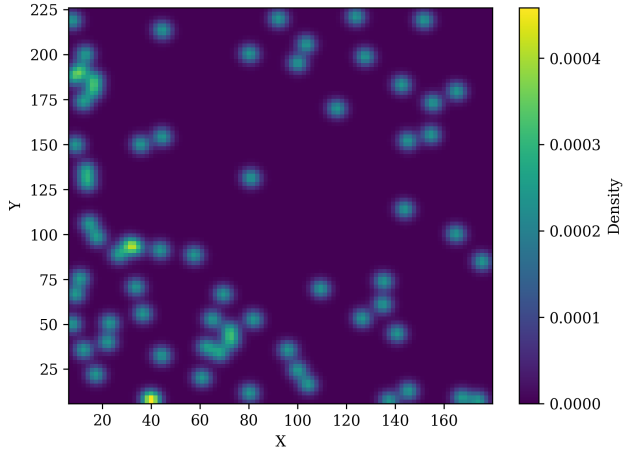
For both approaches we measured the rate of convergence by recording the fitness of the best solution by generation. In addition, we kept a record of ball positions of the best solution by generation, offering insight into behavior at a high level.

Because the fitness evaluation is tightly coupled to the approach, we need to carefully consider our means of comparison. Because the informed approach does not emphasize a characteristic of the game, unlike our naive approach, algorithm 4 is a strong candidate. In principle, dedicating more resources to an informed approach yields a closer approximation of the underlying objective function.

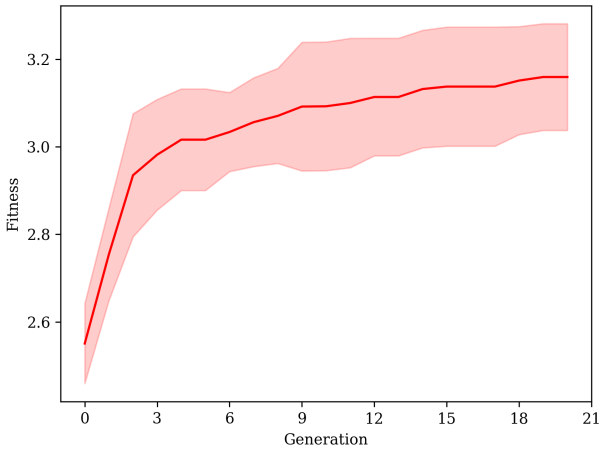
While we were concerned for performance when "evolving" our population, our concern was rooted in the fact that we needed to evaluate the fitness of each solution for each generation. Without this concern being relevant, we may relax restrictions on our search to improve accuracy. We raised the depth limit from 4 to 10 and increased the number of iterations from 20 to 50.



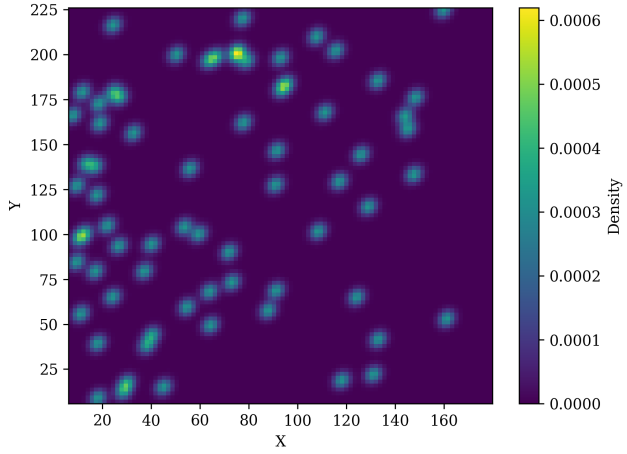
(a) Naive approach best solution fitness by generation.



(c) Kernel density estimation of naive solutions.



(b) Informed approach best solution fitness by generation.



(d) Kernel density estimation of informed solutions.

Figure 3: Fitness convergence and solution density estimation for 10 iterations of the naive approach and informed approach. 20 generations comprise a single iteration.

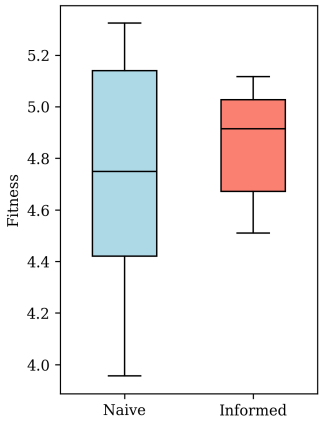


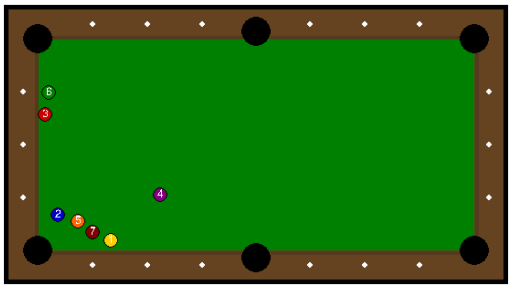
Figure 4: Common fitness evaluation.

6 Analysis

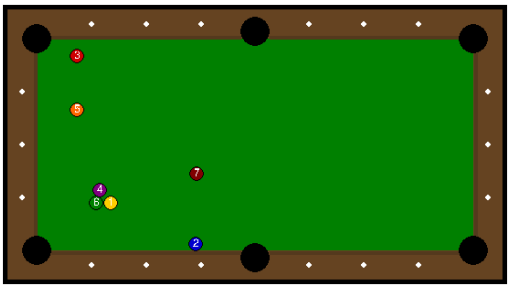
Figures 3a and 3b inform us that the naive approach and informed approach converge on a local optimum. The convergence behavior of both approaches is similar. The informed approach potentially demonstrates a stronger early improvement. Interestingly, the standard deviation for each generation appears to be proportionally smaller for the informed approach. The large variance present for the naive approach suggests that the noisy nature of the naive fitness function hinders exploitation. We emphasize that when comparing these figures, we should not judge fitness values.

Figure 4 provides insight into comparing the solutions of the naive approach and the informed approach. The median of the informed approach, 4.915, is greater than the median of the naive approach, 4.749, suggesting superior performance. We also note that the IQR of the informed approach, 0.402, is smaller than the IQR of the naive approach, 0.782, demonstrating superior consistency.

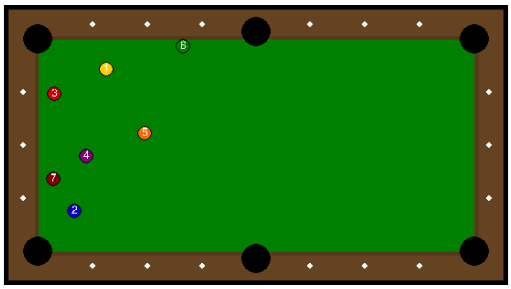
Figures 3c and 3d illustrate high-level trends of the naive approach's solutions and the informed approach's solutions. We note solutions to both approaches favor the closest boundaries of the player's side. However, the naive approach exhibits a stronger affinity for the closest boundaries than the informed approach. In fact, the informed approach appears to demonstrate only a slight bias toward the closest boundaries. We gather from the diversity of our figures that the fitness landscape is likely rugged, with many local optima.



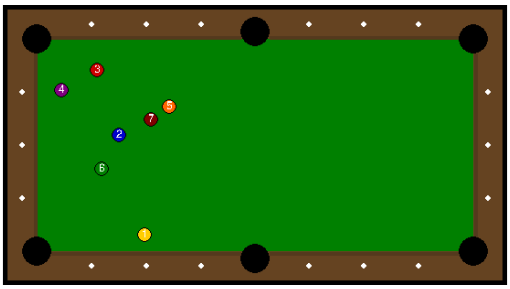
(a) Solution from the naive approach.



(c) Solution from the informed approach.



(b) Solution from the naive approach.



(d) Solution from the informed approach.

Figure 5: Noteworthy solutions from the naive approach and informed approach.

While we can only speculate as to why a particular solution is successful, we can apply our insight as players of the game.

At first glance, the setup in figure 5a may appear haphazardly thrown together. Clustering balls together is counterintuitive. However, note the arc these balls take, the small gaps between them, and the placement in the corner. The arc technique offers a degree of predictability in the transfer of momentum.

The setup in figure 5c presents captivating asymmetry. The tightly coupled group of three perhaps favors a controlled loss of energy. The two pairs of balls spaced apart may seek to minimize collateral damage when collided with. Building on the latter, figures 5b and 5d potentially seek to spread balls apart for a similar reason.

7 Conclusions and future work

We demonstrate an encouraging approach to optimizing the initial state of a game with a continuous state space. Our evolutionary algorithm exhibited convergence toward feasible solutions that offer an advantage over an opponent with a random setup. We considered a naive fitness function that emphasized the initial scatter and an informed fitness function that considers playouts of the game using MCTS-EPT. We found the informed approach to outperform our naive approach both in consistency and accuracy.

Evidently our fitness landscape is rugged. We applied techniques to mitigate premature convergence such as a dynamic mutation standard deviation, an adaptive mutation probability, and a dynamic constraint violation penalty multiplier. Despite improvement, becoming stuck in local optima appears to still be a problem.

Exploring Bayesian optimization with a simplified model is a worthwhile consideration. The frequent and expensive evaluation of our unknown objective function bottlenecks the accuracy of our fitness functions and likely contributes noise.

We also should consider exploring alternative crossover techniques. Relationships exist between the dimensions of our solution representation. Perhaps adopting BLXICA would allow us to mitigate constraint violations and improve exploitation.

Our knowledge-heavy approach hinders exploration; if we provide a greater variance in actions, we may improve the accuracy of our fitness evaluations. We should consider adopting Kernel Regression UCT.

A common approach in human solutions is symmetry. We could potentially experiment with mutations that increase symmetry, leveraging our domain knowledge. Similarly, experimenting with a hand-selected starting population of promising human solutions may yield interesting results.

References

[1] Bianchi, F., Bonanni, L., Castellini, A., and Farinelli, A. Monte carlo tree search planning for continuous action and state spaces, 11 2023.

[2] Bäck, T. H. W., Kononova, A. V., van Stein, B., Wang, H., Antonov, K. A., Kalkreuth, R. T., de Nobel, J., Vermetten, D., de Winter, R., and Ye, F. Evolutionary Algorithms for Parameter Optimization—Thirty Years Later. *Evolutionary Computation 31*, 2 (June 2023), 81–122.

[3] Frazier, P. I. A Tutorial on Bayesian Optimization, July 2018. arXiv:1807.02811.

[4] Fu, M. C. Monte carlo tree search: A tutorial. In *2018 Winter Simulation Conference (WSC)* (2018), pp. 222–236.

[5] Hvarfner, C., Hellsten, E. O., and Nardi, L. Vanilla Bayesian Optimization Performs Great in High Dimensions, June 2024. arXiv:2402.02229.

[6] James, S., Rosman, B., and Konidaris, G. An investigation into the effectiveness of heavy rollouts in uct. In *General Intelligence in Game-Playing Agents Workshop at IJCAI* (2016), pp. 55–61.

[7] Kocsis, L., and Szepesvári, C. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 282–293. Series Title: Lecture Notes in Computer Science.

[8] Li, J.-Y., Zhan, Z.-H., and Zhang, J. Evolutionary Computation for Expensive Optimization: A Survey. *Machine Intelligence Research 19*, 1 (2022), 3–23.

[9] Lorentz, R. Using evaluation functions in Monte-Carlo Tree Search. *Theoretical Computer Science 644* (Sept. 2016), 106–113.

[10] Michalewicz, Z., and Schoenauer, M. Evolutionary Algorithms for Constrained Parameter Optimization Problems. *Evolutionary Computation 4*, 1 (Mar. 1996), 1–32.

[11] Takahashi, M., and Kita, H. A crossover operator using independent component analysis for real-coded genetic algorithms. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)* (Seoul, South Korea, 2001), vol. 1, IEEE, pp. 643–649.

[12] Wright, A. Genetic algorithms for real parameter optimization. *Foundations of Genetic Algorithms 1* (06 1999).