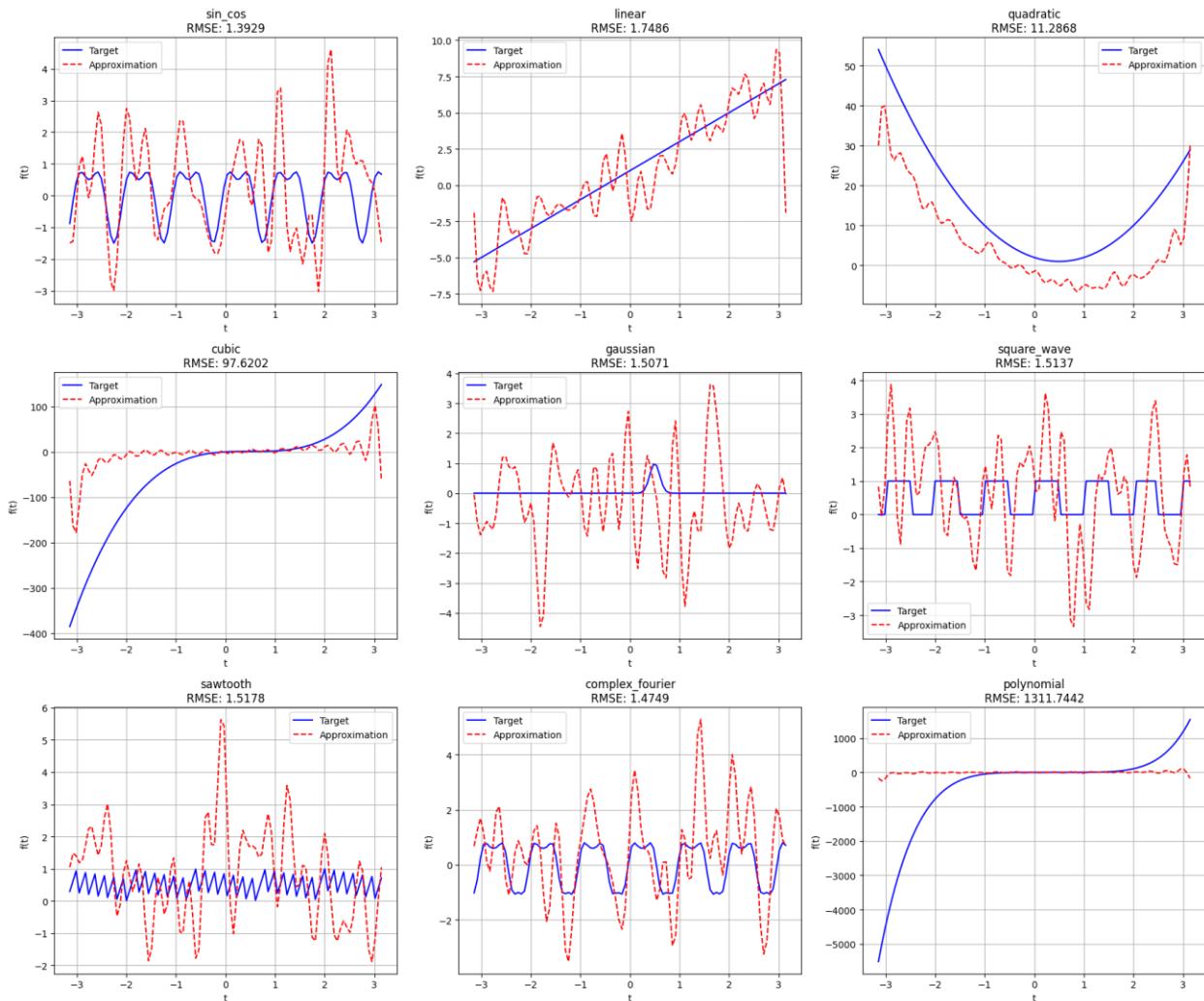
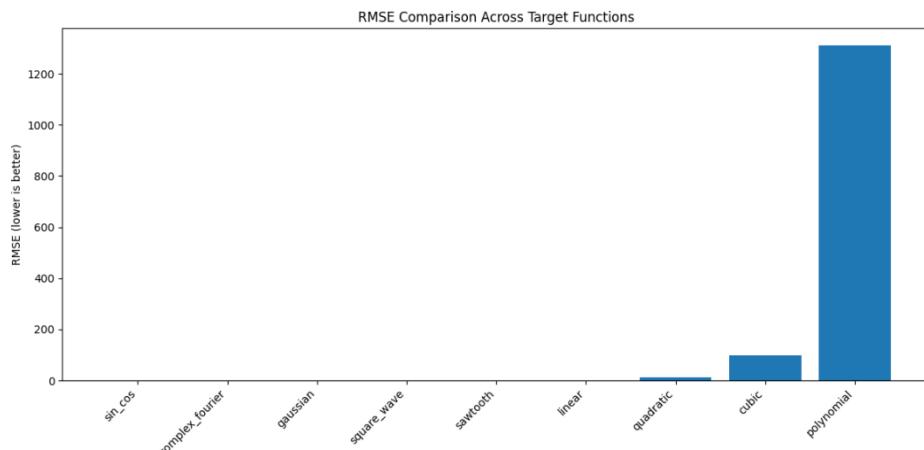


Part One: Genetic Algorithm

First I tried to use the RMSE fitness function and the result was like below:

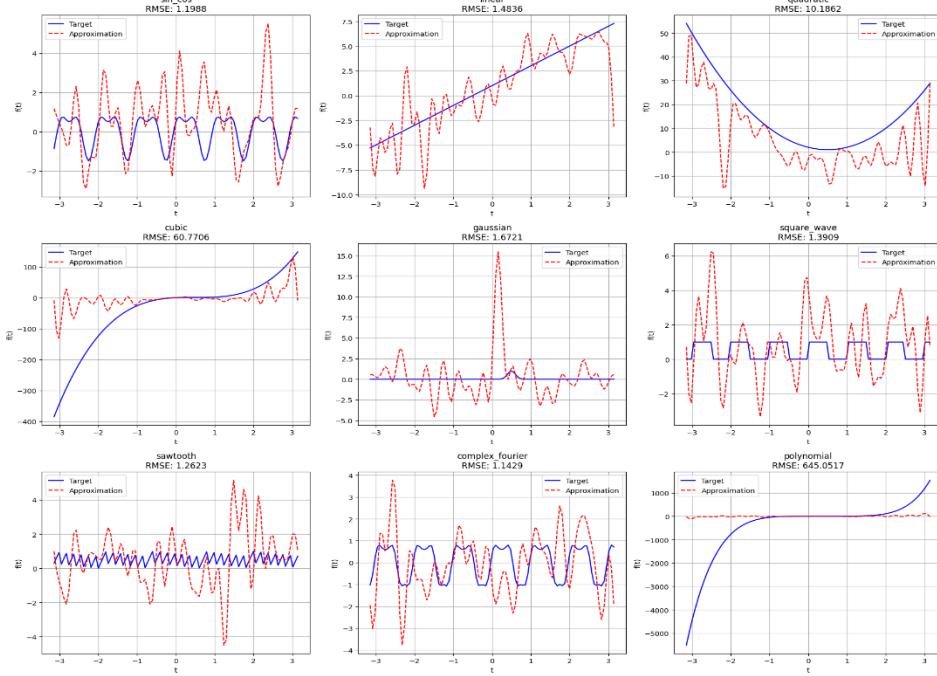


The results have a lot of noise and they are not good enough:

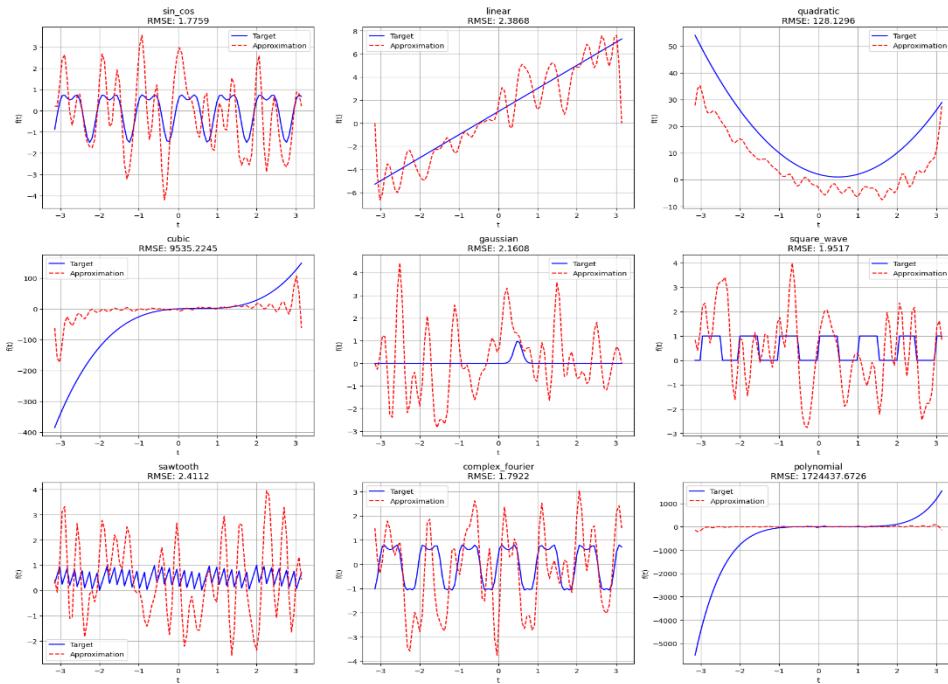


So maybe changing the Fitness metric can evaluate results. RMSE was **Root Mean Square Error** means The square root of the average of squared differences. Penalizes larger errors more heavily than smaller ones.

MAE (Mean Absolute Error): The average of absolute differences. Treats all errors equally regardless of magnitude:



MSE (Mean Squared Error): The average of squared differences. Similar to RMSE but without the square root:



Despite examining three different Fitness Metrics, the results are still noisy. We obtained these results with the following parameters:

```
# Algorithm parameters
NUM_COEFFS = 41 # a0, a1 to a20, b1 to b20
POPULATION_SIZE = 100
GENERATIONS = 200
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.9
SELECTION_RATE = 0.5
FUNCTION_RANGE = (-np.pi, np.pi)
SAMPLE_COUNT = 100
A = 3 # Domain of coefficients [-A, A]
```

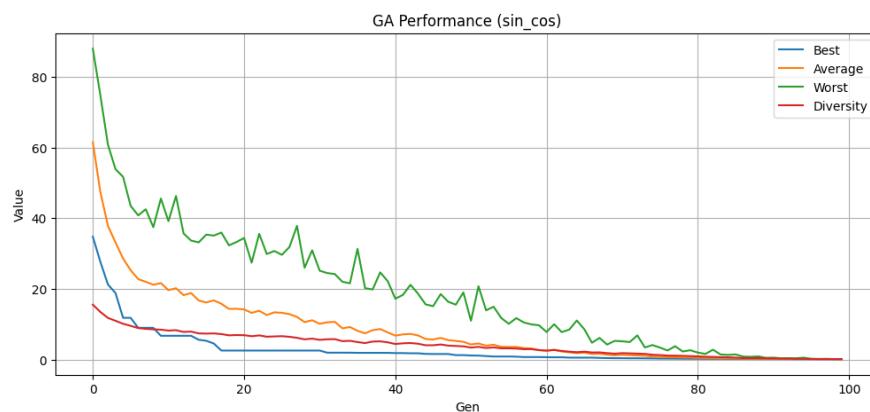
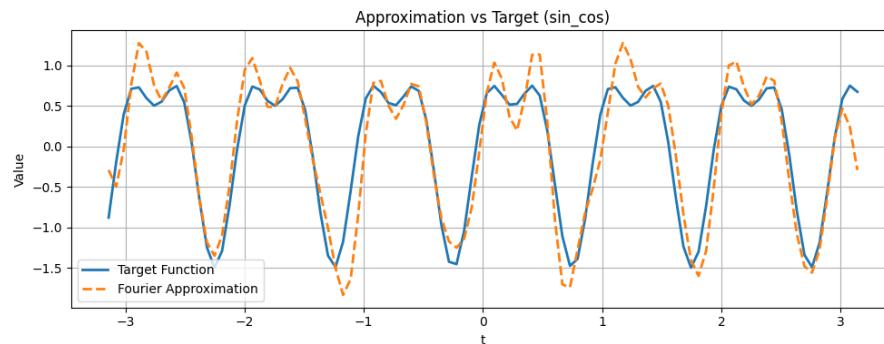
According to mathematic logics, The best A is MAX – MIN of the target function. For example for the sin_cos function A=3 is a good choice.

I tried different strategies for selection, crossover, mutation, and elitism to find a better approximation. I used these parameters:

```
# Algorithm parameters
NUM_COEFFS = 41 # a0, a1 to a20, b1 to b20
POPULATION_SIZE = 100
GENERATIONS = 100
MUTATION_RATE = 0.15
CROSSOVER_RATE = 0.9
SELECTION_RATE = 0.5
FUNCTION_RANGE = (-np.pi, np.pi)
SAMPLE_COUNT = 100
A = 3 # Domain of coefficients [-A, A]
ELITISM RATE = 0.14 # keep top 2%
```

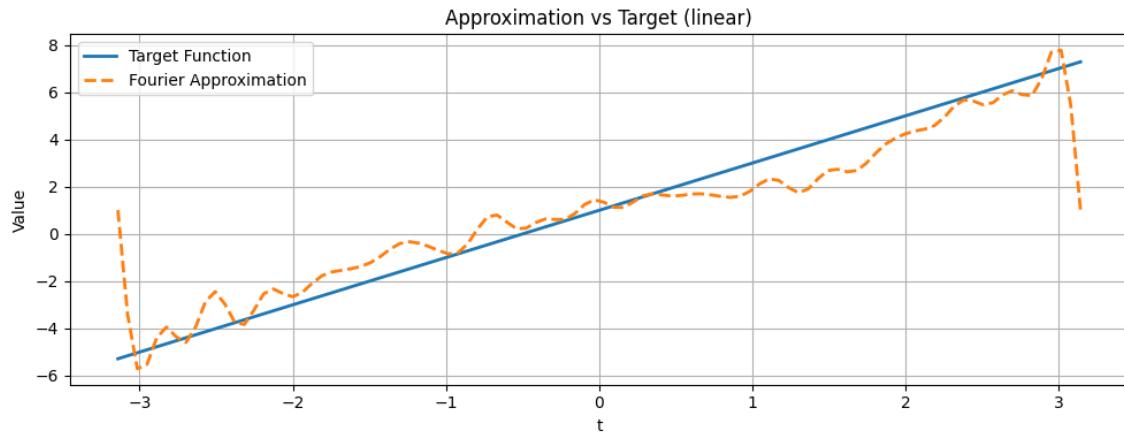
With these strategies:

The result becomes enhanced and more accurate:

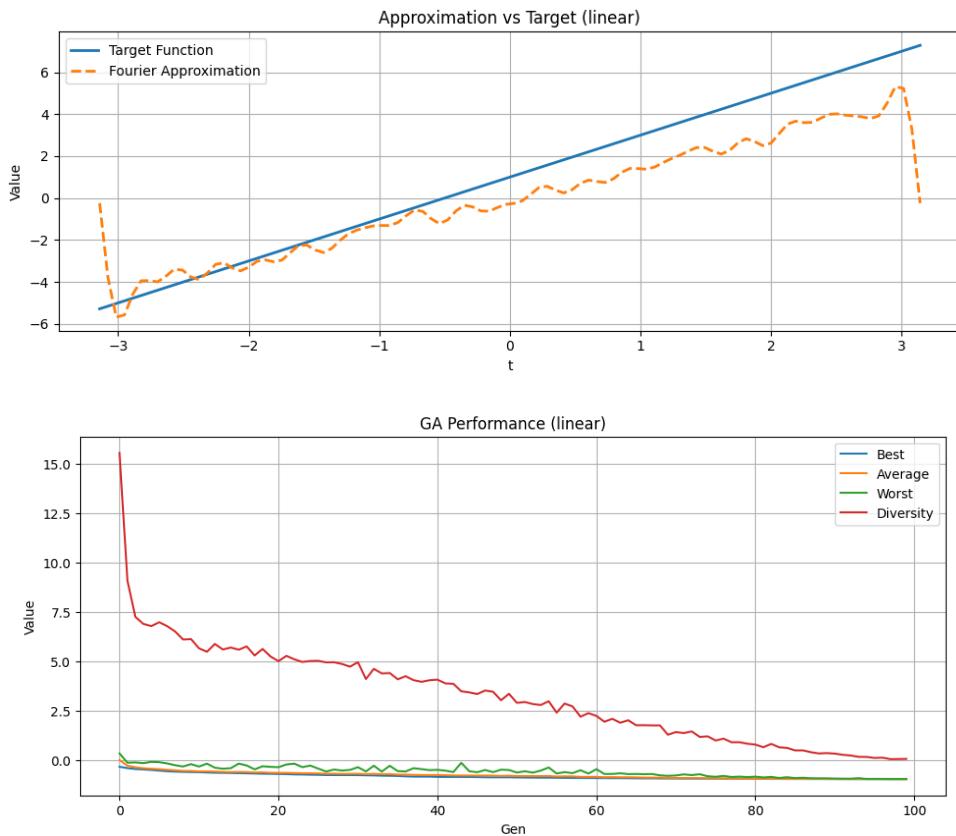


This shows that we have reached a relatively good result for the sin_cos function in 100 generations. Obviously, if we increase the number of generations, the result will be better. Now we test these set for the other functions:

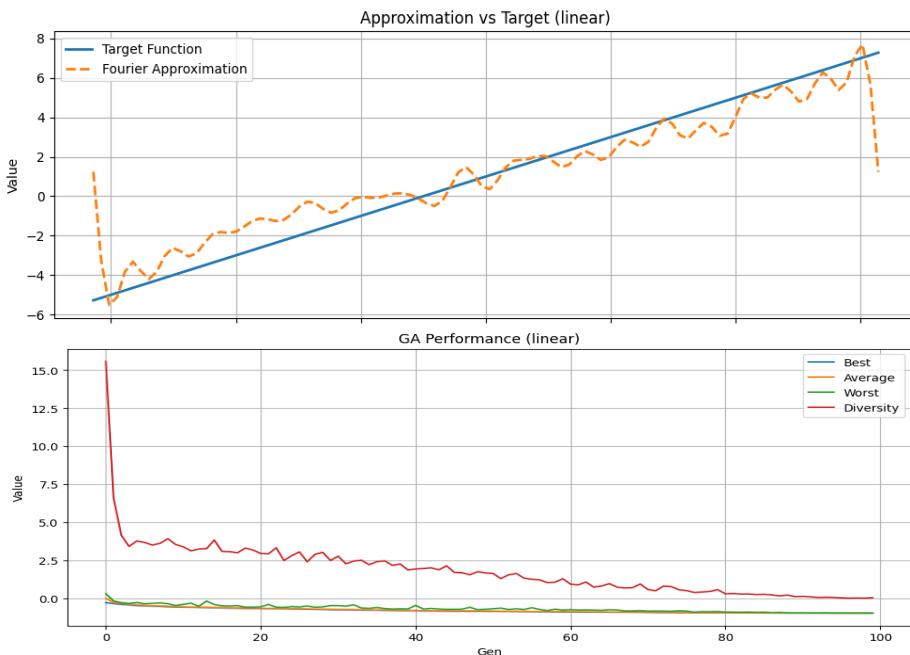
Linear Function:



It is not good enough. So we use another way. `fitness_metric="normalized_cross_correlation"` and `selection_method="roulette"` and `mutation_rate = 0.15`:



Then we change the mutation rate to 0.05:



Before we examine the graph of the quadratic function, we need to look for a way to reduce noise in the estimated graphs. Parameters that affect noise reduction:

1. Mutation Rate

Excessively high mutation rate causes noise because many changes are applied randomly to chromosomes.

Suggestion: Reduce mutation rate (mutation_rate). For example, from 0.2 → 0.05 or even 0.01.

2. Mutation Strategy

For example, Gaussian mutation with a high standard deviation introduces a lot of noise.

Suggestion: If we use Gaussian, we must reduce the standard deviation (std). Or choose a simpler mutation strategy such as uniform or random_reset.

3. Number of Generations

A low number of generations will cause the algorithm to not converge to a good answer

Suggestion: Increase the number of generations. For example, 100 → 300.

4. Population Size

A small population cannot generate enough genetic diversity for accurate learning.

Suggestion: Increase pop_size. For example, from 20 → 50 or 100.

5. Number of Fourier Terms

Overfitting may occur if you have too many sine/cosine terms.

Suggestion: Limit the number of Fourier terms (harmonics or N), for example, 5 to 10.

6. Regularization

We can add a term to the fitness function that penalizes larger weights (L2 regularization) to make the curve smoother.

Suggestion: We can modify the fitness function to take this into account.

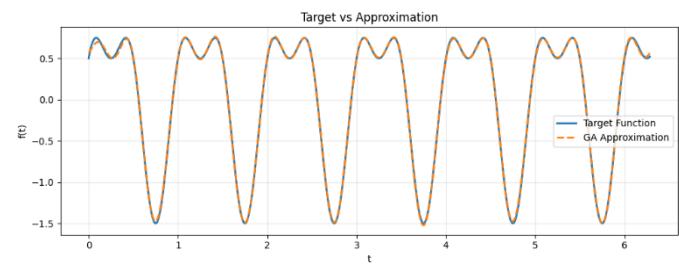
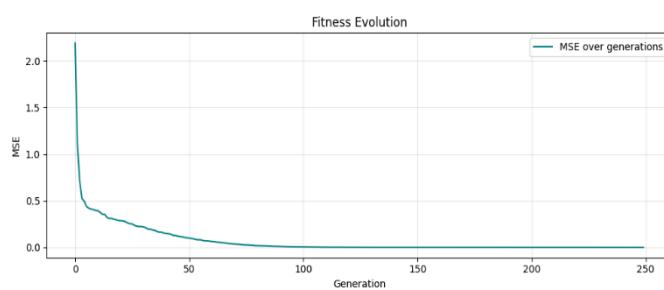
In this way we can reach the approximation with minimum noise:

the new class GENETIC FOURIER is good enough for periodic target functions but awful for estimating other functions.

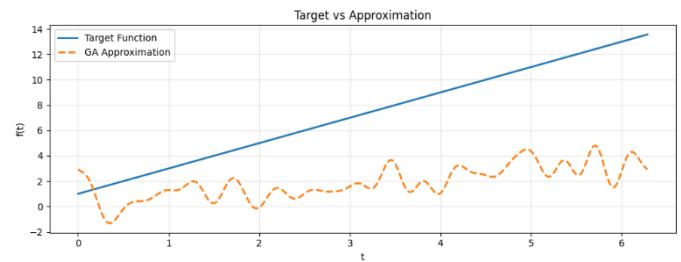
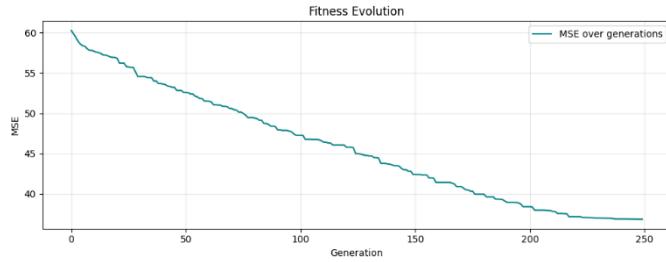
I set the parameters like this:

```
NUM_COEFFS = 41 # a0 + 20 a_n + 20 b_n
MUTATION_RATE = 0.05
ELITISM_COUNT = 5
```

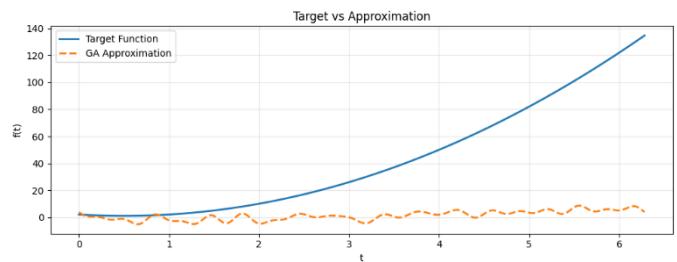
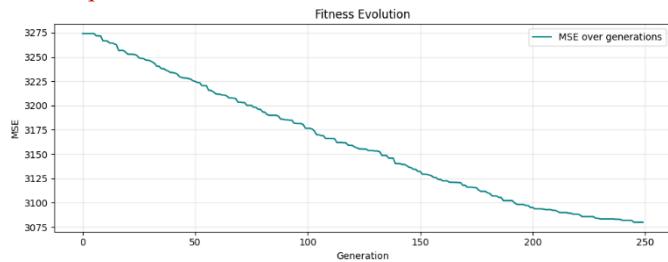
sin_cos:



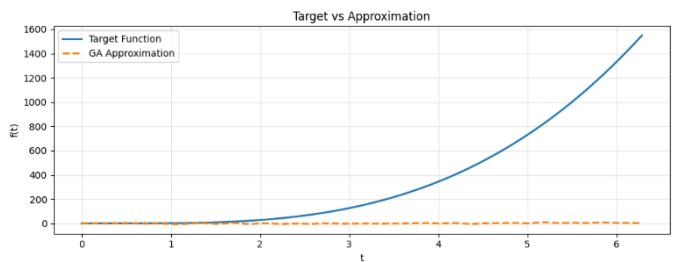
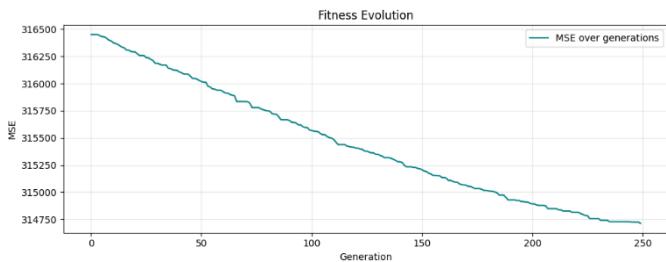
linear:



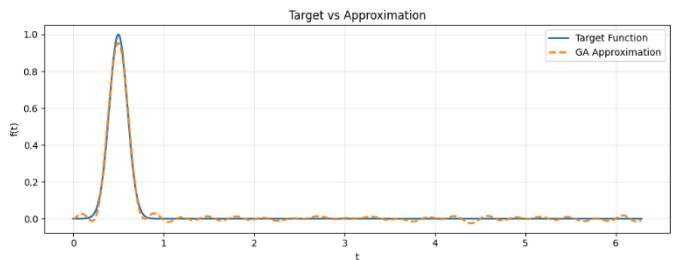
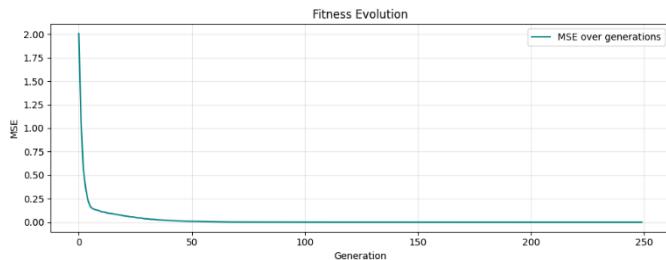
quadratic:



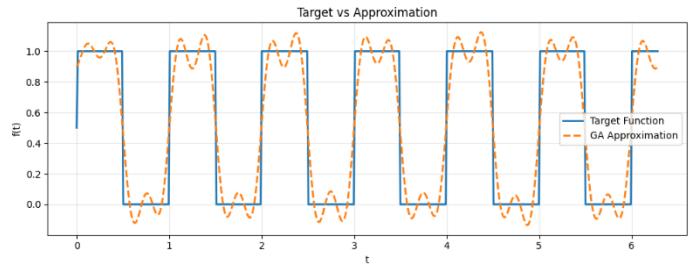
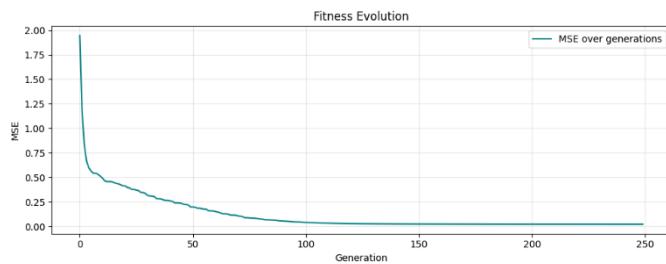
cubic:



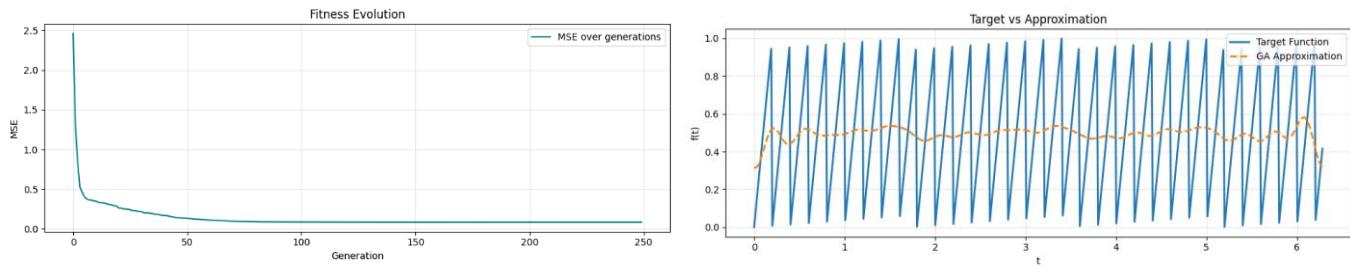
Gaussian:



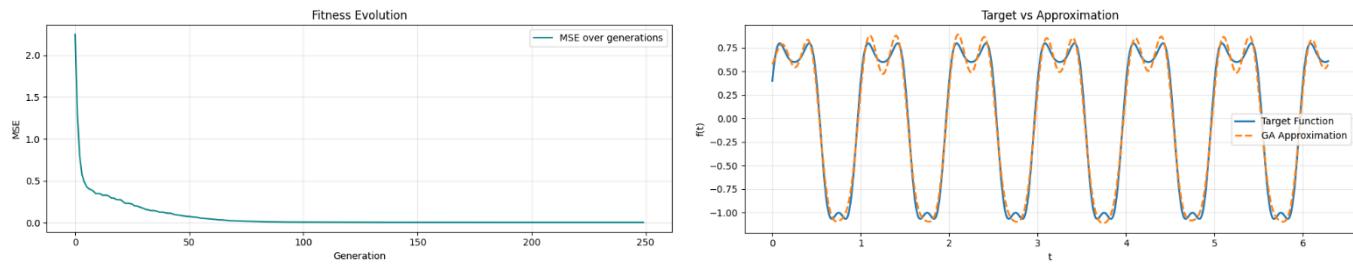
Square Wave:



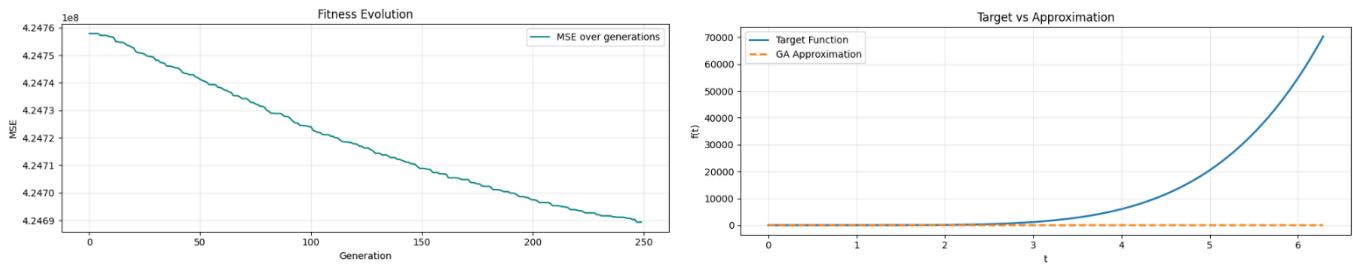
Sawtooth:



Complex Fourier:



Polynomial:



The **Fourier series** is fundamentally designed for approximating **periodic** functions. But we had a relatively good approximation in our first GA class for them. So, we try to make an accurate approximation for non-periodic functions.

Some ways to make our GA fourier estimator work for non-periodic functions:

1. **Use a Different Basis:** Replace the Fourier basis with a basis that's better suited for non-periodic functions (such as Legendre polynomials, Chebyshev polynomials, Wavelets, and so on).
2. **Windowing the Input Function (Tapering):** Multiply the input function by a **window function** (like Hann or Hamming) to gradually reduce the function to zero at the edges of the interval, making it "look" periodic.
3. **Mirror Extension:** Extend the function outside the interval $[0, 2\pi]$ by mirroring it. This can help make the function continuous at the boundaries.

There is a trade-off with the observations. If we want to achieve a relatively good result for most functions, we use the first model. If we want to achieve an excellent result for intermittent functions but for other functions the result is negligible, the second model is a unique choice.

گزارش فارسی برخی از مفاهیم به کاربرده شده و ابهام های موجود در پروژه:

توضیح یک: تعریف مفاهیم

در این پروژه هدف ما تقریب تابع هدف با استفاده از سری فوریه و الگوریتم ژنتیک است. در اینجا منظور از ژن یک عدد حقیقی است که نمایانگر یکی از ضرایب سری فوریه است. بنابراین ژن ها شامل مقادیر بی ان و آن و آ صفر هستند. هر کروموزوم در این پروژه یک آرایه ۴۱ عضوی از اعداد حقیقی است که نمایانگر یک سری فوریه خاص است. این سری فوریه تلاش می کند تابع هدف را تقریب بزند. بنابراین هر کروموزوم یک پاسخ احتمالی برای تقریب تابع مورد نظر است. جمعیت اولیه مشکل از چندین کروموزوم است (مثلاً ۱۰۰ عدد)، که به صورت تصادفی مقداردهی اولیه می شوند. این جمعیت اولیه تلاش می کند فضای پاسخها را به صورت یکنواخت پوشش دهد.

توضیح دوم: استفاده از سایر استراتژی های انتخاب، ترکیب و جهش برای همه حالت ها مشخص است اما نمودار برای تمام حالت های ممکن آورده نشده است زیرا حجم گزارش و پروژه بسیار زیاد می شد. لذا به آوردن نمونه هایی از بررسی توابع هدف با فیتنس فانکشن های مختلف و استراتژی های متفاوت در گام های پروژه کفايت کرده ایم.

بخش سوالات:

سوال ۱) برای کروموزومی که در نظر گرفته اید فضای حالت آن را محاسبه کنید.

در این پروژه، هر کروموزوم شامل ۴۱ ژن می شود که هر کدام می تواند مقداری بین A- و A+ به خود بگیرند. اگر هر ضریب تا ۴ رقم اعشار نمایش داده شود، هر ژن یا ضریب، A ضربدر ۱۰ به توان ۴ حالت دارد. حالا هر کروموزوم از ۴۱ ژن یا ضریب تشکیل شده است پس فضای حالت آن حدوداً برابر است با:

$$(A \times 10^4)^{41} = A^{41} \times 10^{64}$$

که یک فضای حالت بسیار بزرگ و غیرقابل جستجو به روشهای brute-force است. همینجاست که الگوریتم ژنتیک وارد عمل می شود.

سوال ۲) دو تا از ایده هایی که از نظر شما میتواند باعث سریعتر همگرا شدن این مسئله شود را توضیح دهید.

الف. انتخاب نخبگان: (Elitism)

نگه داشتن بهترین کروموزوم ها از نسل فعلی بدون تغییر در نسل بعد باعث حفظ کیفیت و جلوگیری از فراموشی جواب های خوب می شود. این کار در کد ما با پارامتر ELITISM_COUNT پیاده شده است.

ب. تنظیم نرخ جهش به صورت تطبیقی: (Adaptive Mutation)

در ابتدای فرایند نرخ جهش زیاد نگه داشته می شود تا تنوع بالا باشد، و به مرور که جمعیت بهتر می شود نرخ جهش کاهش می یابد تا از نوسان اضافی جلوگیری شود. در کد با تابع self._adjust_parameters(gen) این استراتژی اجرا شده است.

سوال ۳) استراتیهای متفاوتی برای انتخاب نسل بعد در الگوریتمهای ژنتیک وجود دارد. درباره دو مورد از آنها توضیح دهید.

الف. انتخاب تورنمنتی (Tournament Selection):

در این روش، چند کروموزوم به صورت تصادفی انتخاب شده و بهترین آنها به نسل

بعد می رود. این روش تعادل خوبی بین بهره برداری و اکتشاف دارد و فشار انتخابی متوسطی دارد.

ب. انتخاب ترتیبی (Rank Selection): کروموزوم ها بر اساس کیفیت مرتب شده و به آنها وزن اختصاص می گیرد (رتبه های بهتر وزن بیشتری دارند). این روش از غلبه کروموزوم های خیلی خوب جلوگیری کرده و از همگرایی زودرس کم می کند.

سوال ۴) یکی از چالشهای اصلی در الگوریتم های ژنتیک، جلوگیری از همگرایی زودرس است. این پدیده زمانی رخ می دهد که جمعیت به سرعت به یک نقطه بهینه محلی همگرا می شود و از کشف راه حل های بهتر باز می ماند. دو روش برای جلوگیری از همگرایی زودرس در الگوریتم ژنتیک نام ببرید و توضیح دهید که هر کدام چگونه باعث افزایش تنوع ژنتیکی در جمعیت می شوند.

الف. افزایش تنوع ژنتیکی با جهش تصادفی (Random Mutation): با اعمال جهش های غیرقابل پیش بینی به بخشی از ژن ها، جمعیت از افادن در یک نقطه بهینه محلی خارج می شود.

ب. نگه داشتن تنوع از طریق انتخاب تصادفی محدود (Tournament or Rank): انتخاب هایی مانند تورنمنتی یا رنکی، به جای انتخاب قطعی بهترین ها، باعث ورود کروموزوم های مختلف تر به نسل های بعدی می شوند و جلوی شیوه شدن سریع جمعیت را می گیرند.

سوال ۵) یکی از توابع خطایی که در کارهای آماری و مسائل یادگیری ماشین استفاده می شود تابع R-squared است. نحوه کار این تابع را توضیح دهید و توضیح دهید آیا در انجام این مسئله میتواند کاربردی باشد یا نه؟

ضریب تعیین یا R-squared یک معیار آماری است که نشان می دهد چه مقدار از تغییرات داده های واقعی توسط مدل شما توضیح داده می شود. این معیار بیشتر در مدل های رگرسیون به کار می رود، ولی برای سنجش کیفیت تقریب توابع نیز بسیار مفید است. فرمول کلی آن به صورت زیر است:

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

عبارت موجود در صورت کسر: مجموع مربعات خطاهای بین مقدار واقعی و مقدار پیش بینی شده

عبارت موجود در مخرج کسر: مجموع مربعات کل بین مقدار واقعی و میانگین مقادیر واقعی

مدل بسیار خوب است و به خوبی داده ها را تقریب زده است	آر ۲ نزدیک به ۱
مدل تقریباً نصف واریانس داده ها را توضیح می دهد	حدود ۰/۵
مدل هیچ توضیحی برای داده ها ندارد (مثل حدس زدن میانگین)	نزدیک به صفر
مدل از حدس زدن میانگین هم بدتر عمل کرده است	منفی

مزایای استفاده از آر^۲ در پروژه ما این است که:

قابل تفسیر است: مقدار عددی بین صفر و یک می دهد که می توان به سادگی آن را تحلیل کرد.

مقایسه بین مدل ها: اگر بخواهیم روش های مختلف را با هم مقایسه کنیم، آر^۲ عدد واضحی برای مقایسه می دهد.

مکملی خوب برای MSE: ممکن است دو مدل نزدیک به هم داشته باشند ولی یکی از آنها تطابق بهتری با ساختار کلی تابع داشته باشد که این مسئله با آر^۲ بهتر نشان داده می شود.

در پروژه نیز از Fitness Evaluation با عنوان R-squared استفاده شده است و جزو فیتنس فانکشن های مفید است.



این ویدئو یوتوب باعث درک اولیه من از الگوریتم ژنتیک شد که لازم دیدم اینجا لینک ویدئو رو قرار بدم

Part Two: Pentago Game

Here we want to implement the MinMax algorithm in Pentago Game, check the results of our agent in different depths, with(out) alpha-beta pruning and, and so on.

How each function works and its inputs and outputs are provided as comments in the project. According to the basic algorithm of MiniMax in games I have a `minimax_no_pruning()` function:

```
def minimax_no_pruning(self, board, depth, is_maximizer):
    """
    Minimax algorithm without alpha-beta pruning.

    Args:
        board: The current game board
        depth: Search depth remaining
        is_maximizer: True if current player is maximizing

    Returns:
        The best score for the current position
    """
    self.nodes_visited += 1

    # Check for terminal states
    winner = self.check_winner(board)
    if winner is not None:
        if winner == -1:  # Computer wins
            return 1000
        elif winner == 1:  # Human wins
            return -1000
        else:  # Draw
            return 0

    # If maximum depth reached, evaluate board
    if depth == 0:
        return self.evaluate_board(board)

    player = -1 if is_maximizer else 1
    moves = self.get_possible_moves(board, player)

    if is_maximizer:
        best_score = float('-inf')
        for move in moves:
            new_board = self.apply_move(board, move, player)  # apply_move
already does copying
            if new_board is None:
                continue
            score = self.minimax_no_pruning(new_board, depth - 1, False)
            if score > best_score:
                best_score = score
    else:
        best_score = float('inf')
        for move in moves:
            new_board = self.apply_move(board, move, player)  # apply_move
already does copying
            if new_board is None:
                continue
            score = self.minimax_no_pruning(new_board, depth - 1, True)
            if score < best_score:
                best_score = score
    return best_score
```

```
        continue
        score = self.minimax_no_pruning(new_board, depth - 1, False)
        best_score = max(score, best_score)
    return best_score
else:
    best_score = float('inf')
    for move in moves:
        new_board = self.apply_move(board, move, player) # apply_move
already does copying
        if new_board is None:
            continue
        score = self.minimax_no_pruning(new_board, depth - 1, True)
        best_score = min(score, best_score)
    return best_score
```

But When we run the game with this basic algorithm with depth = 2 for 20 games, It lasts about 5 minute to reach the result that computer wins all 20 games.

I won't run this for 50 to 100 times without pruning because it takes a lot of time. I change the depth to 1: [0.5 seconds, 20 wins for computer.](#)

The algorithm works well. But the problem is Time. When I play with the flag ui=True, It takes a lot of time for computer to decide in each move. So we use the pruning:

By using Alpha-Beta pruning, Depth = 2:

20 games:

Last: 2m 4.9s

It means it work well to reduce the duration of running the code. So by using the pruning, the minimax algorithm will be boosted.

گزارش فارسی برخی از مفاهیم به کاربرده شده و ابهام های موجود در پروژه:

طبق مطالب درس می دانیم که **evaluation function** وظیفه تولید کردن یک تخمین درست برای **state value** هر **state** از درخت حالات را دارد. در اینجا تابع **evaluation** تعریف شده توسط من به این صورت است که:

اگر کامپیوتر برنده شود: ۱۰۰۰

اگر حریف برنده شود: -۱۰۰۰

اگر مساوی باشد: ۰

در غیر اینصورت براساس عوامل زیر امتیازدهی می کند:

تعداد مهره های متوالی هر بازیکن، کنترل موقعیت های مرکزی، پتانسیل ایجاد خطوط برنده

تحلیل نتایج: با عمق جستجوی ۲

با این عمق کامپیوتر می تواند حرکات کوتاه مدت را به خوبی پیش بینی کند. اما برای پیش بینی استراتژی های بلندمدت ضعیف است. در برابر بازیکن رندوم معمولاً بهتر عمل می کند. ولی اگر عمق افزایش یابد در برابر بازیکنی که همیشه آپتیمال رفتار می کند شکست ناپذیر می شویم ولی در برابر بازیکن رندوم چندان عملکرد خوبی نشان نخواهد داد.

سوالات:

۱. آیا میان عمق الگوریتم و پارامترهای حساب شده در بخش بالا روابطی میبینید؟ بررسی کنید که عمق الگوریتم چه تاثیراتی بر روی شанс پیروزی، زمان و گره های دیده شده می گذارد.

هر چه عمق بیشتر باشد، شанс پیروزی بیشتر است. در عمق ۲ کامپیوتر ۲ حرکت جلوتر را می بیند. در عمق ۴-۵ می تواند استراتژی های بلندمدت تری را برنامه ریزی کند. در عمق های بالاتر از ۶ تقریباً می توان گفت کامپیوتر بی نقص بازی می کند و شکست دادنش بسیار دشوار است. البته از یک عمق به بعد دیگر تغییر محسوسی احساس نمی شود چون بازی به حالت پرفکت پلی نزدیک می شود.

اما مشکل بزرگ عمق زیاد، زمان زیاد است. هر چه عمق افزایش یابد، زمان به صورت نمایی رشد می کند. منطقاً هر چه که عمق بیشتری از درخت را پایین برویم، تعداد نودهای بیشتری را نیز می بینیم.

۲. آیا میتوان ترتیب دیدن فرزندان هر نод را به گونه ای انتخاب کنیم که بیشترین هرس را داشته باشیم؟ اگر جواب شما مثبت است روش خود را توضیح دهید و در غیر اینصورت توضیح دهید که چرا این عمل امکان پذیر نیست.

پاسخ بله است. این نه تنها امکان پذیر است بلکه یکی از روش های بهینه سازی الگوریتم **minimax** نیز محسوب می شود. روش های پیشرفته برای بهینه سازی ترتیب حرکات:

(الف) مرتب سازی حرکات (Move Ordering):

```
def get_ordered_moves(self, board, player):
    moves = []
```

```

for i in range(6):
    for j in range(6):
        if board[i][j] == 0:
            for block in range(4):
                for dir in ["cw", "ccw"]:
                    moves.append((i, j, block, dir))

# مرتبسازی بر اساس معیارهای هوشمندانه
moves.sort(key=lambda m: self.move_heuristic(board, m, player), reverse=True)
return moves

def move_heuristic(self, board, move, player):
    row, col, block, dir = move
    score = 0

    # 1. اولویت به حرکات برنده فوری
    new_board = self.apply_move(self.copy_board(board), move, player)
    if self.check_winner(new_board) == player:
        return float('inf')

    # 2. اولویت به مرکز و نقاط استراتژیک
    if (row, col) in [(2,2), (2,3), (3,2), (3,3)]:
        score += 3

    # 3. اولویت به ایجاد خطوط ئ تابی
    score += self.count_potential_lines(new_board, player) * 2

    return score

```

ب) تکنیک killer Heuristic : حرکاتی که در سطوح بالاتر درخت منجر به هرس شده اند را در اولویت قرار می دهیم.

```

def __init__(self, ...):
    self.killer_moves = {} # ذخیره حرکات کشته شده در هر عمق

def get_ordered_moves(self, board, player, depth):
    moves = []
    killers = self.killer_moves.get(depth, [])

    # اول حرکات کشته شده را اضافه کنید
    for move in killers:
        if self.is_valid_move(board, move):
            moves.append(move)

    # بقیه حرکات

```

ج) تاریخچه حرکات (History Heuristic)

```

def __init__(self, ...):
    self.history_table = np.zeros((6, 6, 4, 2)) # ذخیره امتیاز حرکات تاریخی

def update_history(self, move, depth):
    row, col, block, dir_idx = move
    self.history_table[row][col][block][0 if dir == "cw" else 1] += 2 ** depth

def get_ordered_moves(self, ...):
    # ...
    moves.sort(key=lambda m: self.history_table[m[0]][m[1]][m[2]][0 if m[3]=="cw" else 1], reverse=True)
    return moves

```

۳. توضیح دهید و بگویید که با پیشرفت این بازی چه تغییراتی میکند؟

ضریب انشعاب یا برنچینگ فکتور به میانگین تعداد حرکات ممکن در هر حالت از بازی گفته میشود. این پارامتر تعیین می کند که هر نod در جستجوی درختی چند فرزند داشته باشد. در ابتدای بازی پتاگو ۳۶ خانه خالی * ۴ بلوک چرخشی * ۲ جهت چرخش وجود دارد، یعنی حدود ۲۸۸ حرکت ممکن! در عمل بسته به استراتژی بازیکن در طول بازی برنچینگ فکتور بین ۵۰-۳۰ است. اما با پیشرفت بازی این مقدار برنچینگ فکتور کاهش می یابد.

۴. توضیح دهید که چرا به هنگام هرس کردن الگوریتم بدون از دست دادن دقت خود سریعتر میشود.

هنگام هرس ما صرفا شاخه هایی که شانسی برای انتخاب شدن به عنوان **max** یا **min** بسته به موقعیت را ندارند، از ابتدا یا میانه مسیر بررسی کردن رها می کنیم و آن ها و فرزندانشان را بررسی نمی کنیم. در واقع با **branching factor** را کاهش می دهیم و این کار با کنار گذاشتن موارد بیهوده است که باید آنها را می خواندیم و بررسی می کردیم و در نهایت به این نتیجه می رسیدیم که این شاخه مسیر انتخابی ما نبوده است، در حالی که از قبل هم با هرس کردن می توانستیم به همان نتیجه بررسیم و دیگر تا این عمق پایین نیاییم و به بررسی خودش و فرزندانش نپردازیم. بنابراین دقت الگوریتم عوض نمی شود و فقط از دین تعدادی از شاخه های بیهوده جلوگیری می شود.

۵. چرا در حالاتی که حریف به صورت شانسی عمل میکند، مانند این پروژه استفاده از **minimax** بهینه ترین روش نیست؟ چه الگوریتمی میتواند جایگزین این الگوریتم باشد؟ توضیح دهید.

در مباحث درس نیز این مسئله وجود داشت که الگوریتم **minimax** یک پیش فرض دارد: حریف نیز همانند من همیشه **optimal** بازی می کند.

الگوریتم با این فرض انجام می شد و تصمیم می گرفت. اما وقتی حریف انتخاب های تصادفی انجام دهد، راه های بهتری و انتخاب های بهینه تری

در درخت تصمیم وجود خواهد داشت که الگوریتم **minimax** آنها را نمی بیند و با پیش فرض غلط حریف باهوش، راه با پاداش کمتر را انتخاب می کند. در واقع الگوریتم **minimax** به دنبال استراتژی های کاملاً ایمن برای پیروزی است و این در حالی است که در مقابل حریف رندوم می توان با ریسک بیشتر سریعتر برنده شد.

الگوریتم های جایگزین در حالتی که حریف رندوم بازی کند:

الف. الگوریتم بهینه ترین جایگزین: به جای فرض بدترین حالت ممکن، میانگین امتیاز را محاسبه می کند و برای حریف تصادفی مناسب است:

```
def expectimax(board, depth, is_maximizer):
    if depth == 0 or game_over(board):
        return evaluate(board)

    if is_maximizer:
        value = -infinity
        for move in get_moves(board, AI_PLAYER):
            new_board = make_move(board, move)
            value = max(value, expectimax(new_board, depth-1, False))
        return value
    else:
        value = 0
        moves = get_moves(board, OPPONENT)
        for move in moves:
            new_board = make_move(board, move)
            value += expectimax(new_board, depth-1, True) * (1/len(moves))
        return value
```

چرا این الگوریتم بهتر عمل می کند؟ چون به جای فرض هوشمندی کامل حریف، تصادفی بودن را مدل می کند. هر حرکت حریف با احتمال مساوی در نظر گرفته می شود.

حرکاتی که در بیشتر سناریو ها خوب هستند را انتخاب می کند نه فقط حرکاتی که در بدترین حالت قابل قبول هستند.

نسبت به الگوریتم مینی مکس به محاسبات کمتری نیاز دارد و می تواند با عمق بیشتر اجرا شود.