Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 3 з дисципліни
«ПІІС»

Виконав(ла) *ІТ-02 Макаров И.С.*
(шифр, прізвище, ім'я, по батькові)

Перевірив _____ (прізвище, ім'я, по батькові)

Київ 2021

ЗМІСТ

МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи - вивчити основні підходи до формалізації алгоритмів знаходження рішень задач в умовах протидії. Ознайомитися з підходами до програмування алгоритмів штучного інтелекту в іграх з повною інформацією.

ЗАВДАННЯ

Завдання:

1. Реалізувати алгоритм ***Negamax*** (можна вносячи зміни в алгоритм другої лабораторної, а можна реалізувавши нову гру, наприклад, хрестики-нолики або Гомоку).
2. Реалізувати алгоритм ***Negamax з альфа-бета відсіканням*** (можна вносячи зміни у алгоритм другої лабораторної).
3. Реалізувати алгоритм **NegaScout.**

ВИКОНАННЯ

Для прикладу, на якому я буду реалізовувати алгоритмів, я взяв гру «Гомоку». В рамках лабораторної роботи, я переписав клієнт та реалізував примітивний ШІ опонент для гри. Написаний ШІ базується на алгоритмі negamax або nega_scout, та з ціллю оптимізації алгоритму я написав модифікацію до negamax з Alpha Beta відсіканнями. **Надалі мій ШІ також може називатись агентом, супротивником, опонентом.**

Зупинятись на тому як функціонує сам клієнт для гри я не буду, в нас всеж пара алгоритмів, тому перейдемо одразу до реалізації супротивника.

Метод, що відповідає за хід агента виглядає так, як зображено знизу, приймаючи на вхід стан дошки, що є зараз та колір того хто ходить (завжди Black, бо за White граємо ми). Повертає метод нову дошку, що замінить поточну.

```
kinn4 +1 *
def make_move(self, current_board, color_making_move):
    depth = self.difficulties_to_depth[self.difficulty]

    # _, new_board = self.plain_negamax(current_board, depth, color_making_move)
    # _, new_board = self.negamax(current_board, depth, color_making_move, float('-inf'), float('inf'))
    _, new_board = self.nega_scout(current_board, depth, color_making_move, float('-inf'), float('inf'))

    # _, new_board = self.plain_minimax(current_board, depth, color_making_move)
    # _, new_board = self.minimax(current_board, depth, color_making_move, float('-inf'), float('inf'))

    return new_board
```

Як видно тут закоментовані всі можливі алгоритми на базі яких мій агент може приймати рішення, перші три відносяться до нашої лаби, два останніх до попередньої.

Ну далі тут нічого не звичайного, просто три алгоритми, та і все, евристика та всі інші деталі в мене лишились з попередньої лаби.

Просто negamax, без альфа-бета

```
new *
def plain_negamax(self, board: Board, depth: int, color_making_move) -> Tuple[int, Board]:
    previous_color = CheckerType.BLACK if color_making_move == CheckerType.WHITE else CheckerType.WHITE
    if depth == 0 or board.is_game_winner(previous_color):
        return -board.evaluate_board(whos_move=previous_color), board

    max_eval = float('-inf')
    best_board = None
    for new_board in self.get_all_possible_children_boards(board, color_making_move):
        new_evaluation = self.plain_negamax(new_board, depth - 1, previous_color)[0]

        if new_evaluation > max_eval:
            max_eval = new_evaluation
            best_board = new_board

    return -max_eval, best_board
```

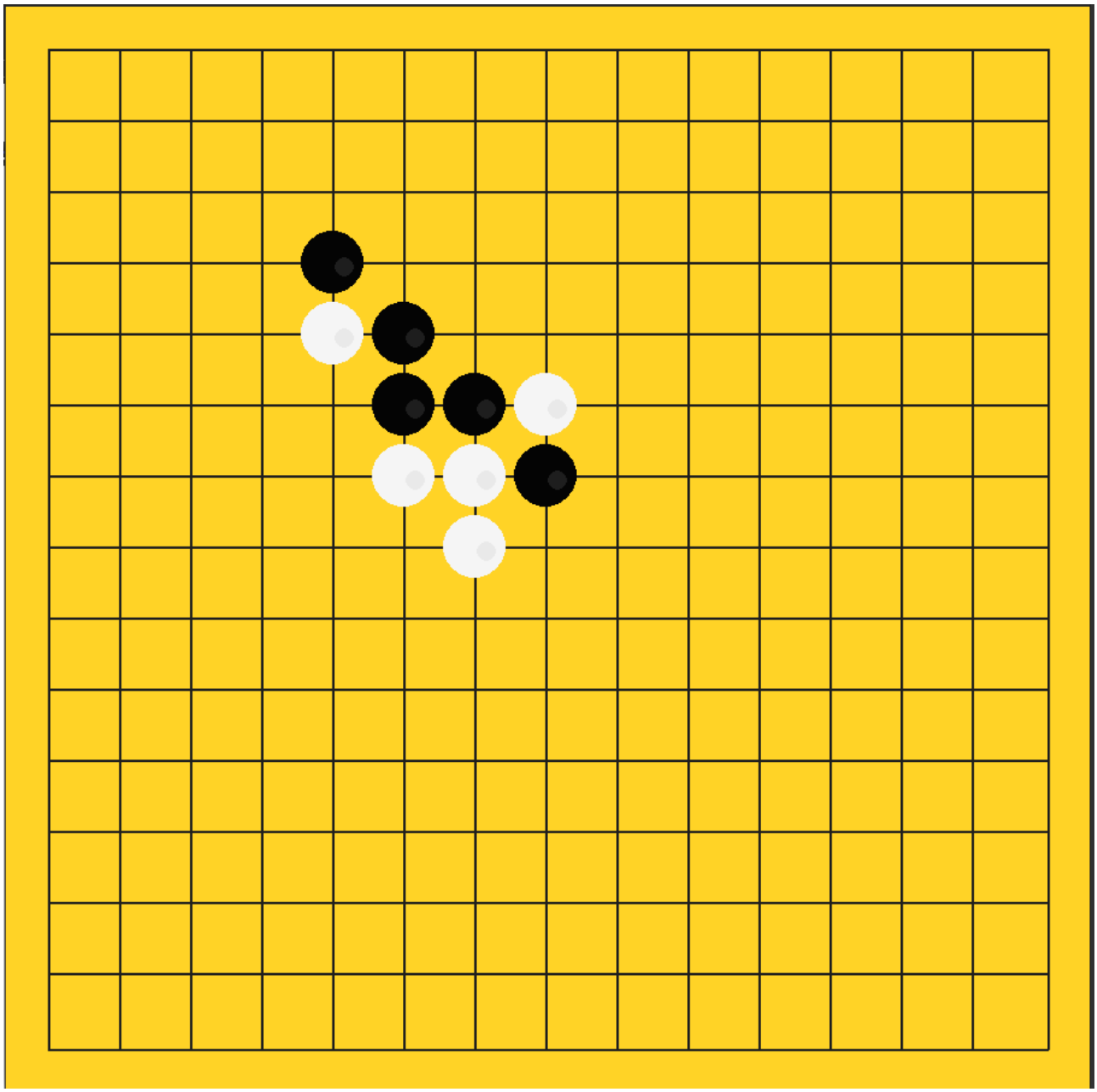
negamax, але вже з альфа-бета

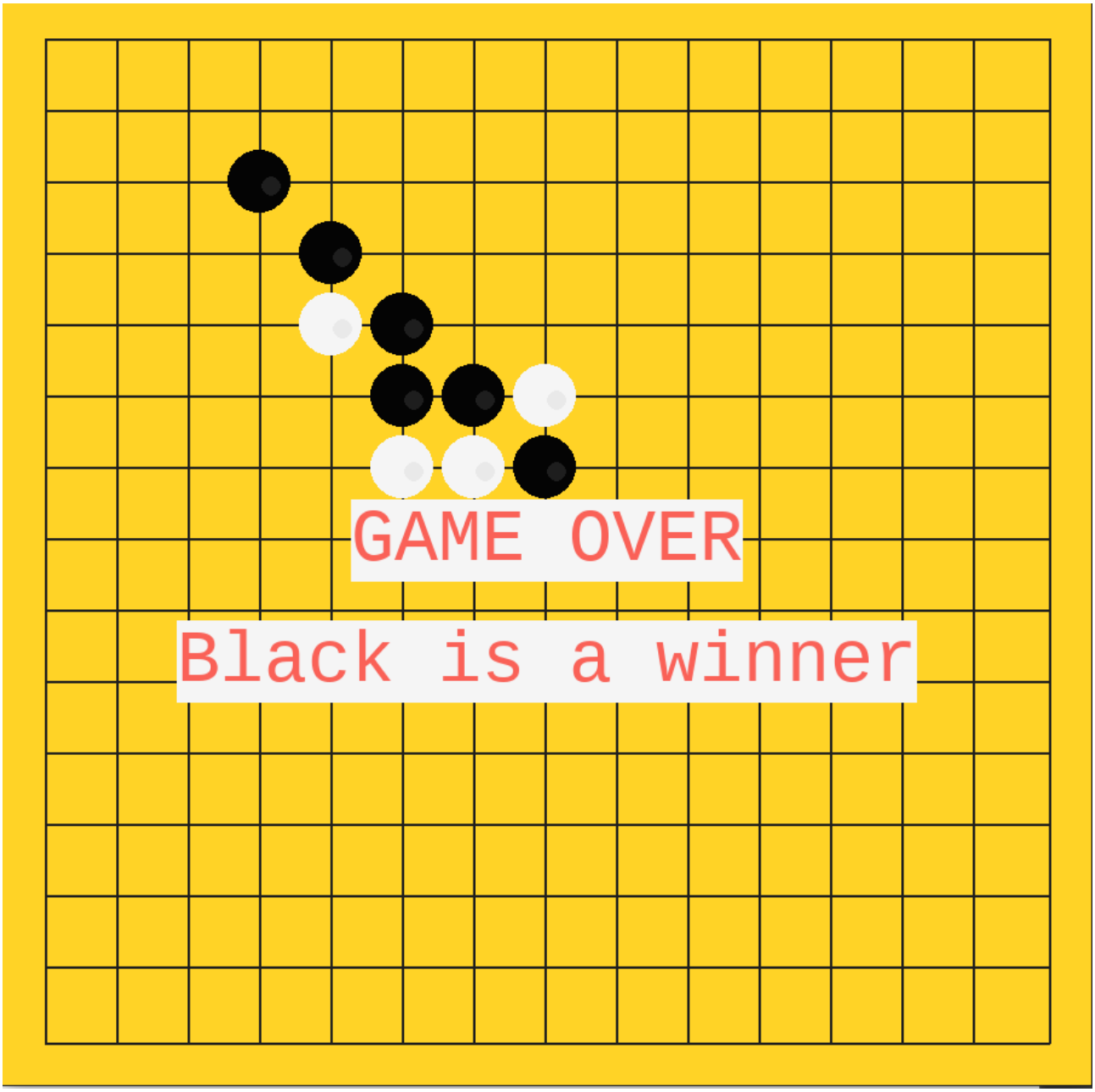
```
def negamax(self, board: Board, depth: int, color_making_move, i_alpha, i_beta) -> Tuple[int, Board]:
    previous_color = CheckerType.BLACK if color_making_move == CheckerType.WHITE else CheckerType.WHITE
    if depth == 0 or board.is_game_winner(previous_color):
        return -board.evaluate_board(whos_move=color_making_move), board

    max_eval = float('-inf')
    best_board = None
    for new_board in self.get_all_possible_children_boards(board, color_making_move):
        new_evaluation = self.negamax(new_board, depth - 1, previous_color, -i_beta, -i_alpha)[0]

        if new_evaluation > max_eval:
            max_eval = new_evaluation
            best_board = new_board

        i_alpha = max(new_evaluation, i_alpha)
        if i_beta <= i_alpha:
            break

    return -max_eval, best_board
```

І, останній, це nega scout

```
def nega_scout(self, board: Board, depth: int, color_making_move, i_alpha, i_beta):
    previous_color = CheckerType.BLACK if color_making_move == CheckerType.WHITE else CheckerType.WHITE
    if depth == 0 or board.is_game_winner(previous_color):
        return -board.evaluate_board(whos_move=color_making_move), board

    max_eval = float('-inf')
    best_board = None
    b = i_beta
    for board_idx, new_board in enumerate(self.get_all_possible_children_boards(board, color_making_move)):
        new_evaluation = self.nega_scout(new_board, depth - 1, previous_color, -b, -i_alpha)[0]

        if i_alpha < new_evaluation < i_beta and board_idx > 0:
            new_evaluation = self.nega_scout(new_board, depth - 1, previous_color, -i_beta, -i_alpha)[0]

        if new_evaluation > max_eval:
            max_eval = new_evaluation
            best_board = new_board

        i_alpha = max(new_evaluation, i_alpha)
        if i_beta <= i_alpha:
            break

        b = i_alpha + 1

    return -max_eval, best_board
```

ПРИКЛАД РОБОТИ





Висновок: багато писати тут не бачу сенсу, основну частину висновку, як мені здається я написав на початку у вступі. Робота виявилась досить не поганою, хоча я і не сильно бачу сенсу в negamax алгоритмі, так як