ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ

КУРСОВ ПРОЕКТ

ПО КОМПЮТЪРЕН ИНТЕЛЕКТ

Тема: Софтуерна реализация на морски шах чрез методи на изкуствен интелект

Изготвил:

Боян Зарев
фак. номер: 123222004
група: 42
специалност: КСИ
курс: IV
e-mail: bzarev@tu-sofia.bg

Ръководител:

проф. д-р инж. Румен Трифонов

София, 2025

Съдържание

1. Въведение 1
2. Интелигенти агенти в игрите 1
3. Минимакс алгоритъм 2
   3.1. Алфа-бета отсичане 3
4. Софтуерна реализация на морски шах 4
   4.1. Репрезентация на поле 4
   4.2. Минимакс с алфа-бета отсичане 5
   4.3. Графичен интерфейс и част от логиката на минимакс 7
Литература 10
Приложения 11

_

1. Въведение

Морският шах е една от най-популярните и лесни за разбиране стратегически игри. Правилата на играта са прости – двама участници се редуват в поставянето на съответните си символи („X“ и „O“) върху квадратна решетка с размери 3x3. Целта на играта е формиране на последователност от три еднакви символа по хоризонтала, вертикала или диагонала. Поради своята елементарност, играта се разглежда като класически пример за изследване на стратегии, решения и изкуствен интелект в игрите.

В този курсов проект е реализиран софтуерен вариант на морски шах, в който единият играч е интелигентен агент. За вземането на решения от страна на агента е използван алгоритъма минимакс с алфа-бета отсичане. Програмата също така включва и графичен интерфейс, където потребителят може да избере в кое поле да сложи своя символ, както и бутон, който позволява на потребителя да смени символа, с който играе. Програмата следи броя на спечелените, изгубените и завършилите се наравно игри. Играта е направена на python, а за графичния интерфейс е използвана графичната библиотека pygame.

Имплементираният алгоритъм гарантира, че интелигентният агент не може да бъде победен. Възможните изходи на играта са победа на агента или равенство.

2. Интелигенти агенти в игрите

Интелигентен агент е система, която възприема информацията от средата, в която се намира и тази информация обработва и анализира с помощта на знания, правила, алгоритми и/или методи на машинно обучение. Въз основа на възприетите знания от средата и метода на взимане на решения, агентът влияе върху средата чрез „изпълнители“. Агентът насочва своите действия към точно определени една или повече цели. За интелигентен агент се казва, че е рационален ако той възприема добре информацията от средата, получава най-възможно добра представа за състоянието ѝ и взима възможно най-доброто решение насочено към дадената цел.

Под състояние на играта се подразбира конфигурация на агента и средата, в която се той намира. Конкретно за морския шах, състояние на играта е точното разположение на всички символи („X“, „O“ и празна клетка) на полето. При моделирането на игрите необходимо е да се определи пространството на състоянията, което представлява сбор на последствия след изпълнението на всички възможни акции. Акция е преход от едно състояние в друго. Кога се определят всички възможни акции, получава се дърво на решенията.

В игрите акциите, които агентът прави биват наградени и наказани. Целта на агента е да донася решения, където ще бъде повече възнаграден, а по-малко наказан. Акциите биват оценени чрез евристична функция. Евристичната функция преценява състоянието на играта въз основа на това, колко е полезно.

Евристичните функции могат да бъдат допустими и недопустими. При допустимите евристики състоянието на играта може да се подцени, но не и да се прецени. Недопустимите характеристики преценяват състоянието на агента. Когато състоянието се подцени, тогава агентът прави решения, които може би няма да доведат до крайната цел, но поне ще прави това, което е в полза на него, докато при преценяването на състоянието агентът ще донася решения, които водят до загуба или до много лошо състояние на играта, а пък той ще мисли да прави най-доброто решение.

3. Минимакс алгоритъм

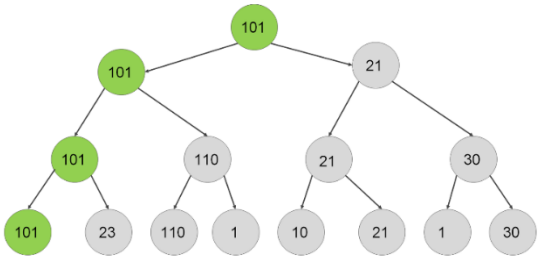
Минимакс е алгоритъм за взимане на решения в игри за двама. Целта на алгоритъма е да избере оптимален ход предполагайки, че и двата играча ще направят най-доброто решение при дадено състояние на играта. От гледна точка на единия играч, ако той взима най-доброто решение в даден момент, то това ще доведе до добро състояние на играта според евристичната функция, докато ако противникът взима добри решения, то това ще намали неговото предимство. С други думи казано, печалбата на единия играч води до загубата на другия.

- Максимизиращ играч (Max) е този, чиято цел е да максимизира своя резултат. Избира хода, който води до най-висока възможна оценка, като се предполага, че противникът ще играе оптимално.
- Минимизиращ играч (Min) е този, чиято цел е да минимизира резултата на максимизатора. Избира хода, който води до най-ниска възможна оценка за максимизатора, като се предполага, че той ще играе оптимално.

Стъпките на алгоритъма:

1. Генериране на дърво на решенията от дадено състояние на играта. Всеки възел представлява състояние на играта, а всеки ръб акцията, която е генерирала това състояние.
2. Асоцииране на стойности към крайните възли на дървото чрез евристичната функция.
3. Разпространяване на оценките нагоре по дървото. Ако ходът е на максимизиращият играч, алгоритъмът избира възел с най-голямата стойност от евристичната функция. Ако е ходът на минимизиращият играч, алгоритъмът избира възел с най-малка стойност от евристичната функция.
4. Играчът избира хода, който води до най-добрата оценка. Този ход всъщност ще е корена на дървото.

На фигура 1 е показано примерно дърво. Крайните възли са получили следните стойности от евристичната функция: 101, 23, 110, 1, 10, 21, 1, 30. Максимизиращият играч е на ход. Той извиква алгоритъма минимакс и прави три рекурсивни извиквания, където на последното задава стойностите на възлите. Максимизиращият играч избира ход, който ще бъде в полза на него. В конкретния пример, максимизиращият играч трябва да избере възлите с по-големите стойности, в случая 101, 110, 21 и 30. В по-горното рекурсивно извикване алгоритъмът прави решение от гледна точка на минимизиращият играч, съответно ще избира възлите с по-малки стойности, в случая 101 и 21. В корена на дървото, от където е и извикан минимакс, максимизиращият играч ще избере възела с по-голямата стойност, който е 101 и ще направи решение, което ще доведе състоянието на играта до състоянието на играта на този възел.



(Фиг.1) Илюстрация на минимакс

3.1. Алфа-бета отсичане

Алфа-бета отсичане е оптимизационна техника за алгоритъма минимакс. То намалява броя на възлите, които се оценяват в дървото на решенията, като елиминира клоновете, които не могат да окажат влияние върху крайното решение. Това се постига чрез поддържане на две променливи **alpha** и **beta**.

- alpha: Най-добрата (най-висока) стойност, която максимизаторът може да гарантира при текущото състояние.
- beta: Най-добрата (най-ниската) стойност, която минимизаторът може да гарантира при текущо състояние.

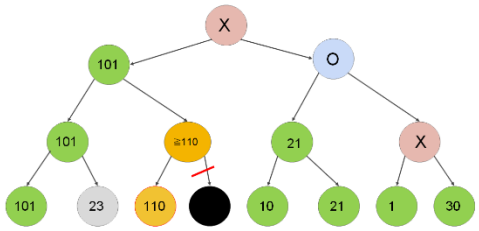
Алфа-бета отсичането обхожда дървото на играта по същия начин като минимакс, но премахва клонове, които не е необходимо да бъдат изследвани.

Стъпките на алгоритъма:

1. Инициализиране на променливите **alpha** и **beta**. **Alpha** става голяма отрицателна стойност, а **beta** голяма положителна стойност.
2. Изчислява се стойността на дъщерния възел с рекурсивно извикване на алгоритъма. Ако стойността на дъщерния възел е по-голяма от текущата стойност на **alpha**, тогава **alpha** присвоява стойността на дъщерния възел. Ако **alpha** е по-голяма или равна на **beta**, останалите дъщерни възли не се посещават.

3. Изчислява се стойността на дъщерния възел с рекурсивно извикване на алгоритъма. Ако стойността на дъщерния възел е по-малка от текущата стойност на **beta**, тогава **beta** приема стойността на дъщерния възел. Ако **beta** е по-малка или равна на **alpha**, останалите дъщерни възли не се посещават.

На фигура 2 е показано примерно дърво на решенията, илюстриращо алфа-бета отсичане при възлите със стойности 101 и 110. При изчисляване на възела със стойността 101, променливата **beta** е получила нова стойност 101. При следващия възел минимизаторът знае, че максимизаторът ще избере възел със стойност по-голяма или равна на 110, което става нова стойност на **alpha**. Във всеки случай на минимизаторът по-добър ще бъде ход, който води до възела със стойност 101. Променливата **beta** е по-малка от **alpha**, което ще доведе до отсичане на възела оцветен с черно на картинката.



(Фиг. 2) Илюстрация на алфа-бета отсичане

4. Софтуерна реализация на морски шах

4.1. Репрезентация на поле

Полето е представено като двумерен масив. При създаването му инициализират се 9 празни клетки. Функцията **reset** изтрива текущото състояние на полето и му присвоява инициализиращото. Функцията **insert_move** като параметри взима символа на играча („X“ или „O“), реда и колоната. Целта на тази функция е вкарването на играча на съответната позиция подадена като параметър. Функцията хвърля грешка в случай ако като параметър са подадени невалидни ред или стълб, подаден невалиден играч или когато в клетката има вече играч. Описаните член-променливи и член-функции се намират в класа **Board**.

```
class Board:
    def __init__(self):
        self.PLAYER_X = "X"
        self.PLAYER_O = "O"
        self.EMPTY = " "

        self.state = [[self.EMPTY for _ in range(3)] for _ in range(3)]

    def insert_move(self, row, col, player):
        if player not in [self.PLAYER_X, self.PLAYER_O]:
            raise ValueError("Invalid player")
        if row < 0 or row >= 3 or col < 0 or col >= 3:
            raise ValueError("Invalid cell position ({}, {})".format(row, col))
        if self.state[row][col] != self.EMPTY:
            raise ValueError("Cell already occupied")

        self.state[row][col] = player

    def reset(self):
        self.state = [[self.EMPTY for _ in range(3)] for _ in range(3)]
```

4.2. Минимакс с алфа-бета отсичане

За реализацията на минимакс с алфа-бета отсичане са необходими функция, която ще изпълнява основната философия на минимакс, която рекурсивно ще се извиква, евристична функция и две допълнителни функции, които ще проверяват дали полето е пълно и дали един от играчите е победил.

Кодът на двете допълните функции е следния:

```python
def is_win(player_sign, board):
    # Check rows
    for row in board.state:
        if row[0] == row[1] == row[2] == player_sign:
            return True

    # Check columns
    for col in range(3):
        column = (board.state[0][col], board.state[1][col], board.state[2][col])
        if column[0] == column[1] == column[2] == player_sign:
            return True

    # Check diagonals
    if board.state[0][0] == board.state[1][1] == board.state[2][2] == player_sign:
        return True
    if board.state[0][2] == board.state[1][1] == board.state[2][0] == player_sign:
        return True

    return False

def is_full(board):
    for row in board.state:
        for cell in row:
            if cell == board.EMPTY:
                return False
    return True
```

Функцията **minimax** взима като параметри инстанция от **Board**, дълбочината на претърсването на дървото на решения, алфа, бета, индикатор, който казва дали е на ход максимизиращият или минимизиращият играч и символ, с който агентът играе („X“ или „O“). Последният параметър е необходим, защото в приложението е добавена опция потребителят да си смени символа с който играе. Функцията се дели на две части. Когато е максимизиращият играч на ход, търси се решение с най-добър изход. Когато е минимизиращият играч на ход, търси се решение с най-лош изход. При всяко посещение на възел се проверява дали могат останалите дъщерни възли да се отсекът. В случай, че може агентът излиза от фор цикъла. Кодът е следния:

```python
def minimax(board, depth, alpha, beta, is_maximizing, ai_player):
    human_player = board.PLAYER_O if ai_player == board.PLAYER_X else board.PLAYER_X

    if is_win(ai_player, board):
        return inf
    if is_win(human_player, board):
        return -inf
    if is_full(board):
        return 0  # Draw
    if depth == 0:
        return evaluate(board, ai_player)

    if is_maximizing:
        best_score = -inf
        for r in range(3):
            for c in range(3):
                if board.state[r][c] == board.EMPTY:
                    board.insert_move(r, c, ai_player)
                    score = minimax(board, depth - 1,  alpha, beta, False, ai_player)
                    board.state[r][c] = board.EMPTY  # Undo move
                    best_score = max(score, best_score)
                    alpha = max(alpha, best_score)
                    if beta <= alpha:
                        return best_score
        return best_score
```

```
    else:
        best_score = inf
        for r in range(3):
            for c in range(3):
                if board.state[r][c] == board.EMPTY:
                    board.insert_move(r, c, human_player)
                    score = minimax(board, depth - 1,  alpha, beta, True, ai_player)
                    board.state[r][c] = board.EMPTY  # Undo move
                    best_score = min(score, best_score)
                    beta = min(beta, best_score)
                    if beta <= alpha:
                        return best_score

        return best_score
```

Оценяване с евристичната функция се извършва по няколко критерии:

1. Ако максимизиращият играч има два свои символа и една празна клетка в линия, се прибавя положителна стойност към резултата. Ако минимизиращият играч има същата конфигурация, се изважда същата стойност. По този начин алгоритъмът стимулира търсенето на печеливша трета стъпка и едновременно наказва ситуации, в които противникът е близо до победа.
2. В случай ако играчите имат два знака на ъглите и празна клетка в средата на ред, колона или диагонал се дават допълнителни точки, защото такова подреждане увеличава стратегическите възможности на играча. Ако такава конфигурация принадлежи на противника, се отнемат точки.
3. При победа или загуба, евристичната функция връща ∞, съответно -∞.

Кодът е следния:

```
score_two_signs = 1
score_two_sign_on_edges = 2

def evaluate(board, ai_player):
    # # #
    # if the player_turn is False -> the maximazing player is X
    # if the player_turn is True -> the maximazing player is O
    # # #
    maximazing_player = ai_player
    minimizing_player = board.PLAYER_O if ai_player == board.PLAYER_X else board.PLAYER_X

    # Check for terminal states
    score = 0

    main_diagonal = (board.state[0][0], board.state[1][1], board.state[2][2])
    anti_diagonal = (board.state[0][2], board.state[1][1], board.state[2][0])

    # evaluate the position where theres 2 signs
    for row in board.state:
        if row.count(maximazing_player) == 2 and row.count(board.EMPTY) == 1:
            score += score_two_signs
        if row.count(maximazing_player) == 2 and row.count(board.EMPTY) == 1:
            score -= score_two_signs

    for col in range(3):
        column = (board.state[0][col], board.state[1][col], board.state[2][col])
        if column.count(maximazing_player) == 2 and column.count(board.EMPTY) == 1:
            score += score_two_signs
        if column.count(minimizing_player) == 2 and column.count(board.EMPTY) == 1:
            score -= score_two_signs

    if main_diagonal.count(maximazing_player) == 2 and main_diagonal.count(board.EMPTY) == 1:
        score += score_two_signs
    if main_diagonal.count(minimizing_player) == 2 and main_diagonal.count(board.EMPTY) == 1:
        score -= score_two_signs

    if anti_diagonal.count(maximazing_player) == 2 and anti_diagonal.count(board.EMPTY) == 1:
        score += score_two_signs
    if anti_diagonal.count(minimizing_player) == 2 and anti_diagonal.count(board.EMPTY) == 1:
        score -= score_two_signs
```

```
    # evaluate the move where the player places its sign on the corners
    for row in board.state:
        if row[0] == row[2] == maximazing_player and row[1] == board.EMPTY:
            score += score_two_sign_on_edges
        if row[0] == row[2] == minimizing_player and row[1] == board.EMPTY:
            score -= score_two_sign_on_edges

    for col in range(3):
        column = (board.state[0][col], board.state[1][col], board.state[2][col])
        if column[0] == column[2] == maximazing_player and column[1] == board.EMPTY:
            score += score_two_sign_on_edges
        if column[0] == column[2] == minimizing_player and column[1] == board.EMPTY:
            score -= score_two_sign_on_edges

    if main_diagonal[0] == main_diagonal[2] == maximazing_player and main_diagonal[1] == board.EMPTY:
        score += score_two_sign_on_edges
    if main_diagonal[0] == main_diagonal[2] == minimizing_player and main_diagonal[1] == board.EMPTY:
        score -= score_two_sign_on_edges

    if anti_diagonal[0] == anti_diagonal[2] == maximazing_player and anti_diagonal[1] == board.EMPTY:
        score += score_two_sign_on_edges
    if anti_diagonal[0] == anti_diagonal[2] == minimizing_player and anti_diagonal[1] == board.EMPTY:
        score -= score_two_sign_on_edges

    return score
```

4.3. Графичен интерфейс и част от логиката на минимакс

За реализацията на графичния интерфейс е създаден клас **Game_Window**, който управлява цикъла на играта и логиката на взаимодействието между играча и агента. Класът използва библиотеката **Pygame** за визуализация на полето и контрол на събитията. Част от логиката на минимакс е имплементирана в този клас. Интелигентния агент преди да донесе решението, той подрежда решенията, по начина, на който е много по-вероятно в дървото на решенията да се осъществи алфа-бета отсичане. Колкото по-рано се яви възел с голяма стойност, толкова повече възли ще отсече. В случая, във функцията **find_moves** възлите се пренареждат, така че състоянията, където сложения знак е в ъглите или в полето в средата, да бъдат посетени първи, т.е. първо се посещават възлите, където сбора на индексите на реда и колоната в съответната играна позиция, дава четно число, имайки в предвид, че индексирането на колоните и редовете започва от 0. Функцията **generate_moves** при вход на агента, прави извикване на минимакс във всяко празно поле, сравнява решенията и взима най-доброто. Дълбочината на дървото е 9, защото това е максималният брой на итерации, които алгоритъмът може да направи, защото полето има 9 клетки. Минимакс се извиква с минимизиращият играч, защото агента е направил възможен ход. Кодът е следния:

```
    def find_moves(self):
        moves = []
        even_cells = []
        odd_cells = []
        for r in range(3):
            for c in range(3):
                if self.board.state[r][c] == self.board.EMPTY:
                    if (r + c) % 2 == 0:
                        even_cells.append((r, c))
                    else:
                        odd_cells.append((r, c))
        shuffle(even_cells)
        shuffle(odd_cells)
        moves.extend(even_cells)
        moves.extend(odd_cells)
        return moves
```

```
58      def generate_move(self):
59          if self.turn != self.human_playing:
60              best_move = None
61              best_score = -inf
62              player = self.board.PLAYER_O if not self.human_playing else self.board.PLAYER_X
63
64              for move in self.find_moves():
65                  r, c = move
66                  if self.board.state[r][c] == self.board.EMPTY:
67                      self.board.insert_move(r, c, player)
68                      score = minimax(self.board, 9, -inf, inf, False, player)
69                      self.board.state[r][c] = self.board.EMPTY  # Undo move
70                      if score > best_score:
71                          best_score = score
72                          best_move = (r, c)
73              if best_move:
74                  self.board.insert_move(best_move[0], best_move[1], player)
75                  self.turn = not self.turn
```

Класът **Game_Window** има следните член-променливи:

- col_width: широчина на клетка
- row_height: дължина на клетка
- human_playing: ако е True, потребителят е избрал да играе с „X“, ако е False потребителят е избрал да играе с „O“
- board: инстанция на класа Board
- human_player_score: променлива съхраняваща брой на игрите спечелени от потребителя
- ai_player_score: променлива съхраняваща брой на игрите спечелени агента
- draw_score: променлива съхраняваща брой на игрите, където изходът е бил равен
- change_side_button: копче, което служи да промени символа, с който потребителят играе

```
9       def __init__(self, board=None):
10          self.window = pygame.display.set_mode((WIDTH, HEIGHT_WITH_BAR))
11          pygame.display.set_caption("Tic Tac Toe")
12
13          self.col_width = WIDTH // 3
14          self.row_height = HEIGHT // 3
15
16          self.human_playing = False  # False for PLAYER_X, True for PLAYER_O, TODO add fucntinon to switch sides
17          self.turn = False # False for PLAYER_X, True for PLAYER_O
18
19          self.board = board
20
21          self.human_player_score = 0
22          self.ai_player_score = 0
23          self.draw_score = 0
24
25          button_width = 100
26          button_height = 30
27          self.change_side_button = Button(WIDTH - button_width - GAP, HEIGHT + (BAR - button_height) // 2, button_width, button_height, display_text = "Change side")
```

Функцията **choose_move** обработва събитието, където потребителят избира клетка, където да сложи своя символ. Функцията взима като параметри събитието и координатите на курсора. Кодът е следния:

```
29      def choose_move(self, mouse_pos, event):
30          if self.human_playing == self.turn and event.type == pygame.MOUSEBUTTONDOWN:
31              x, y = mouse_pos
32              if 0 < x < WIDTH and 0 < y < HEIGHT:
33                  col = x // self.col_width
34                  row = y // self.row_height
35
36                  if self.board.state[row][col] == self.board.EMPTY:
37                      player = self.board.PLAYER_O if self.turn else self.board.PLAYER_X
38                      self.board.insert_move(row, col, player)
39                      self.turn = not self.turn
```

Класът **Game_Window** има член-функции **draw_board**, **draw_bar**, **draw_winning_line** и **finish**. Те служат за визуализация на играта в прозореца.

Функцията **loop** реализира основния цикъл на програмата. В него се следят събитията от клавиатурата и мишката, извършват се ходовете на потребителя и агента, обновява се визуализацията и се проверява за край на играта. Цикълът се изпълнява, докато прозорецът не бъде затворен от потребителя. Кодът на функцията е следния:

```
173     def loop(self):
174         running = True
175         clock = pygame.time.Clock()
176         while running:
177             clock.tick(FPS)
178             mouse_pos = pygame.mouse.get_pos()
179             for event in pygame.event.get():
180                 if event.type == pygame.QUIT:
181                     running = False
182                 self.choose_move(mouse_pos, event)
183                 if self.change_side_button.is_clicked(event, mouse_pos):
184                     self.human_playing = not self.human_playing
185                     self.board.reset()
186                     self.turn = False
187
188             self.generate_move()
189             self.draw_board()
190             self.finish()
191
192             self.change_side_button.draw(self.window)
193             self.change_side_button.is_pressed(mouse_pos, pygame.mouse.get_pressed())
194
195             pygame.display.update()
196
197         pygame.quit()
```

Точката на вход е дефинирана в **main.py**. Кодът изглежда по следния начин:

```
5   def main():
6       b = Board()
7       game_window = Game_Window(b)
8       game_window.loop()
9
10  if __name__ == "__main__":
11      main()
```

Литература

1. Artificial Intelligence and Machine Learning Fundamentals – Zsolt Nagy, издателство Kompjuter biblioteka, 2019, ISBN 978-86-7310-544-4
2. Search: Games, Minimax, and Alpha-Beta – видеоурок от MIT OpenCourseWare - https://youtu.be/STjW3eH0Cik?si=KTdo6BcfnCDSQaHu
3. Harvard CS50's Artificial Intelligence with Python – видеоурок от Harvard – част 1. Search - https://youtu.be/5NgNicANyqM?si=p98N33Kvatrp4I0N
4. GeekForGeeks - Alpha-Beta pruning in Adversarial Search Algorithms (23.7.2025) - https://www.geeksforgeeks.org/artificial-intelligence/alpha-beta-pruning-in-adversarial-search-algorithms/
5. GeekForGeeks - Mini-Max Algorithm in Artificial Intelligence (7.4.2025) - https://www.geeksforgeeks.org/artificial-intelligence/mini-max-algorithm-in-artificial-intelligence/
6. Фиг. 1 (Artificial Intelligence and Machine Learning Fundamentals – Zsolt Nagy, стр. 68)
7. Фиг. 2 (Artificial Intelligence and Machine Learning Fundamentals – Zsolt Nagy, стр. 70)

Приложения

1. Сорс-код: https://github.com/boce1/tictactoe_KI_project.git
2. Демонстрационно видео: https://youtu.be/RKImUf7Z9-U?si=uc6pdFi_gLdrUH3V