שם התלמיד : ארי לרנר

שם העבודה : שחמט

שם התלמיד : ארי לרנר

תעודת זהות : 326126877

שם המנחה : נורית קרצר

שם החלופה : מערכות מומחה

תאריך הגשה : 05/04/2020

תוכן

מבוא

במסגרת לימודי הנדסת תוכנה היה עלינו לתכנת משחק לוח קיים ב-python בספריית kivy תוך שימוש בבינה מלאכותית. בפרויקט שלי פיתחתי את משחק השחמט. משחק לוח אסטרטגי קלאסי מופשט וענף ספורט המיועד לשני שחקנים. זהו אחד המשחקים המוכרים והמורכבים ביותר הקיימים בתרבות האנושית ובמהלך שנים רבות נחשב לדוגמת האינטלקט האנושי הטובה ביותר. המשחק נערך על לוח משחק בגודל 8 על 8 משבצות שחורות לבנות. במשחק יש סוגים שונים של כלים כגון: חייל, פרש, רץ, צריח, מלכה ומלך, וכל אחד מאלה נע בצורה ייחודית לו. הדברים הללו הם שגורמים למשחק להיות כל כך מסובך ולכן ומוערך כי במשחק יש בין 43^10 עד 47^10 פוזיציות ייחודיות שיכולות להתקבל. פופולאריות המשחק הביאה לכך שיש וריאציות נוספות רבות של המשחק המקורי כמו שחמט ל-3 שחקנים ושחמט תלת ממדי, אך וריאציות אלו שונות מהותית מהמשחק המקורי לכן אתמקד בגרסה הקלאסית שלו.

לא הרבה לפני שהתחלתי לעבוד על הפרויקט התחלתי להתעניין בשחמט ושיחקתי בו שעות רבות. כיום אי אפשר שמישהו שמשחק בשחמט לא ישמע על chess engines ואני לא הייתי יוצא מן הכלל. תחום "Chess programming" עורר בי עניין רב. שנים ארוכות היה נהוג לחשוב שמחשבים נחותים לבני אדם בכוח המחשבה על בסיס העובדה שאף מחשב לא יכול לנצח אדם בשחמט. עד שבשנת 1997 המחשב "Deep Blue" ניצח את גארי קספרוב, אלוף העולם בשחמט. ניצחון זה היה נקודת מפנה בעולם השחמט והתכנות בכלל.

לא רק שחקן מתחיל יכול לשחק נגד מחשב שחמט. כיום, גם אלופי העולם מעמיקים את הידע שלהם במחשק באמצעות המנועים המתקדמים ביותר כמו Stockfish 13 Alpha zero בעל ניקוד Elo של 3546. כאשר הניקוד הכי גבוה בשחקן אנושי הוא 2882. השיא מוחזק על ידי מגנוס קרלסן. שיטות ואסטרטגיות משחק חדשות מתפתחות בזכות המנועים הללו.

בחרתי את משחק זה מכיוון שרציתי פרויקט מאתגר שאוכל להתעניין לשקוע בו. אני אוהב לזרוק את העצמי לפרויקט מסובך ולהבין איך עלי לפתור את הבעיות הצצות בתהליך הפיתוח. במשחק השחמט שלי יש את כל החוקים והאפשרויות שיש במשחק שחמט קלאסי כמו הצרחה, En Passant, קידום חיילים, stalemate ועוד. במשחק שלי אין את האופציה להחזיר לאחור מהלכים ואין הגבלת זמן

המחשבים המתקדמים ביותר בתחום כיום משתמשים ב-Neural Networks ולמרות שקראתי עליהן הרבה מאוד החלטתי להסתפק באלגוריתם פשוט יותר. האלגוריתם עליו מבוססת הבינה המלאכותית של שחקן המחשב בפרויקט שלי הוא minimax. חוץ מה-minimax נעשה במחשק שלי שימוש בגיזום alpha beta. אלגוריתם ה-minimax מריץ את המשחק מספר

כלשהו של מהלכים קדימה (depth) ונותן ניקוד מספרי לכל פוזיציה מתקבלת ולאחר מכן בוחן ובוחר את המהלך שיביא לניקוד הגבוה ביותר (המחשב מניח שמדובר בשחקן "חכם" שיבחר במהלכים טובים). גיזום אלפה-בטא מאפשר למחשב לבדוק פחות מצבים כדי לאפשר זמן הריצה קצר יותר לחישוב המהלך הטוב ביותר (דבר חשוב מאוד כשמדובר במשחק עם כל כך הרבה פוזיציות אפשריות כמו שחמט). את פונקציית ההערכה שלי ביססתי על המאמר Simplified Evaluation Function אשר סופר את ניקוד הכלים של כל צד לפי ערכים קבועים מראש ומתחשב במיקום הכלים על הלוח באמצעות לוחות הנקראים Piece-Square tables. אופן פעולת האלגוריתם ושיטת ההערכה יוסבר בפירוט רבה בהמשך הספר.

מבנה / ארכיטקטורה

בסיס הידע :

מסכים

במשחק נעשה שימוש במספר מסכים ומטרתם להגביר את נוחות למשתמש. חשוב לציין שלא מדובר בפופ אפים אלה במסכים נפרדים הנמצאים בתוך window manager של kivy. בתוך המסכים נכללים: מסך פתיחה, מסך ההוראות ומסך המשחק עצמו. כל המסכים הללו נוצרים בקובץ chess.kv

קובץ chess.kv

אחד הכלים החזקים ביותר שספריית Kivy מביאה איתה הוא קובץ ה-kv. אשר מאפשר לשלוט על המבנה הגרפי של מסכים וחלונות במשחק. קובץ זה הוא חלק אינטגרלי למסכי הפתיחה אשר נמצאים בפרויקט.

מחלקת מנהל החלונות (WindowManager)

מחלקה זו מאפשרת את המעבר בין מסכים שונים במשחק מבלי ליצור כמות גדולה של פופ אפים (popups). המחלקה מוגדרים החלונות השונים בניהם אפשר לעבור.

```
WindowManager:
    StartWindow:
    RulesWindow:
    Board:
```

מחלקת מסך הפתיחה (StartWindow)

מחלקה זו יוצרת את מסך הפתיחה של המשחק אשר מכיל את שם המשחק וכפתורים (התחלת משחק, חוקים ויציאה מהמשחק).

להלן גרסה מקוצרת של המחלקה ללא אלמנטים עיצוביים (הקוד קוצר להבהרת המבנה)

```
<StartWindow>:
    name: 'first'
    BoxLayout:
        orientation: 'vertical'
        Label:
            text: 'Chess AI'
        BoxLayout
            orientation: 'horizontal'
            Label:
                text: ''
            Button:
                text: 'Play'
                bold: True
                on_release: app.root.current = 'game'
            Button:
                text: 'Rules'
                on_release: app.root.current = "rules"
            Button:
                text: 'Quit'
                on_release: quit()
            Label:
                Text: ''
        Label:
            text: "By Ari Lerner"
```

מחלקת מסך ההוראות (RulesWindow)

מחלקה זו יוצרת את מסך ההוראות של המשחק. היא מכילה תמונה בה כתוב טקסט ההוראות בעברית וכפתור חזרה למסך הראשי.

להלן גרסה מקוצרת של המחלקה ללא אלמנטים עיצוביים (הקוד קוצר להבהרת המבנה)

```
<RulesWindow>:
    name: "rules"
    BoxLayout:
        orientation: "vertical"
        Image:
            source: 'chess_rules_hebrew.png'
        Button:
            text: '<- main menu'
            on_release: app.root.current = "first"
```

מחלקת לוח המשחק (Board)

מחלקה זו יוצרת את מסך לוח המשחק ומכילה תמונה של הלוח.

להלן גרסה מקוצרת של המחלקה ללא אלמנטים עיצוביים (הקוד קוצר להבהרת המבנה)

```
<Board>:
    name: 'game'
    canvas.before:
        Rectangle:
            pos: self.pos
            size: self.size
            source: 'white_green_board.png'
```

קובץ main.py

מחלקת משחק

מחלקה זו מאתחלת את חלון המשחק - מגדירה את גודל החלון ושם החלון

שם	משמעות
def build(self):	Method creates the game window

מחלקת הלוח

מחלקת הלוח אחראית על המשחק עצמו. ברגע שנפתח המסך מתחיל המשחק. המחלקה אחראית על חוקים ואירועים הקשורים לפעילות המיידית של המשחק

תכונות המחלקה:

שם	משמעות
cols	Number of columns for kivy graphics to work
curr_b	Currently delt with button

curr_p	Currently delt with piece
move_to	Array of cells piece can move to
first_click	(boolean) is it the first click (selecting piece or moving it)
human	Color of human’s pieces
comp	Color of computer’s pieces
board	2d array describing logical side of the board
turn	Who’s turn it is (1 or -1: black or white)
neutral_moves	Variable counting neutral moves that lead to a draw
showing_modal	(boolean) is a popup currently up on the screen
vis_b	2d array of tiles describing the visual side of the board

פעולות המחלקה:

שם	משמעות
def __init__(self):	Method initializes the board, the pieces and all relevant variables.
def click(self, btn):	Method selects or moves pieces according to: first or second click, legal moves, game status (over or not) and so on… and changes state of board accordingly
def comp_move(self):	Method preforms the computes move including search for best move and making changes to the board
def make_move(self, b, mv, f_mv=False):	Method selects cells that the player can move to by placing a grey dot or coloring capturable pieces grey
def promote(self, btn):	Method promotes piece (logically and visually)
# AI related methods	
def minimax(self, game_state, depth):	Method finds best move possible for computer on received depth and board

def min_play(self, game_state, depth, alpha, beta):	Method finds best score for human player to be played inside the search for the best computer move
def max_play(self, game_state, depth, alpha, beta):	Method finds the best computer move to play inside the minimax algorithm
def evaluate(self, b):	Method evaluates received board state and gives it a score for computer to decide which move is best later
# legal and end related methods	
def is_legal(self, b, move):	Method determines move legality (doesn't cause immediate mate to self, not a castling move if castling side is under check)
def is_check(self, b, color):	Method determines if player of received color is under check threat
def over_state(self, b, color):	Method returns the received board end status for received side (0 = not over, 1 = draw, 2 = mate, 3 = stalemate)
# visual methods	
def show_go(self, event):	Method opens a game over popup

def show_promote(self, event):	Method opens a promotion popup
def highlight_cells(self, color):	Method highlights cells that player can move to and colors the moving piece red
def apply_move(self, mv, f_mv=False):	Method applies received move on the visual board.
def restart(self, btn):	Method restarts the game
Def quit(self, btn):	Method quits the game.

מחלקת משבצת

הלוח הוויזואלי שמשתמשים בו במחלקת לוח המשחק בנוי ממשבצות (tiles) לכל משבצת יש תכונות ייחודיות לה. מחלקת המשבצת נועדה ליצור ולשלוט על משבצות אלה.

תכונות המחלקה:

שם	משמעות
j	Who’s turn it is (1 or -1: black or white)
value	Value of the cell (0 = empty, 1 = pawn, 2 = knight, 3 = bishop, 4 = rook, 5 = queen, 6 = king). Positive numbers – black, negative – white
keep_ratio	Visual property of an image so image wont be distorted
allow_stretch	Visual property of an image so image wont be distorted
Background_normal	Image of the cell (either empty or image of a piece)

פעולות המחלקה:

שם	משמעות
def __init__(self, i, j, value **kwargs):	Method creates a tile according to received values
def set_value(self, new_value):	method sets new value to a cell

קובץ utils.py

בקובץ זה נמצאות כל הפעולות שלא קשורות ישירות ללוח המשחק, למשבצת או לחלונות. אלה הן פעולות עזר המקובצות לצורכי סידור והגיון בקובץ נפרד. קובץ זה יכיל פעולות כמו get_score_value() אשר מקבלת שם של כלי ומחזירה את הערכו הנקודתי לצורכי הערכת לוח המשחק.

תכונות הקובץ:

תכונות הקובץ יהיו ה-piece-square tables אשר משמות את המחשב לתת הערכה לכלי לפי מיקומו על הלוח. הטבלאות נכונות לכלים לבנים, על מנת לקבל את הערך הנכון עבור כלים שחורים האינדקס של הנקודה יספר מהסוף (len(pst) - idx)

שם	משמעות
pawn	Piece-square table for white pawn
knight	Piece-square table for white knight
bishop	Piece-square table for white bishop
rook	Piece-square table for white rook
queen	Piece-square table for white queen
king_midgame	Piece-square table for white king before endgame
king_endgame	Piece-square table for white king during endgame

פעולות הקובץ:

שם	משמעות
def get_type(val):	Method returns the type of received piece (separates from the move counter)
def is_same_color(val1, val2):	Method determines if 2 received pieces are of the same color
def name_to_val(str):	Method returns the integer type of by a piece's name
def is_valid(i, j):	Method determines if received coordinate values are in the 8 by 8 board
def get_new_val(val):	Method adds to the move counter of a piece
def get_score_value(val):	Method returns the score value of a piece
def is_castle_move(fj, tj):	Method determines of a move is a castling move (method is called only if a king is moving and if he moves 2 squares it's a castling move)
def get_go_message(event, player):	Method returns a game over message depending on the type of event and the player's color
# board comparison / evaluation	

def translate_fen(fen):	Method translates a FEN board representation into a 2d int numpy array
def get_pst_val(val, i, j, endgame):	Method returns the score value of a piece by examining it's Piece-Square Table
def is_endgame(b):	Method determines if we're in the endgame now
def get_moves(b, i, j):	Method returns the move of a piece in received board in received coordinates
# getting a piece's moves	
def get_moves(b, i, j):	Method returns the moves of any piece at received coordinates
def get_all_moves(b, i, j):	Method returns the moves of all moves of received side
def get_pawn_moves(b, i, j):	Method returns the moves of a pawn in received coordinates
def get_knight_moves(b, i, j):	Method returns the moves of a knight in received coordinates
def get_bishop_moves(b, i, j):	Method returns the moves of a bishop in received coordinates

def get_rook_moves(b, i, j):	Method returns the moves of a rook in received coordinates
def get_queen_moves(b, i, j):	Method returns the moves of a queen in received coordinates
def get_king_moves(b, i, j):	Method returns the moves of a king in received coordinates

תיאור האלגוריתם:

אלגוריתם minimax

אלגוריתם ה-minimax הוא עץ אשר פורס את כל המהלכים האפשריים של המחשב ואת כל התגובות האפשריות של השחקן האנושי על פי עומק נתון. המחשב בודק את כל עץ האפשרויות, ולאחר מכן המחשב בוחר את המהלך הכי טוב עבורו מכל העץ בעזרת פונקציית הערכה (evaluate) שנותנת ניקוד מספרי לכל פוזיציה. בנוסף לאלגוריתם השתמשתי בגיזום האלפא-בטא שחוסך למחשב בדיקת אופציות שהם בוודאות פחות טובות עבור המחשב ובכך המחשב לא צריך לעבור על כך אץ האפשרויות שיכול להיות מאוד גדול במשחק כמו שחמט ומתקצר זמן ריצת פונקציית ה-minimax משמעותית.

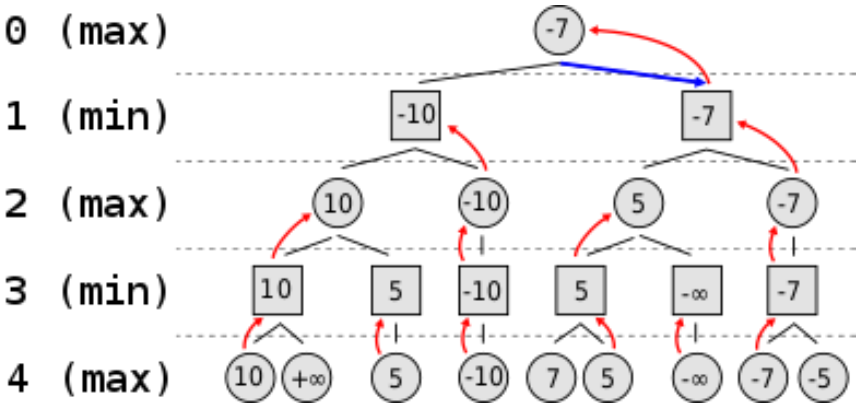
4 פוקנציות יוצרות יחד את הפלא שהוא אלגוריתם ה-minimax.

- def minimax()
- def min_play()
- def max_play()
- def evaluate()

כשמגיע תור המחשב, התוכנית קוראת לפונקציית minimax, בה נמצאים כל המהלכים האפשריים. המחשב מריץ כל אחד מהמהלכים ושולח את הלוח החדש לפונקציית min_play בה נעשה אותו תהליך עם הכלים של השחקן האנושי (סימולציה של מה השחקן היה בוחר).

תהליך זה של "מסירות" בתורות נמשך לפי עומק האלגוריתם (depth) אשר נקבע מראש (בפרויקט שלי - 3). באמצעות פונקציית evaluate הוא מחליט איזה ניקוד לתת למצב הנוכחי של הלוח בכך אחד מהשלבים כמו שמתואר בתרשים (אם הניקוד חיובי אז המצב טוב למחשב ואם הניקוד שלילי אז המצב טוב לשחקן האנושי).

בתרשים המחשב בחר "במהלך הימני" מכיוון שהתוצאה פחות שלילית אם יתבצע המהלך הזה. בשורות ה-max, המחשב בוחר מתוך האפשרויות בשורה הבאה את אופציה הכי גדולה (לכיוון +אינסוף). בשורות ה-min, בוחר השחקן האנושי (המדומה) את מהלך שיביא לניקוד הכי נמוך בשורה הבאה.

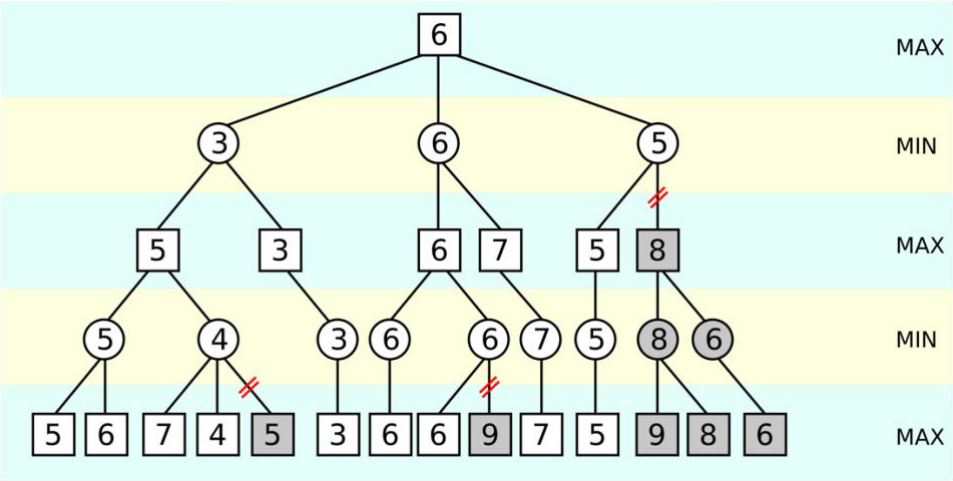


הניקוד של כל אחת מהמשבצות בתרשים נקבע לפי העדיפות בשורה הבאה. לדוגמה: בשורה

3 במשבצת הימינית נבחר -7 כי השחקן min (אנושי), יבחר בהכרח באופציה הקטנה יותר. המחשב עושה את החשיבה לאחור הזו לכל אחד מהמהלכים עד שמחליט איזה מהם יביא לניקוד הכי גבוהה

גיזום אלפא-בטא (Alpha-beta pruning)

אלגוריתם ה-minimax הוא כלי מאוד חזק כשמדובר בבינה מלאכותית אך יש לו בעיה אחת גדולה. זמן הריצה של האלגוריתם ארוך מאוד. זה למה השתמשתי בגיזום אלפא-בטא. כאשר משתמשים בגיזום אלפא-בטא אופציות שכבר בוודאות קיבלו ניקוד נמוך יותר לא נבדקות. כלומר, נחסך זמן הריצה של פונקציית minimax והמחשב מבצע החלטות יותר מהר. אלגוריתם זה משתמש בשני משתנים, אלפא ובטא. אלפא הוא התוצאה המינימלית המובטחת למחשב (בעל הניקוד החיובי) ובטא מייצג את התוצאה המינימלית המובטחת לשחקן האנושי (בעל הניקוד השלילי). אלפא ובטא מאותחלים למינוס אינסוף ואינסוף בהתאם (המצב הגרוע ביותר עבור השחקנים). עם התקדמות המשחק, ערכי אלפא ובטא משתנים. ברגע שבטא מקבל ערך נמוך מאלפא, ניתן לדעת בוודאות כי העמדה הנוכחית אינה מייצגת משחק אופטימלי עבוד אחד מהשחקנים ולכן ניתן לפסול אותה.



הערכה בפעולת ה-evaluate

הערכה טובה בפעולת ה-evaluate מאוד חשובה למחשב חכם. קראתי וחקרתי שיטות הערכה שונות רבות אך בסוף החלטתי להשתמש בשיטה המתאורת במאמר של Tomasz Michniewski, Simplified Evaluation Function.

במאמר זה מתאר המחבר שיטה פשוטה יחסית ליצירת פעולות הערכה לשחמט. ההערכה במאמרו מחולקת לשני חלקים:

- הערכת כלים
- טבלאות כלי-משבצת (Piece-Square Tables)

על מנת לתת לכל כלי הערכה לחשיבותו ראשית יש להעמיד כמה תנאים

```
B > N > 3P
B + N = R + 1.5P
Q + P = 2R
```

- התנאי הראשון אומר שרץ שווה יותר מאשר פרש ושניהם שווים יותר משלושה חיילים.
- התנאי השני אומר שרץ ופרש שווים ביחד יותר מצריח וחייל אחד. לכן נקבע שרץ ופרש שווים לצריח אחד וחייל וחצי.

מכן מגיעה שיטת הניקוד הבאה

```
P = 100
N = 320
B = 330
R = 500
Q = 900
K = 20000
```

כעת נעבור לטבלאות כלי-משבצת.

כאשר כלי נמצא במרכז הלוח יש לו יותר מהלכים והוא מגן על יותר כלים. כמו כן, חיילים הופכים למלכות כאשר מגיעים לסוף הלוח. מכאן נשאלת השאלה איך אפשר לגרום לכלים "להבין" שעליהם לפעול בצורה שתקדם את מצבם ותגביר את חשיבותם ומשקלם במשחק. אלגוריתם ה-minimax הוא אלגוריתם מצוין אך במשחק כמו שחמט, הגעה לעומק שבו המחשב ידע רק באמצעות השימוש בהסתכלות קדימה מה עליו לעשות הוא תהליך ארוך מאוד. על מנת לשמור על זמן תגובה סביר של המחשב משתמשים בטבלאות כלי-משבצת אשר מאפשרות לרמות את התהליך ומקצרות את תהליך מציאת מהלך טוב.

לכל זוג של כלי יש טבלה ייחודית משל עצמו (על מנת לתאים את הטבלה לצבע הכלי, נהפוך את הטבלה – הספירה תתבצע מהסוף להתחלה במקום מהתחלה לסוף).

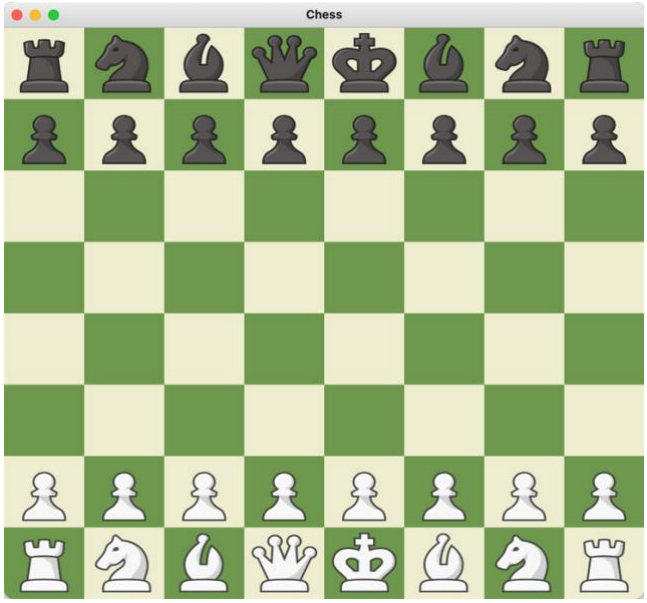
להלן דוגמה של טבלת כלי-משבצת של חייל לבן:

```
0,  0,  0,  0,  0,  0,  0,  0,
50, 50, 50, 50, 50, 50, 50, 50,
10, 10, 20, 30, 30, 20, 10, 10,
 5,  5, 10, 25, 25, 10,  5,  5,
 0,  0,  0, 20, 20,  0,  0,  0,
 5, -5,-10,  0,  0,-10, -5,  5,
 5, 10, 10,-20,-20, 10, 10,  5,
 0,  0,  0,  0,  0,  0,  0,  0
```

המידע על הטבלאות הללו נשמר במערך numpy חד מימדי.

שיטת רישום FEN

שיטת רישות fen ניתן לקחת כל לוח שחמט קיים ותאר אותו באמצעות משתנה sting אחד.
דבר זה מאפשר לי לקחת כל פוזיציה של פאזל ולשחזר אותה



"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq"

- כל "/" הוא מעבר שורה. כלומר, השורה הראשונה מתוארת ע"י rnbqkbnr
- כל אחד מסוגי הכלים מצוין ע"י אות אחת (p = pawn, n = knight, b = bishop, r = rook, q = queen, k = king)
- הכלים הלבנים מצוינים באות גדולה והכלים השחורים באות קטנה.
- האות הבודדת לאחד כתיבת הלוח מצינת את התור נוכחי (w או b). במצב זה השחקן הלבן מתחיל.
- לאחד ציון התור, מופיע ציון ההצרחות האפשריות:
  - K = king side castle (for white player)
  - Q = queen side castle (for white player)
  - k = king side castle (for black player)
  - q = queen side castle (for black player)

אם לא מופיעה אחת מהאותיות, אי אפשר לבצע את אותה הצרחה.

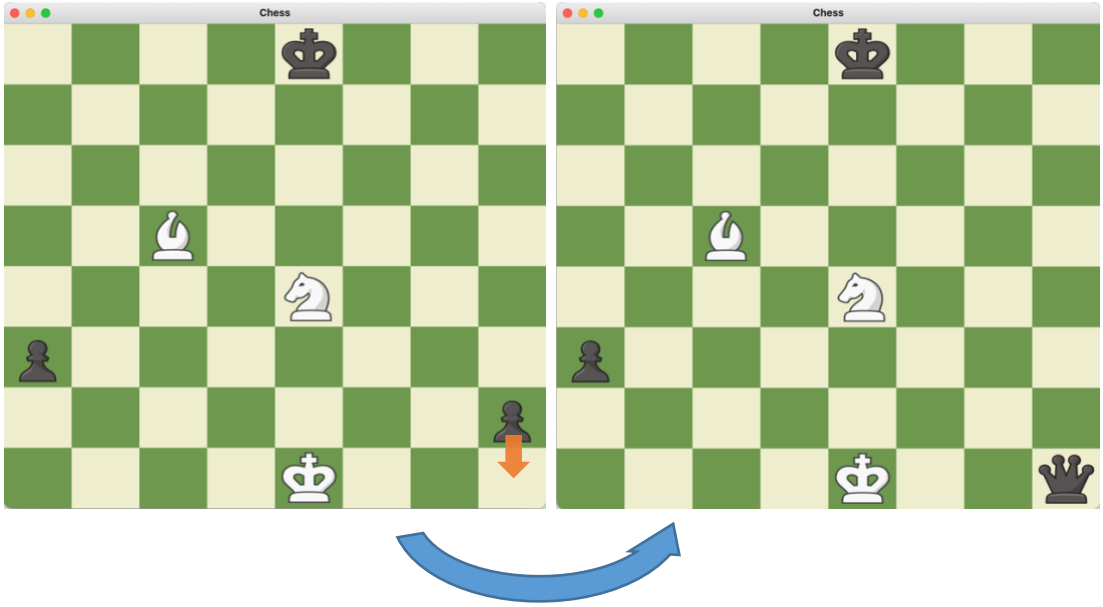
הדגמת הבינה במשחק

להלן דוגמאות המראות את יכולת המחשב (כלים שחורים).

לאורך כל הדוגמאות אשתמש בשיטת מספור המשבצות המתבסס על אותיות באנגלית ומספרים. האותיות מייצגות את התורים משמאל לימין והמספרים מייצגים את השורות מלטה למעלה.

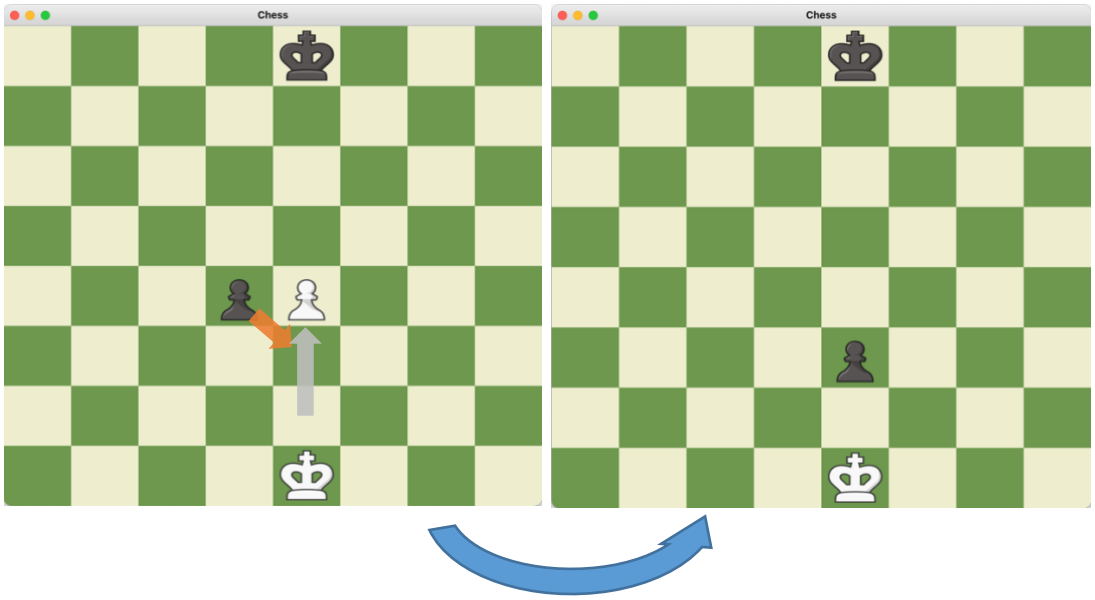
כל הדוגמאות ההללו הועלו לתוך המשחק באמצעות שיטת הרישום FEN שתוארה לפניכן.

בדוגמא 1 ניתן לראות שהמחשב בחר לאכול את החייל הלבן ב-d6 אשר מאיים על המלכה ב-c7 לא עם המלכה אלא אם החייל ב-e7. כך המלכה לא מסתכנת מהחייל הלבן ב-e5 שיכול לאכול אותה אם היא עצמה הייתה אוכלת את החייל הלבן ב-d6.



בדוגמה זו בתורו של המחשב, הוא ראה שאם יזיז את החייל ב-h2 ל-h1 הוא יהפוך למלכה בגלל שמלכה הרבה יותר חזקה מחייל הניקוד שמתקבל מהקידום יותר גבוה מכל ניקוד אחר שהמחשב היה יכול לקבל

בדוגמה זו שני השחקנים הגיעו לסיום הפתיחה. כלומר, יש עימות על האמצע, הפרשים ורצים נפתחו והשחקנים מוכנים להצרחה. במצב זה נהוג לעשות הצרחה כדי להגן על המלך לפני שפותחים בהתקפה שתסכן את אותו.



בדוגמה זו בתורו של המחשב החייל ב-d4 רואה שהוא יכול לאכול את החייל הלבן ב-e4 ניתן לעשות זאת לפי חוק "En passant". חוק זה אומר שאם חייל היריב עשה בתורו הראשון מהלך של שתי משבצות, ניתן לאכול אותו בצורה שנראית בדוגמה. אם החייל היא עובר את שתי המשבצות הללו בשתי מהלכים נפרדים החוק לא היה תקף.

מדריך למשתמש

הוראות התקנה

כדי לשחק במשחק יש להוריד את הקבצים והתוכנות הבאות:

תוכנות וספריות:

- Python 3.7
- Kivy 2.0.0
- numpy

קבצי קוד:

- Chess.kv
- Main.py
- Utils.py

תמונות:

- White_green_board.png
- Chess_rules_hewbrew.png
- Grey_circle.png
- **Pieces (folder)**
  - 0.png
  - 1.png
  - 2.png
  - 3.png
  - 4.png
  - 5.png
  - 6.png
  - -1.png
  - -2.png
  - -3.png
  - -4.png

- הוראות המשחק (יש להקפיד על התאמה בין חוקי /הוראות המשחק שאתם מעלים לבין הפרויקטים שלכם!)

~ חוקי המשחק ~

הלוח והכלים

בשחמט משתמשים בלוח שחור לבן (או כל שני צבעים אחרים שיתבלטו אחד מהשני) כמתואר באיור 1. על הלוח יונחו הכלים (שחורים ולבנים). משבצות הלוח מתוארות על ידי מספר המציין את מיקומם "לגובה" (1-8) ובאות המציינת את מיקומם "לרוחב" (א-ח או H-A). בשורות הקרובות לקצוות הלוח (שורה 8 ו-1) ימוקמו הכלים שווי הערך (צריחים, פרשים, רצים, המלך והמלכה). ובשורה השנייה מהקצה ימוקמו החיילים הרגליים.

(איור 1)

רגלי (Pawn)

החייל הרגלי הוא הכלי הכי פשוט ו**בעל הערך הנקודתי הכי נמוך בחישוב מצב הלוח (1)**.
אם זהו תורו הראשון הוא יכול להתקדם 2 משבצות קדימה (בתנאי שלא עומד שם כלי אחר).
נוסף על כך, בכל תור יכול הרגלי ללכת משבצת אחת קדימה (בתנאי שלא עומד שם כלי) ולאכול כלי עוין באלכסון קדימה למשבצת אחת.

אם רגלי מגיע לקצה שנגדי של הלוח הוא יכול להפוך לרץ, צריח, פרש או מלכה ולנוע לפי חוקי תנועתם. (ניקודו גם עולה לניקוד הכלי אליו הוא הופך).

נוסף על כך רגלי אחד התקדם בשתי משבצות בתורו הראשון, רגלי אחר יכול לאכול אותו כאילו התקדם משבצת אחת. בכך הרגלי האוכל עובר את הרגלי הנאכל ועומד מאחוריו. מהלך זה נקרא "הכאה דרך הילוכו" או

" en passant"

צריח (Rook)

הצריח נע בקווים ישרים (במקביל לאחד צדי הלוח). הוא לא מוגבל למשבצת אחת ויכול לנוע לאן שירצה השחקן כל עוד זה במקביל לאחד צדי הלוח ולא עומד שם כלי מאותו הצבע של הצריח.

ערכו שנקודתי הוא 5.

פרש (Knight)

לפרש תנועה מיוחדת, הוא נע שתי משבצות לכיוון מסוים ומשבצת אחת לכיוון המאונך לו. הפרש בתנועתו יכול "לקפץ" מעל כלים אחרים ויכול לנוע למשבצת מסוימת אם לא תופס אותה כלי מאותו הצבע.

ערכו הנקודתי הוא 3.

רץ (Bishop)

הרץ נע בצורה דומה לצריח אך באלכסונים. **ערכו הנקודתי הוא 3.**

מלכה (Queen)

המלכה היא הכלי החזק ביותר על הלוח. היא יכולה לנוע בצירוף יכולות התנועה של הרץ ושל הצריח **וערכה הנקודתי הוא 9.**

מלך (King)

המלך הוא הכלי החשוב ביותר במשחק. אם השחק מכריז על שחמט (המלך לא ניתן להצלה) המשחק נגמר והשחקן שהכריז על שחמט מנצח. המלך יכול לנוע ישר או באלכסון בכל כיוון למשבצת אחת בלבד אלא אם כן הדבר מעמיד אותו בסכנה להיאכל או שעומד שם כלי מאותו הצבע. נוסף על כך, המלך יכול לבצע מהלך שנקרא "הצרחה".

הצרחה

במהלך זה, המלך והצריח זזים ממקומם. מהלך זה יכול להתבצע אך ורק אם המלך והצריח אתו ההצרחה נעשית לא זזו מתחילת המשחק ואין בינם כלים. אם נעשית הצרחה קצרה (עם הצריח הקרוב יותר למלך) המלך יעבור למשבצת הנמצאת ישר משמאלו של הצריח והצריח יעבור למשבצת משמאל למיקום החדש של המלך. אם נעשית הצרחה ארוכה (עם הצריח הרחוק מהמלך), המלך יעבור למשבצת שנמצאת שתי משבצות מימין לצריח הצריח יעבור למשבצת מימין למיקומו החדש של מלך. ערכו הנקודתי של המלך הוא 2.5 נקודות.

חוקים עבור כל הכלים הסובבים את ההגנה המלך

לא ניתן לעשות מהלך אף אחד מהכלים אשר יפתח את המלך לסכנה מידית מכיוון שאי אפשר לוותר על המלך. דבר זה מוביל לכך שכלים אשר מגנים על המלך יכולים "להיתקע" במקום מכיוון שאינם יכולים לזוז. (שחמט)

מהלך המשחק

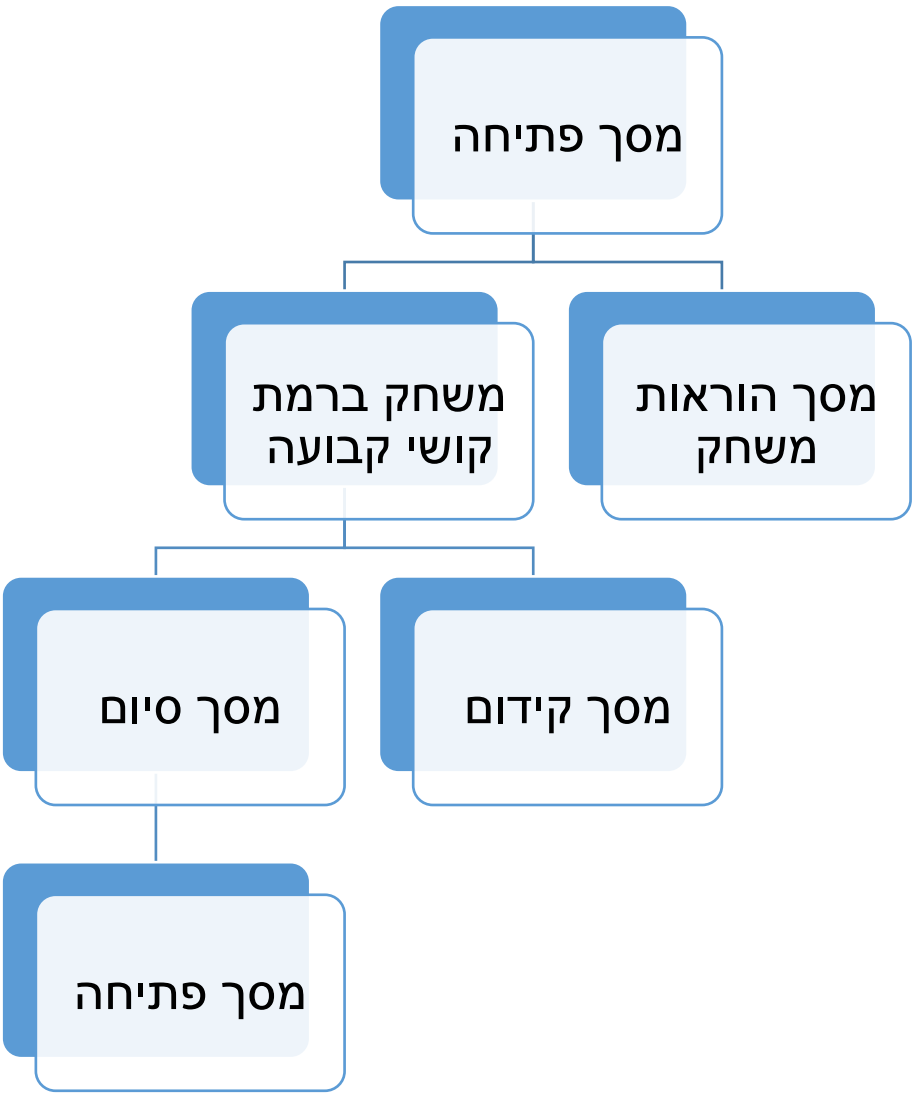
השחקן המחזיק בכלים הלבנים הוא זה שיתחיל את המשחק. כל שחקן בתורו עושה מהלך עם אחד הכלים שלו לפי המהלכים האפשריים לכל אחד מן הסוגים השונים של הכלים (כמפורט בהמשך). שחקן יכול להניע כלי אחד בלבד בכל תור (חוץ מאשר בהצרכה –

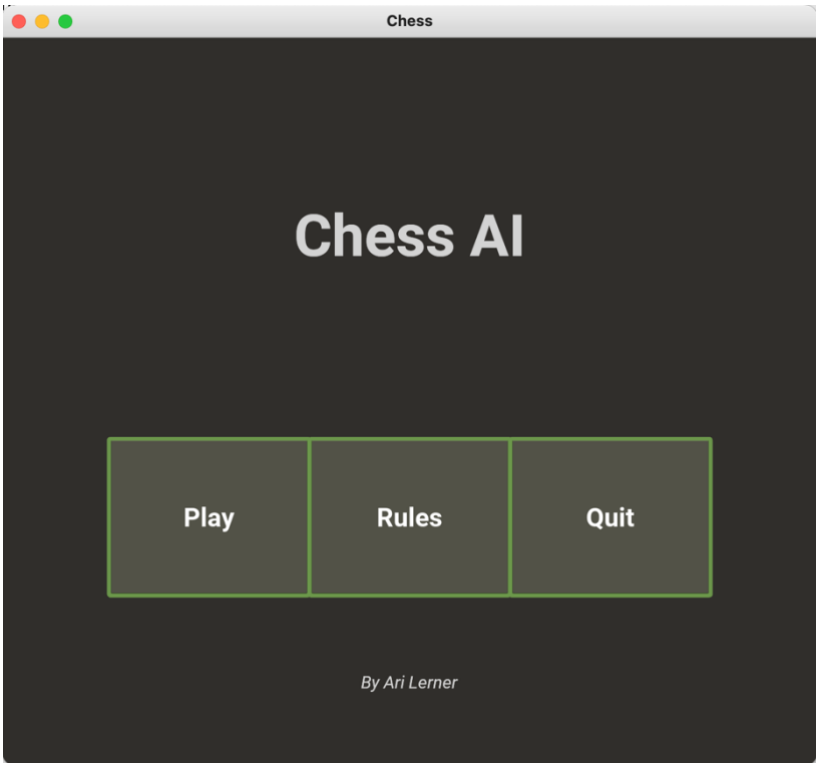
כמפורט לעיל). המהלך חייב להיות "חוקי" משמע לא יביא למצב של מט בתרו הבא. הכלים יכולים לזוז למשבצות פנויות לפי חוקי תנועתם או "לאכול" כלי של השחקן היריב.

סיום המשחק

1. שחמט – כאשר יש איום של שח על המלך ואין שום מהלך שהיריב יכול לבצע כדי להציל את המלך. במקרה זה השחקן שהכריז שחמט על היריב ינצח ללא חשיבות בנקודות של החשקנים
2. פט – אין שום מהלכים ששחקן יכול לבצע אך אין איום שח על המלך. במקרה כזה נקבע בין השחקנים תיקו.
3. הכרעה על ידי זמן – כאשר נגמר הזמן המוקצב לאחד השחקנים נגמר השחקן מפסיד.

תרשים מסכי המשחק

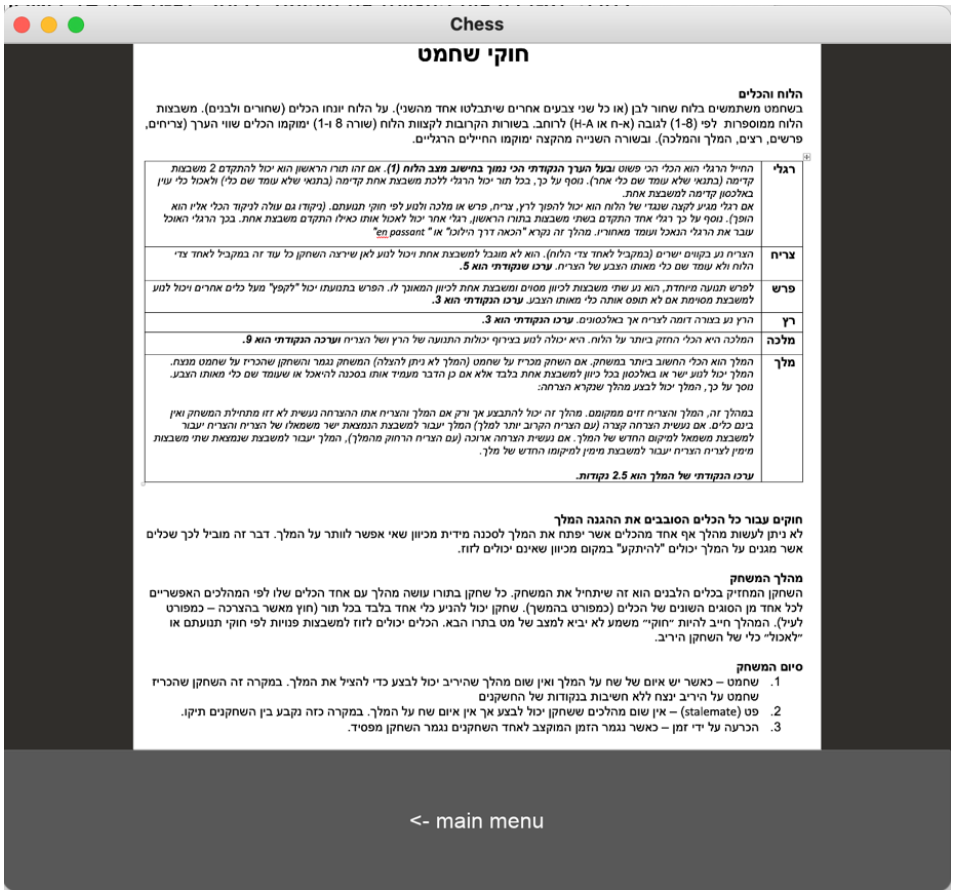




מסך הפתיחה

מסך זה פוגש את המשתמש כשהוא מפעיל את המשחק. הוא מציג את שמו של המשחק, שם היוצר ומכיל שלושה כפתורים:

- **כפתור "Play"** אשר מפעיל את המשחק.
- **כפתור "Rules"** אשר מעביר את המשתמש למסך החוקים.
- **כפתור "Quit"** אשר יוצא מן המשחק.



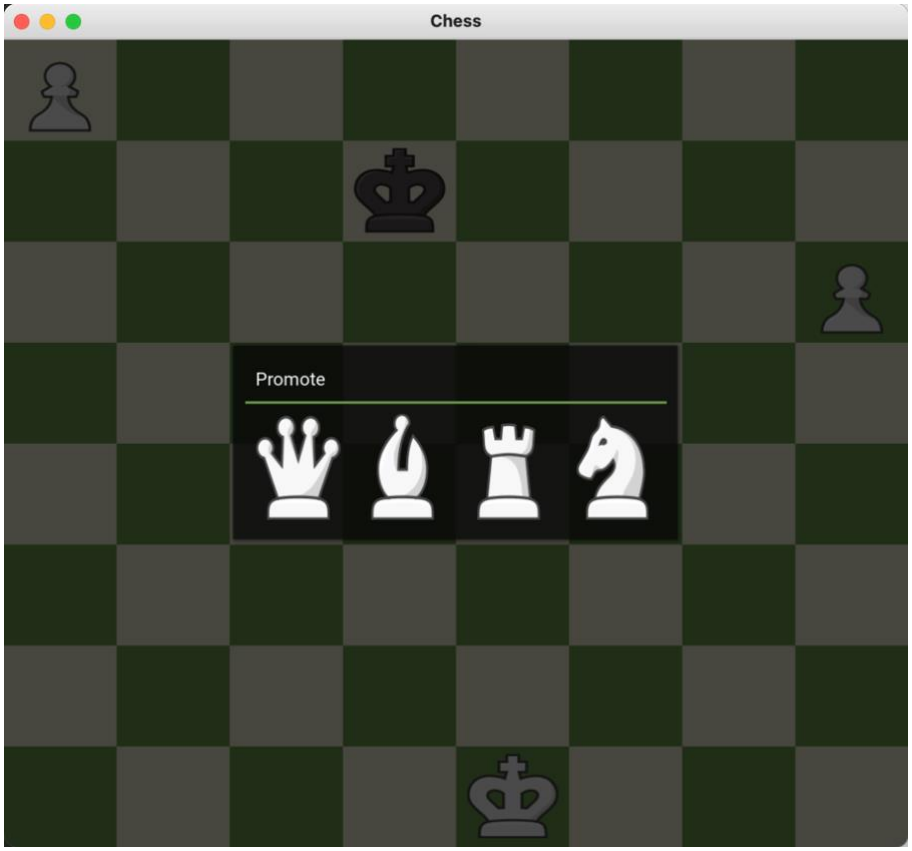
מסך הוראות המשחק

מסך זה מציג למשתמש את חוקי המשחק.

המסך מכיל כפתור אחד אשר שולח את המשתמש חזרה למסך הראשי (מסך הפתיחה)

משחק ברמת קושי קבועה

מסך זה מכיל את המשחק עצמו. כל משבצת בלוח היא כפתור וניתן לבחור את חייליים הלבנים ולהזיז אותם. באיור התחתון המלכה במשבצת a7 נלחצה והיא יכולה לזוז למשבצות המסומנות או לאכול את הכלים המסומנים (נצבעים בצבע אפור בהיר יותר).

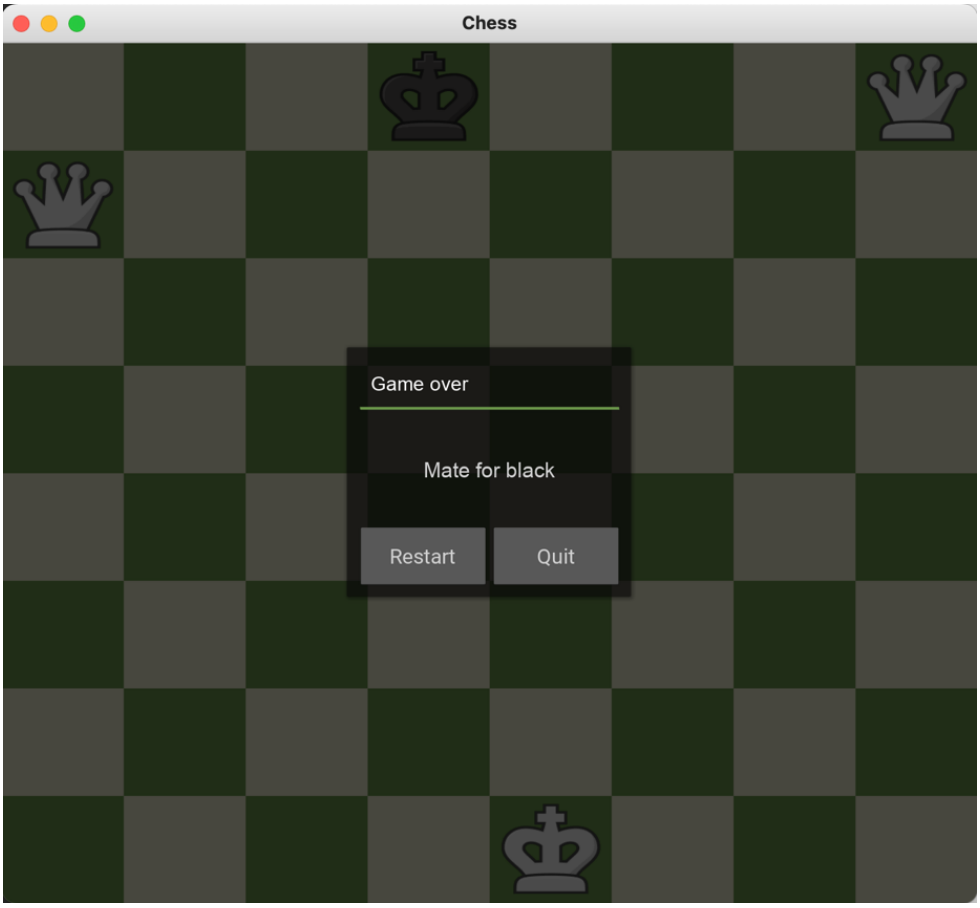


מסך קידום

מסך הקידום נפתח ברגע שחייל של השחקן האנושי מגיע לשורה האחרונה (שורה 8). המסך מאפשר לקדם את החייל לאחת מ4 האפשרויות:

- מלכה
- רץ
- צריח
- פרש

כל אחת מתמונות הכלים היא כפתור אשר ברגע לחיצה עליו מקודם החייל עבורו נפתח המסך לאותו הכלי



מסך סיום

מסך הסיום נפתח ברגע שהמשחק נגמר. הוא מכיל שלושה אלמנטים:

- טקסט סיום
  - Mate
  - Stalemate
  - Draw

כל אחד מהאפשריות הללו (חוץ מאשר תיקו) מלווה בשם השחקן שביצע את הפעולה הביאה ליום המשחק (White / Black)

- כפתור אתחול המשחק. הכפתור מוגר את המשחק הקיים ומתחיל אחד חדש. המשתמש מגיע למסך הפתיחה
- כפתור יציאה מהמשחק. כפתור זה סוגר את התוכנה

מדריך למפתח

מבנה הקבצים בפרויקט הוא:

- Chess.kv (מבנה וגרפי של מסכים)
- Main.py (קובץ תוכנה ראשי)
- Utils.py (קובץ תוכנה משני – מכיל בעיקר פעולות עזר)
- White_green_board.png (תמונת לוח המשחק)
- Chess_rules_hewbrew.png (תמונת חוקי המשחק)
- Grey_circle.png (נקודה אפורה לסימון משבצות אליהן אפשר לעבור)
- Pieces (folder)
    - 0.png
    - 1.png
    - 2.png
    - 3.png
    - 4.png
    - 5.png
    - 6.png
    - -1.png
    - -2.png
    - -3.png
    - -4.png
    - -5.png
    - -6.png

קוד המשחק מצורף בסוף תיק הפרוייקט

רפלקציה

מאוד נהנתי מהעבודה על הפרויקט. הוא היה מאתגר והכין הרבה מכשולים בדרך. לטעמי כמות זמן שניתנה לסיום הפרוייקט עלתה על הדרוש והייתי יכול לעשות את כל הפרויקט בהרבה פחות זמן אך החופש למשוך את התהליך ליותר מחצי שנה גרם לי לסיים את הפרויקט קרוב לתאריך הסיום.

למדתי הרבה מאוד על המשחק כגון שיטות רישום כמו FEN המשמשות לתאר את כל הלוח ב-string אחד. ומתוך עניין החלטתי להתעמק ולקרוא על תחומים כגון neural networks ולמדתי להשתמש ב-git. נוסף לכך כמוגן שהעמקתי את הידע שלי ב-python וב-kivy.

היו הרבה מאוד דברים שלא הייתי חייב בשביל פרוייקט טוב אבל אלה היו דברים שאני שמח שלמדתי והתעמקתי. להמשכך דרכי אקח את הידיעה שעל מנת לעשות את הפרוייקט הכי טוב שאני יכול כדי לפעמים להתעמק וללמוד דברים בסיסיים יותר שמרחיבות לא רק את הידע שלי אלא גם את היכולות שלי בהרבה.

מכיוון ששחמט הוא משחק מאוד מסובך חשוב מאוד שתהיה פונקצית evaluate טובה אשר תוכל לקרוא את הלוח בצורה המייטבית ביותר. היו לי קשיים רבים בפיתוח פונקציה זו והיא עברה הרבה מאוד שינויים. נוסף על כך בתחילת הפיתוח שיניתי את מבנה הנתונים שלי פעמים רבות ואף פעם לא הייתי מרוצה, מה שבסופו של דבר איפשר לפרויקט שלי להיות כה קצר (ולעניין).

מסקנותיי מהתהליך הן שיש חשיבות מאוד גדולה לתכנון טוב של מבנה הנתונים. במהלך הפיתוח שיניתי את מבנה הנתונים שלי מספר פעמים כדי לאפשר פעילות יותר אפקטיבית של התוכנה (וכדי לקצר את מפר השורות...). נוסף על כך, גדלה הערכתי לחשיבות המחקר בקשר למשחק. אני למדתי המון גם על טכנולוגיות שבסופו של דבר לא נכנסו לשימוש בפרויקט כגון neural networks, אלגוריטמים להערכת לוח (פונקציית evaluate) ושיטות רישום כמו FEN.

אם הייתי עושה מחדש את הפרויקט הייתי חושב המון, משרבט ומשרטט אלמנטים שונים של המשחק לפני תחילת כתיבת הקוד. שלב התכנון הוא שלב מאוד חשוב בפיתוח תוכנה.

על מנת לאפשר עבודה יותר יעילה הייתי מעדיף לעבוד יותר מהבית מאשר בכיתה כדי לאפשר נוחות. כשאני מתכנת בבית בין אם זה במסגרת בית הספר או מחוצה לה, אני מאוד אוהב לעבוד מהבית כי זה מאפשר לי להתרכז יותר, אני מכיר יותר את המחשב והציוד שאני משתמש בו ואני יכול לשתות קפה. למרות זאת, לא תמיד תהיה לי האפשרות לעבוד בתנאים אידיאליים שכאלה, אז אני הייתי רוצה לפתח שיטת עבודה שתאפשר לי לדמות כל אזור לנוחות שאני מרגיש בבית.

ביבליוגרפיה

Michniewski, T. (n.d.). *Simplified Evaluation Function.* Retrieved from Chess Programming Wiki: https://www.chessprogramming.org/Simplified_Evaluation_Function

Chess.com. (n.d.). *Board & pieces assets.*

שחמט. (n.d.). Retrieved from ויקיפדיה: https://he.wikipedia.org/wiki/שחמט

קובץ main.py

```
import numpy as np import numpy as np
from kivy.app import App
from kivy.core.window import Window
from kivy.lang import Builder
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.button import Button
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.popup import Popup
from kivy.uix.screenmanager import ScreenManager, Screen

from utils import *


class Tile(Button):
    def __init__(self, i, j, val, **kwargs):
        # method initializes a tile object
        Button.__init__(self, **kwargs)
        self.i = i  # row / y coordinate
        self.j = j  # col / x coordinate
        self.value = val
        self.keep_ratio = True  # kivy visual property
        self.allow_stretch = False  # kivy visual
property
        self.background_normal = "pieces/" +
str(get_type(val)) + ".png"

    def set_value(self, new_val):
        # method sets new value to a tile
        self.value = new_val
        self.background_normal = "pieces/" +
str(get_type(self.value)) + ".png"


class Board(GridLayout, Screen):
    def __init__(self, **kwargs):
        # method initializes the board object
        GridLayout.__init__(self)
        self.cols = 8  # for building kivy grid
        self.curr_b = None  # pressed piece button
        self.curr_p = None  # pressed piece object
        self.move_to = []  # list of options to move
selected piece
        self.first_click = True  # True - selecting
piece, False - moving selected piece
        self.human = -1  # human player
        self.comp = 1  # computer player
```

```
        start_b =
"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR b KQkq - 0
1"
        self.board = np.array(translate_fen(start_b))   #
the logical side of the board (2d array of strings)
        self.depth = 3  # depth to which computer goes
each move
        self.turn = 1  # who's turn it is
        if start_b[start_b.index(" ") + 1] == "w":
            self.turn = -1
        self.neutral_moves = 0  # for 50 moves rule
        self.showing_modal = False  # if a modal opens
changes to true
        vis_b_temp = []  # 2d array of buttons (later
made into numpy array)
        for i in range(self.cols):
            line_list = []
            for j in range(self.cols):
                cell = Tile(i, j, self.board[i][j])
                cell.bind(on_press=self.click)
                self.add_widget(cell)
                line_list.append(cell)
            vis_b_temp.append(line_list)
        self.vis_b = np.array(vis_b_temp)  # visual
representation of the chess board
        if self.turn == self.comp:
            self.comp_move()

    #  immediate move methods
    def click(self, btn):
        # method reacts to clicking on a cell
        if self.turn == self.human:
            over = self.over_state(self.board, self.turn)
# checking if game is over
            if self.first_click and btn.value != 0 and
(btn.value / abs(btn.value)) == self.turn:  # if
selecting click
                if over == 0:  # if not over
                    self.curr_b = btn  # remembering
which button was pressed
                    self.curr_p = (btn.i, btn.j)  #
remembering which piece is moving
                    self.move_list =
get_moves(self.board, btn.i, btn.j)  # array of piece's
possible moves
                    self.highlight_cells(True)  #
highlighting cells that player can move to
                    self.first_click = False

            elif not self.first_click and
(self.curr_b.value / abs(self.curr_b.value)) ==
```

```
self.turn:
                self.highlight_cells(False)  #
deselecting cells
                move = (self.curr_p[0], self.curr_p[1],
btn.i, btn.j)
                if self.move_list.count(move) != 0 and
self.is_legal(self.board, move):
                    self.make_move(self.board, move,
True)  # if move is legal, do it
                    if not self.showing_modal:  # if not
promoting
                        self.turn = self.comp
                        self.comp_move()
                if not self.showing_modal:
                    self.first_click = True
            if over != 0:  # if game is over
                self.playing = False
                self.show_go(over)

    def comp_move(self):
        # method preforms the computer's move
        over = self.over_state(self.board, self.turn)  #
checking over state
        if over == 0:
            saved_board = self.board.copy()  # copying
the board so it won't change
            move = self.minimax(saved_board, self.depth)
# generate smart move
            self.board = saved_board.copy()  # updating
the board
            if self.is_legal(self.board, move):  # if
move is legal, do it
                self.curr_b =
self.vis_b[move[0]][move[1]]
                self.make_move(self.board, move, True)
            self.turn = self.human
            over = self.over_state(self.board, self.turn)
        if over != 0:  # if game is over
            self.playing = False
            self.show_go(over)

    def make_move(self, b, mv, f_mv=False):
        # method preforms a move on the logical side of a
board
        val_to = b[mv[2]][mv[3]]  # value of cell that
piece is moving to
        val = b[mv[0]][mv[1]]  # moving piece's value

        # 50 moves rule
        if f_mv:  # if final move (not directly from
inside the minimax algorithm)
```

```
            if 10 <= val < 20 or val_to != 0:
                self.neutral_moves = 0  # if pawn-move or
piece captured, counter is reset

        # en passant - if moving piece is a pawn and mt
square is free and a first move pawn is standing to the
side
        if get_type(val) == 1 and val_to == 0 and mv[3]
!= mv[1] and abs(b[mv[0]][mv[3]]) == 11:
            i = mv[2] - 1
            if self.turn == -1:
                i = mv[2] + 1
            b[i][mv[3]] = 0
            if f_mv:
                self.vis_b[i][mv[3]].set_value(0)

        # castle
        if abs(val) == 60 and (mv[3] == 6 or mv[3] == 2)
and mv[1] == 4:
            cords = (0, 3)  # long castle coordinate
            if mv[3] == 6:
                cords = (7, 5)  # short castle
coordinates
            b[mv[0]][cords[1]] = 41 * self.turn  # moving
castling rook
            b[mv[0]][cords[0]] = 0  # emptying rook's
past cell
            self.apply_move([mv[2], cords[0], mv[2],
cords[1]], f_mv)  # graphical application of move

        b[mv[2]][mv[3]] = get_new_val(val)  # moving the
piece
        b[mv[0]][mv[1]] = 0  # emptying piece's past cell
        self.apply_move(mv, f_mv)  # moving piece to a
new location, leaving empty space

        # promotion
        if get_type(val) == 1 and (mv[2] == 0 or mv[2] ==
7):
            if self.turn == self.human and f_mv:
                self.show_promote()  # only human player
needs to see promotion popup
            else:
                b[mv[2]][mv[3]] = 52 * self.turn  #
automatically setting the piece to a black queen
                if f_mv:  # if needs to move visuals

self.vis_b[mv[2]][mv[3]].set_value(52)

    def promote(self, btn):
        # method promotes received piece
```

```
        new_val = (int(btn.background_normal[8]) * 10 +
2) * self.turn  # extracting value from image's file name

self.vis_b[self.curr_b.i][self.curr_b.j].set_value(new_va
l)  # applying change to visual board
        self.board[self.curr_b.i][self.curr_b.j] =
new_val  # applying change to logical board
        self.popup.dismiss()  # closing popup
        self.first_click = True
        self.showing_modal = False
        self.turn = self.comp
        self.comp_move()

    #  ai methods
    def minimax(self, b, depth):
        # method receives a description of the current
state of the board and a depth to go to
        alpha = float('-inf')
        beta = float('inf')
        moves = get_all_moves(b, self.comp)  # getting
all possible moves in situation
        best_move = moves[0]
        best_score = float('-inf')
        for move in moves:  # go over all possible moves
            if self.is_legal(b, move):
                curr_b = b.copy()  # copying the board to
make changes to it
                self.make_move(curr_b, move)
                score = self.min_play(curr_b, depth - 1,
alpha, beta)  # maybe add "human" parameter
                if score > best_score:
                    best_move = move  # remembering best
move
                    best_score = score  # remembering
best score
                if alpha < best_score:
                    alpha = best_score
                if beta <= alpha:
                    break
        return best_move

    def min_play(self, b, depth, alpha, beta):
        # method plays out the best move for human
        if depth == 0 or self.over_state(b, self.human)
!= 0:  # if game is over or depth 0 is reached
            return self.evaluate(b) * (depth + 1)
        moves = get_all_moves(b, self.human)
        best_score = float('inf')
        for move in moves:  # go over all moves
            if self.is_legal(b, move):
                curr_b = b.copy()  # copy board
```

```
                self.make_move(curr_b, move)  # applying
logical move
                score = self.max_play(curr_b, depth - 1,
alpha, beta)
                if score < best_score:
                    best_move = move  # remembering best
move
                    best_score = score  # remembering
best score
                if beta > best_score:
                    beta = best_score
                if beta <= alpha:
                    break
        return best_score

    def max_play(self, b, depth, alpha, beta):
        # method plays out best move for computer
        if depth == 0 or self.over_state(b, self.comp) !=
0:  # if game is over or depth 0 is reached
            return self.evaluate(b) * (depth + 1)
        moves = get_all_moves(b, self.comp)
        best_score = float('-inf')
        for move in moves:  # go over all moves
            if self.is_legal(b, move):
                curr_b = b.copy()  # copy board
                self.make_move(curr_b, move)  # apply
logical move
                score = self.min_play(curr_b, depth - 1,
alpha, beta)
                if score > best_score:
                    best_move = move  # remembering best
move
                    best_score = score  # remembering
best score
                if alpha < best_score:
                    alpha = best_score
                if beta <= alpha:
                    break
        return best_score

    def evaluate(self, b):
        # method evaluates the board and returns an
integer score
        score = 0
        for i in range(8):
            for j in range(8):
                val = b[i][j]
                if val != 0:
                    is_end = False
                    if get_type(val) == 6:  # check if
it's endgame only if king is related
```

```
                        is_end = is_endgame(b)
                    pst_score =
get_pst_val(get_type(val), i, j, is_end) * 2  # gets
piece's pst value
                    piece_score = get_score_value(val) +
pst_score  # get piece's type value
                    if is_end:
                        human_over_Stat =
self.over_state(b, self.human)
                        if human_over_Stat == 2:  #
promote mate
                            score += 5000
                        elif human_over_Stat == 3:  #
avoid stalemate
                            score -= 2000
                    if val < 0:
                        score -= piece_score
                    else:
                        score += piece_score
        return score

    #  end related checking
    def is_legal(self, b, move):
        # method checks legality of move
        bool = False
        val = b[move[0]][move[1]]  # moving piece
        val_to = b[move[2]][move[3]]  # cell to move to
        color = val / abs(val)  # 1 if black, -1 if white
        if abs(val) == 60 and self.is_check(b, color) and
is_castle_move(move[1], move[3]):
            return False  # castle move when under a
check threat is illegal
        b[move[2]][move[3]] = val
        b[move[0]][move[1]] = 0
        if not self.is_check(b, color):  # move is legal
if it doesn't lead directly to a mate
            bool = True
        b[move[0]][move[1]] = val
        b[move[2]][move[3]] = val_to
        return bool

    def is_check(self, b, color):
        # method receives board and player and checks if
opponent is threatening check
        all_moves = get_all_moves(b, color * -1)  #
opponent's all possible moves
        for move in all_moves:
            if 60 <= abs(b[move[2]][move[3]]) < 70:  # if
any piece can capture the king
                return True
        return False
```

```
    def over_state(self, b, color):
        # method returns if received player has lost or
not (or stale mate)
        # 0 = not over, 1 = draw, 2 = mate, 3 = stalemate
        if self.neutral_moves == 20:  # 50 moves rule
(reduced to 20)
            return 1
        all_moves = get_all_moves(b, color)
        if len(all_moves) != 0:
            for move in all_moves:
                if move and self.is_legal(b, move):
                    return 0  # if possible move found
            if self.is_check(b, color):  # if no moves
and under check
                return 2
        return 3  # no moves and not under check

    #  visual
    def show_go(self, event):
        # method opens a game over popup
        message = get_go_message(event, self.turn)
        self.showing_modal = True
        main_container =
BoxLayout(orientation='vertical', size=(self.width,
self.height))  # general box
        go_label = Label(text=message, font_size=30,
font_name='Arial', color=(.83, .83, .83, 1))  # game over
message
        main_container.add_widget(go_label)
        btn_container =
BoxLayout(orientation='horizontal', size_hint_y=.5,
spacing=10)  # box containing the buttons
        restart_btn = Button(text='Restart', color=(.83,
.83, .83, 1))  # restart button
        restart_btn.bind(on_press=self.restart)

        quit_btn = Button(text='Quit', color=(.83, .83,
.83, 1))  # quit button
        quit_btn.bind(on_press=self.quit)
        btn_container.add_widget(restart_btn)
        btn_container.add_widget(quit_btn)
        main_container.add_widget(btn_container)

        self.popup = Popup(title="Game over",
title_font="Arial", size_hint=(.3, .3),
auto_dismiss=False,
                           separator_color=(.46, .59,
.33, 1), background_color=(.19, .18, .17, .7),
                           content=main_container)
        self.popup.open()
```

```
    def show_promote(self):
        # method opens promotion popup (queen/ bishop /
rook / knight)
        self.showing_modal = True
        box = BoxLayout(orientation="horizontal")
        queen_btn = Button(background_normal="pieces/-
5.png")  # promote to queen button
        queen_btn.bind(on_press=self.promote)
        bishop_btn = Button(background_normal="pieces/-
3.png")  # promote to queen button
        bishop_btn.bind(on_press=self.promote)
        rook_btn = Button(background_normal="pieces/-
4.png")  # promote to rook button
        rook_btn.bind(on_press=self.promote)
        knight_btn = Button(background_normal="pieces/-
2.png")  # promote to rook button
        knight_btn.bind(on_press=self.promote)
        box.add_widget(queen_btn)
        box.add_widget(bishop_btn)
        box.add_widget(rook_btn)
        box.add_widget(knight_btn)
        self.popup = Popup(title="Promote", content=box,
size_hint=(.5, .25), separator_color=(.46, .59, .33, 1),
                           background_color=(.19, .18,
.17, .8), auto_dismiss=False)
        self.popup.open()

    def highlight_cells(self, on):
        # method highlighting cells that a piece can move
to
        for idx in range(len(self.move_list)):
            if self.is_legal(self.board,
self.move_list[idx]):
                i = self.move_list[idx][2]
                j = self.move_list[idx][3]
                if on:  # if method called to highlight
cells
                    self.curr_b.background_color = (.8,
.2, .2, 1)
                    if self.board[i][j] == 0:

self.vis_b[i][j].background_normal = "grey_circle.png"
                    else:
                        self.vis_b[i][j].background_color
= (.38, .38, .25, .5)
                else:  # if method called to de-highlight
cells
                    self.curr_b.background_color = [1, 1,
1, 1]
                    if self.board[i][j] == 0:
```

```
self.vis_b[i][j].background_normal = 'pieces/0.png'
                    else:
                        self.vis_b[i][j].background_color
= [1, 1, 1, 1]

    def apply_move(self, mv, f_mv=False):
        # method applies visual changes of a move to the
board
        if f_mv:  # if needs to move visuals

self.vis_b[mv[2]][mv[3]].set_value(self.vis_b[mv[0]][mv[1
]].value)
            self.vis_b[mv[0]][mv[1]].set_value(0)
            self.curr_b = self.vis_b[mv[2]][mv[3]]

    # essentials
    def restart(self, btn):
        # method restarts the game and send user to start
screen
        Chess.get_running_app().stop()
        Chess().run()

    def quit(self, btn):
        # method quits the game
        Chess.get_running_app().stop()


class StartWindow(Screen):
    pass


class RulesWindow(Screen):
    pass


class WindowManager(ScreenManager):
    pass


kv = Builder.load_file('chess.kv')


class Chess(App):
    def build(self):
        Window.clearcolor = (.19, .18, .17, 1)
        Window.size = (700, 625)
        self.title = 'Chess'


Chess().run()
```

קובץ utils.py

```
# --- Piece-Square Tables --- used to give pieces a bonus
for moving in a certain way and penalties for moving in
# other ways (for evaluation method) tables are taken
from Tomasz Michniewski's article "Simplified Evaluation
# Function" (the tables are not mirrored for black)

pawn = (70, 70, 70, 70, 70, 70, 70, 70,
        50, 50, 50, 50, 50, 50, 50, 50,
        10, 10, 20, 30, 30, 20, 10, 10,
        5, 5, 10, 25, 25, 10, 5, 5,
        0, 0, 0, 20, 20, 0, 0, 0,
        5, -5, -10, 0, 0, -10, -5, 5,
        5, 10, 10, -20, -20, 10, 10, 5,
        0, 0, 0, 0, 0, 0, 0, 0)
knight = (-50, -40, -30, -30, -30, -30, -40, -50,
          -40, -20, 0, 0, 0, 0, -20, -40,
          -30, 0, 10, 15, 15, 10, 0, -30,
          -30, 5, 15, 20, 20, 15, 5, -30,
          -30, 0, 15, 20, 20, 15, 0, -30,
          -30, 5, 10, 15, 15, 10, 5, -30,
          -40, -20, 0, 5, 5, 0, -20, -40,
          -50, -40, -30, -30, -30, -30, -40, -50)
bishop = (-20, -10, -10, -10, -10, -10, -10, -20,
          -10, 0, 0, 0, 0, 0, 0, -10,
          -10, 0, 5, 10, 10, 5, 0, -10,
          -10, 5, 5, 10, 10, 5, 5, -10,
          -10, 0, 10, 10, 10, 10, 0, -10,
          -10, 10, 10, 10, 10, 10, 10, -10,
          -10, 5, 0, 0, 0, 0, 5, -10,
          -20, -10, -10, -10, -10, -10, -10, -20)
rook = (0, 0, 0, 0, 0, 0, 0, 0,
        5, 10, 10, 10, 10, 10, 10, 5,
        -5, 0, 0, 0, 0, 0, 0, -5,
        -5, 0, 0, 0, 0, 0, 0, -5,
        -5, 0, 0, 0, 0, 0, 0, -5,
        -5, 0, 0, 0, 0, 0, 0, -5,
        -5, 0, 0, 0, 0, 0, 0, -5,
        0, 0, 0, 5, 5, 0, 0, 0)
queen = (-20, -10, -10, -5, -5, -10, -10, -20,
         -10, 0, 0, 0, 0, 0, 0, -10,
         -10, 0, 5, 5, 5, 5, 0, -10,
         -5, 0, 5, 5, 5, 5, 0, -5,
         0, 0, 5, 5, 5, 5, 0, -5,
         -10, 5, 5, 5, 5, 5, 0, -10,
         -10, 0, 5, 0, 0, 0, 0, -10,
         -20, -10, -10, -5, -5, -10, -10, -20)
king_midgame = (-30, -40, -40, -50, -50, -40, -40, -30,
                -30, -40, -40, -50, -50, -40, -40, -30,
                -30, -40, -40, -50, -50, -40, -40, -30,
```

```
                 -30, -40, -40, -50, -50, -40, -40, -30,
                 -20, -30, -30, -40, -40, -30, -30, -20,
                 -10, -20, -20, -20, -20, -20, -20, -10,
                 20, 20, 0, 0, 0, 0, 20, 20,
                 20, 30, 10, 0, 0, 10, 30, 20)
king_endgame = (-50, -40, -30, -20, -20, -30, -40, -50,
                 -30, -20, -10, 0, 0, -10, -20, -30,
                 -30, -10, 20, 30, 30, 20, -10, -30,
                 -30, -10, 30, 40, 40, 30, -10, -30,
                 -30, -10, 30, 40, 40, 30, -10, -30,
                 -30, -10, 20, 30, 30, 20, -10, -30,
                 -30, -30, 0, 0, 0, 0, -30, -30,
                 -50, -30, -30, -30, -30, -30, -30, -50)


# simple utility functions (shortens the code)
def get_type(val):
    # method returns the type of received value
    if val != 0:
        return int(divmod(abs(val), 10)[0] * (val /
abs(val)))
    return 0


def is_same_color(val1, val2):
    # method checks if 2 values are of same color
    if (val1 > 0 and val2 > 0) or (val1 < 0 and val2 <
0):
        return True
    return False


def name_to_val(str):
    # method returns an integer value of a piece by name
    if str == "p":
        return 1
    elif str == "n":
        return 2
    elif str == "b":
        return 3
    elif str == "r":
        return 4
    elif str == "q":
        return 5
    elif str == "k":
        return 6
    return 0


def is_valid(i, j):
    # method returns True if the pos is in the board and
```

```
False otherwise
    return 0 <= i < 8 and 0 <= j < 8


def get_new_val(val):
    # method adds to the move count of a piece
    if divmod(abs(val), 10)[1] != 2:
        if val > 0:
            return val + 1
        return val - 1
    return val



def get_score_value(val):
    # method returns score value for each peace type
    score_values = (100, 320, 330, 500, 900, 20000)
    return score_values[abs(get_type(val)) - 1]


def is_castle_move(fj, tj):
    # method determines if a move is a castling move
    return fj == 4 and (tj == 6 or tj == 2)


def get_go_message(event, player):
    # method returns a game over message
    message_str = ""
    if event == 1:
        return "Draw"
    elif event == 2:
        message_str = "Mate for "
    elif event == 3:
        message_str = "Stalemate for "
    if player == 1:
        return message_str + "black"
    return message_str + "white"


# --- board comparison / evaluation ---
def translate_fen(fen):
    # method receives a fen string and translates it into
an actual board setting
    b = [[], [], [], [], [], [], [], []]
    row = 0
    i = 0
    while fen[i] != " ":  # translating the overall board
        if row == 8:
            break
        if fen[i].isdigit():
            for add_empty in range(int(fen[i])):
```

```
                b[row].append(0)
            elif fen[i] == "/" or fen[i] == " ":
                row += 1
            else:
                if fen[i].islower():
                    b[row].append(name_to_val(fen[i]) * 10)
                else:
                    b[row].append(name_to_val(fen[i].lower())
* (-10))
            i += 1
        i += 1
        i += 2
        # making all castling impossible unless said so in
FEN
        if b[0][0] == 40:  # br0
            b[0][0] = 41
        if b[0][7] == 40:  # br0
            b[0][7] = 41
        if b[7][0] == -40:  # wr0
            b[7][0] = -41
        if b[7][7] == -40:  # wr0
            b[7][7] = -41
        while fen[i] != " ":  # correcting board for castling
            if fen[i] == "K" and b[7][7] == -41:
                b[7][7] = -40
            elif fen[i] == "Q" and b[7][0] == -41:
                b[7][0] = -40
            elif fen[i] == "k" and b[0][7] == 41:
                b[0][7] = 40
            elif fen[i] == "q" and b[0][0] == 41:
                b[0][0] = 40
            i += 1
        return b


def get_pst_val(val, i, j, endgame):
    # method receives type of piece and game phase (if
endgame) returns correct piece-value table
    if val < 0:
        idx = i * 8 + j
    else:
        idx = 63 - (i * 8 + j)  # count from end
(reverses the pst for black)
    val = abs(val)
    if val == 1:
        return pawn[idx]
    elif val == 2:
        return knight[idx]
    elif val == 3:
        return bishop[idx]
    elif val == 4:
```

```
        return rook[idx]
    elif val == 5:
        return queen[idx]
    elif val == 6:
        if endgame:
            return king_endgame[idx]
        return king_midgame[idx]


def is_endgame(b):
    # if 3 or less pieces remain (except pawns and kings)
return true
    w_counter = 0
    b_counter = 0
    for i in range(8):
        for j in range(8):
            val = abs(get_type(b[i][j]))
            if val != 1 and val != 6:
                if b[i][j] < 0:
                    w_counter += 1
                else:
                    b_counter += 1
    return w_counter <= 3 or b_counter <= 3


# --- getting moves ---
def get_moves(b, i, j):
    # get moves for all pieces apart from pawns and the
kings
    moves = []
    val = divmod(abs(b[i][j]), 10)[0]
    if val == 1:
        moves = get_pawn_moves(b, i, j)
    elif val == 2:
        moves = get_knight_moves(i, j)
    elif val == 3:
        moves = get_bishop_moves(b, i, j)
    elif val == 4:
        moves = get_rook_moves(b, i, j)
    elif val == 5:
        moves = get_queen_moves(b, i, j)
    elif val == 6:
        moves = get_king_moves(b, i, j)
    new_moves = []

    for move in moves:
        if is_valid(move[2], move[3]):
            if not is_same_color(b[move[2]][move[3]],
b[i][j]):
                new_moves.append(move)
    return new_moves
```

```
def get_all_moves(b, color):
    # method returns all of the moves of the pieces in
received board of received color
    moves = []
    for i in range(8):
        for j in range(8):
            if is_same_color(b[i][j], color):  # if piece
is of received color
                moves.extend(get_moves(b, i, j))  # add
to array of all moves
    return moves


def get_pawn_moves(b, i, j):
    # method receives board and the pos of a pawn and
returns all it's possible moves
    moves = []
    diff = 1
    opponent = -1
    val = b[i][j]
    if val < 0:
        diff = -1
        opponent = 1

    # move 1, 2 forward
    if is_valid(i + diff, j) and b[i + diff][j] == 0:
        moves.append((i, j, i + diff, j))  # move 1
forward
    if ((i == 6 and val < 0) or (i == 1 and val > 0)) and
b[i + diff][j] == 0 and b[i + (diff * 2)][j] == 0:
        moves.append((i, j, i + (diff * 2), j))  # move 2
forward
    # diagonal capture
    for col in range(j - 1, j + 2, 2):
        if is_valid(i + diff, col) and is_same_color(b[i
+ diff][col], opponent):
            moves.append((i, j, i + diff, col))

    # en passant
    if (val < 0 and i == 3) or (val > 0 and i == 4):  #
if white on row 3 or black on row 4
        for col in range(j - 1, j + 2, 2):
            if is_valid(i, col) and b[i][col] == 11 *
opponent and (
                    (is_valid(i + diff, col) and b[i +
diff][col] == 0) or not is_valid(i + diff, col)):
                moves.append((i, j, i + diff, col))  #
left en passant
    return moves
```

```
def get_knight_moves(i, j):
    # method returns all possible moves for a knight at
received position
    # validity is checked in general "get_moves" method
    moves = [(i, j, i + 2, j + 1), (i, j, i + 2, j - 1),
(i, j, i - 2, j + 1), (i, j, i - 2, j - 1),
             (i, j, i + 1, j + 2), (i, j, i - 1, j + 2),
(i, j, i + 1, j - 2), (i, j, i - 1, j - 2)]
    return moves


def get_bishop_moves(b, i, j):
    # method returns all possible moves of bishop in
received board at received position
    moves = []
    row = i
    col = j
    instructions = ((1, 1), (-1, -1), (1, -1), (-1, 1))
    # going in all 4 directions until end of board or
piece is reached
    for direction in range(4):
        while is_valid(row, col) and (b[row][col] == 0 or
(row == i and col == j)):
            row += instructions[direction][0]
            col += instructions[direction][1]
            moves.append((i, j, row, col))
        row = i
        col = j
    return moves


def get_rook_moves(b, i, j):
    # method gets all possible moves of rook in received
board at received position
    moves = []
    row = i
    col = j
    instructions = ((1, 0), (-1, 0), (0, 1), (0, -1))
    # going in all 4 directions until end of board or
piece is reached
    for direction in range(4):
        while is_valid(row, col) and (b[row][col] == 0 or
(row == i and col == j)):
            row += instructions[direction][0]
            col += instructions[direction][1]
            moves.append((i, j, row, col))
        row = i
        col = j
    return moves
```

```
def get_queen_moves(b, i, j):
    # method returns all possible moves of a queen in
received board at received position
    moves = get_rook_moves(b, i, j)
    bishop_moves = get_bishop_moves(b, i, j)
    moves.extend(bishop_moves)
    return moves


def get_king_moves(b, i, j):
    # method returns all possible moves of king in
received board at received pos
    moves = []
    val = b[i][j]
    # adding moves in all 4 directions
    for row in range(i - 1, i + 2):
        for col in range(j - 1, j + 2):
            if is_valid(row, col) and not
is_same_color(b[row][col], val) and not (row == i and col
== j):
                moves.append((i, j, row, col))
    if divmod(val, 10)[1] == 0 and b[i][0] == (val /
abs(val)) * 40 and b[i][1] == 0 and b[i][2] == 0 and
b[i][3] == 0:
        moves.append((i, j, i, 2))  # long castle
    if divmod(val, 10)[1] == 0 and b[i][7] == (val /
abs(val)) * 40 and b[i][5] == 0 and b[i][6] == 0:
        moves.append((i, j, i, 6))  # short castle
    return moves
```

קובץ chess.kv

```
#:import Factory kivy.factory.Factory
#: import FadeTransition
kivy.uix.screenmanager.FadeTransition
WindowManager:
    StartWindow:
    RulesWindow:
    Board:
<StartWindow>:
    name: 'first'
    canvas.before:
        Color:
            rgba: .19, .18, .17, 1
        Rectangle:
            pos: self.pos
            size: self.size
    BoxLayout:
        orientation: 'vertical'
        size: root.width, root.height
        Label:
            text: 'Chess AI'
            font_size: 100
            color: .83, .83, .83, 1
            size_hint_y: 1.2
            bold: True
        BoxLayout:
            orientation: 'horizontal'
            size_hint_y: 0.5
            spacing: 6
            padding: 10
            Label:
                font_size: 54
                size_hint_x: 0.5
            Button:
                text: 'Play'
                font_size: 45
                background_color: (.93, .93, .82, 1)
                bold: True
                canvas.before:
                    Color:
                        rgba: .46, .59, .33, 1
                    Line:
                        width: 6
                        rectangle: self.x, self.y,
self.width, self.height
                on_release:
                    root.manager.transition =
FadeTransition(clearcolor=(.19, .18, .17, 1),
duration=.35)
                    app.root.current = 'game'
```

```
            Button:
                text: 'Rules'
                font_size: 45
                background_color: (.93, .93, .82, 1)
                bold: True
                canvas.before:
                    Color:
                        rgba: .46, .59, .33, 1
                    Line:
                        width: 6
                        rectangle: self.x, self.y,
self.width, self.height
                on_release:
                    root.manager.transition =
FadeTransition(clearcolor=(.19, .18, .17, 1),
duration=.5)
                    app.root.current = "rules"
            Button:
                text: 'Quit'
                font_size: 45
                background_color: (.93, .93, .82, 1)
                bold: True
                canvas.before:
                    Color:
                        rgba: .46, .59, .33, 1
                    Line:
                        width: 6
                        rectangle: self.x, self.y,
self.width, self.height
                on_release: quit()
            Label:
                font_size: 54
                size_hint_x: 0.5
        Label:
            text: "[i]By Ari Lerner[/i]"
            markup: True
            color: .83, .83, .83, 1
            size_hint_y: 0.5
<RulesWindow>:
    name: "rules"
    canvas.before:
        Color:
            rgba: .19, .18, .17, 1
        Rectangle:
            pos: self.pos
            size: self.size
    BoxLayout:
        orientation: "vertical"
        background_color: rgba(.46, .59, .33, 1)
        Image:
            source: 'chess_rules_hebrew.png'
```

```
            allow_stretch: False
        Button:
            text: '<- main menu'
            font_name: 'Arial'
            font_size: 32
            size_hint_y: 0.2
            on_release:
                root.manager.transition =
FadeTransition(clearcolor=(.19, .18, .17, 1),duration=.5)
                app.root.current = "first"
<Board>:
    name: 'game'
    canvas.before:
        Rectangle:
            pos: self.pos
            size: self.size
            source: 'white_green_board.png'
```