שחמט מרובה משתתפים אונליין
עם מימוש אינטליגנציה
מלאכותית מבוססת גיזום אלפא
בטא בפייתון

שם החלופה: למידת מכונה

שם התלמיד: נדב אבירן

ת"ז: 212938930

שם המנחה: יהודה אור

הגשה תאריך: 10.6.2022

מבוא

הפרויקט שלקחתי על עצמי להתמחות בו הוא משחק השחמט. מכיוון שרציתי אתגר, החלטתי שאני אממש בינה מלאכותית פרטית המבוססת על אלגוריתם דטרמיניסטי בשם מינימקס. במימוש הפרויקט צפיתי מספר אתגרים. הראשון מביניהם הבנת החוקים והכללים של השחמט. ללמוד שחמט זהו דבר לא כל כך פשוט, ולבנות את המשחק מ0 זהו דבר קשה בהרבה יותר. על כך, נברתי ולאחר מאמץ מתועד הסתגלתי אל החוקים של המשחק והטמעתי אותם בקוד. אתגר נוסף שצפיתי הוא בנייה לא יעילה של מחלקות ומעשה במחשבה תחילה. לאחר מימוש המשחק גיליתי שעל מנת לממש את הבינה המלאכותית אצטרך לשנות את הקוד מהשורש ולעצב מחדש את המחלקות כולן. זהו למעשה הדבר שחששתי ממנו הכי הרבה לפני ההתחלה של הכתיבה. אני שמח להגיד שכתיבת הפרויקט הייתה חוויה מלמדת אך אשמור זאת לרפלקציה.

הלקוח במערכת הינו למעשה כל אדם שמעוניין להשתפר בשחמט. המערכת מגיעה להציע אתגר לשחקני שחמט הכמהים להשתפר בצורה שאדם ממוצע לא מאפשר להם.

רוב הפתרונות הקיימים אינם מראים כיצד הקוד בנוי והם שומרים את האלגוריתם בצד השרת. דרך התנהלות זו אינה מאפשרת למידה והבנה של אלגוריתמים מעניינים. אני מאמין ששיטה של קוד פתוח מהווה תחלופה הגונה לקוד סגור.

הפרויקט אינו דורש מפרט טכנולוגי ספיציפי מעבר למחשב עם סביבת פייתון מוגדרת וחיבור לאינטרנט על מנת להתחבר לשרת.

המערכת מאפשרת לאנשים מרחבי העולם לתקשר ולשחק יחד אחד עם השני בזמן אמת. השרת עובד עם אלגוריתם המקשיב למשתתפים ויודע לשלוח מסרים בצורה אסינכרונית. כמו כן התקשורת הינה מוצפנת כך שאף אחד מהצדדים לא יוכל לרמות..

המערכת מאפשרת כניסה עם ממשק משתמש, וידוי סיסמה מוצפנת בעזרת בסיס נתונים, למערכת שני משתמשים אדמין ולו הרשאות מיוחדות וגם משתמש רגיל שמבצע אינטראקציה עם המערכת. המערכת תאפשר למשתמש לגשת למשחק און ליין, למשחק לוקלי על המחשב מול אדם הנמצא ליד, ומשחק כמובן מול בינה מלאכותית.

תיאור יכולות מורחב:

יכולת: בקשה למשחק מקוון

- מהות: הוספת בקשה מהשרת לחיבור למשחק מקוון
- אוסף יכולות נדרשות:
  - ממשק משתמש
  - מסך בקשה
  - העברת מידע לשרת
  - תקשורת אסינכרונית

יכולת: שליחה וקבלה של מידע מהמשחק

- מהות: תקשורת בזמן אמת ובה המהלכים וסטטוס המשחק
- אוסף יכולות נדרשות:
  - משחק שחמט בנוי
  - ניהול רב משתמשים
  - שליחת מידע אל השרת
  - קבלה של מידע מהשרת בתור מתווך
  - טיימר הממשיך לפעול בזמן התקשורת

יכולת: בקשה למשחק מול בינה מלאכותית

- מהות: משחק כנגד אלגוריתם
- אוסף יכולות נדרשות:
  - אלגוריתם
  - מסך תצוגת משחק
  - טיימר פועל
  - חוקי משחק תקינים

תיאור הפרויקט

שפת תוכנה - python

Python היא שפת תכנות מונחה עצמים ברמה גבוהה עם סמנטיקה דינמית משולבת בעיקר לפיתוח אתרים ואפליקציות. היא אטרקטיבית ביותר בתחום פיתוח יישומים מהיר מכיוון שהוא מציע אפשרויות הקלדה דינאמיות וכריכה דינמית.
פייתון היא שפה שנחשבת קריאה ופשוטה להבנה בעבור מפתחים ואף ניתן לתרגם אותה. בזכות זה, עלות התחזוקה והפיתוח של התוכנית יורד משמעותית מכיוון שהוא מאפשר לצוותים לעבוד בשיתוף פעולה ללא מחסומי שפה וניסיון משמעותיים.

סביבת עבודה - pycharm

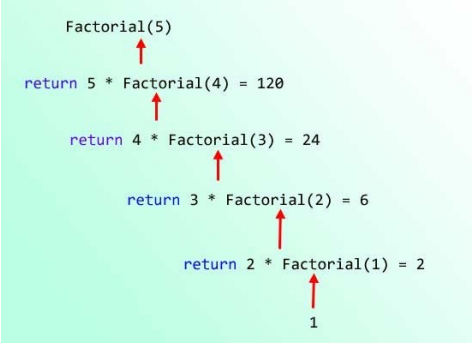
PyCharm היא סביבת עבודה (IDE) המתאימה לכל מערכות ההפעלה. PyCharm היא אחת מסביבות העבודה הטובות ביותר בשביל עבודה ב Python ותומכת בשתי הגרסאות של פייתון. לפייצ'ארם יש מגוון תוספים שמאפשרים עבודה יעילה ונקייה, ערכות וכלים כדי להאיץ את הפיתוח ובו-זמנית למזער את המאמץ הדרוש כדי להשיג את אותו הדבר.
בפייצ'ארם נכתב בציבור בפעם הראשונה בפברואר 2010.

ספרית מנוע משחק - pygame

Pygame היא ספריית פיתון רבת פלטפורמות שנוצרה במיוחד עבור עיצוב משחקי וידאו. היא ספרייה שימושית ביותר המכילה גם גרפיקה, ויזואליה וצלילים שניתן להשתמש בהם כדי לשפר את המשחק המתוכנן. הספריה מכילה ספריות שונות שעובדות עם תמונות וצלילים ויכולות ליצור גרפיקה למשחקים. זה מפשט את כל תהליך עיצוב המשחק ומקל על מפתחי משחקים המתמקדים באלגוריתמיקה.

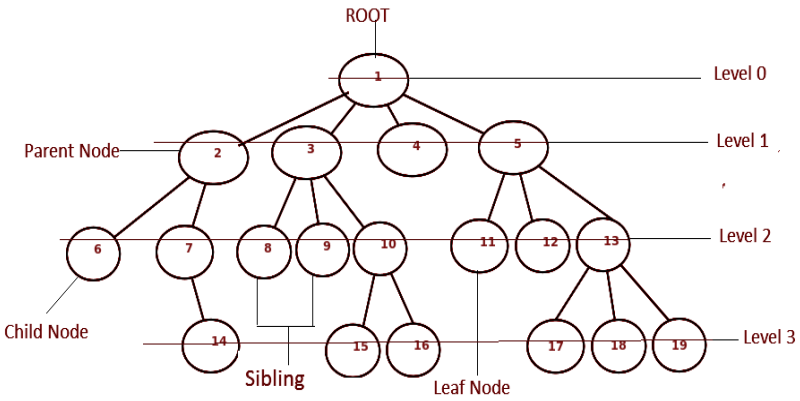
מושגים - ובסיס תיאורטי לאלגוריתם

רקורסיה



התהליך שבו פונקציה קוראת לעצמה באופן ישיר או עקיף נקרא רקורסיה והפונקציה המקבילה נקראת פונקציה רקורסיבית.
באמצעות אלגוריתם רקורסיבי, ניתן לפתור בעיות מסוימות די בקלות. יש לזה יתרונות מסוימים על פני טכניקת האיטרציה. משימה שניתן להגדיר עם תת-משימה דומה לה, רקורסיה היא אחד הפתרונות הטובים ביותר עבורה. לדוגמה; העצרת של מספר.

עצים



עץ הוא מבנה נתונים לא ליניארי והיררכי המורכב מאוסף של צמתים כך שכל צומת של העץ מאחסן ערך, רשימת הפניות לצמתים ("הילדים").
הגדרה רקורסיבית: : עץ מורכב משורש, ומאפס או יותר תת-עצים, כך שיש קצה משורש העץ לשורש של כל תת-עץ.

טרמינולוגיה בסיסית במבנה נתוני עץ:

צומת אב:

הצומת שהוא קודמו של צומת נקרא צומת האב של אותו צומת. {2} הוא צומת האב של {6, 7}.

צומת ילד:

הצומת שהוא היורש המיידי של צומת נקרא צומת הילד של אותו צומת. דוגמאות: {6, 7} הם הצמתים הצאצאים של {2}.

צומת שורש:

הצומת העליון בעץ או הצומת שאין לו שום צומת אב נקרא צומת השורש. {1} הוא צומת השורש של העץ. עץ לא ריק חייב להכיל בדיוק צומת שורש אחד ובדיוק נתיב אחד מהשורש לכל שאר הצמתים של העץ.

דרגת צומת:

הספירה הכוללת של תת-עצים המחוברים לצומת זה נקראת מידת הצומת. דרגת צומת עלים חייבת להיות 0. דרגת עץ היא המדרגה המקסימלית של צומת בין כל הצמתים בעץ.

צומת עלה או צומת חיצוני:

הצמתים שאין להם צמתים צאצאים נקראים צמתים עלים. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} הם צמתי העלים של העץ.

קדמון של צומת:

כל צמתים קודמים בנתיב השורש לצומת זה נקראים קדמונים של אותו צומת. {1, 2} הם צמתי האב של הצומת {7}

צאצא:

כל צומת עוקב בנתיב מצומת העלה לצומת זה. {7, 14} הם צאצאיו של הצומת. {2}.

אחים:

ילדים מאותו צומת הורה נקראים אחים. {8, 9, 10} נקראים אחים.

עומק צומת:

ספירת הקצוות מהשורש לצומת. עומק הצומת {14} הוא 3.

גובה צומת:

מספר הקצוות בנתיב הארוך ביותר מאותו צומת לעלה. גובה הצומת {3} הוא 2.

גובה עץ:

גובה עץ הוא גובה צומת השורש כלומר ספירת הקצוות מהשורש לצומת העמוק ביותר. גובה העץ הנ"ל הוא 3.

רמה של צומת:

ספירת הקצוות בנתיב מצומת השורש לצומת זה. לצומת השורש יש רמה 0.

צומת פנימי:

צומת עם צאצא אחד לפחות נקרא Internal Node.

שכן של צומת:

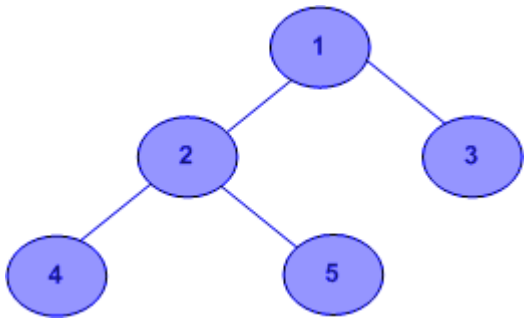
צמתים הורים או ילדים של אותו צומת נקראים שכנים של צומת זה.

תת-עץ:

כל צומת של העץ יחד עם צאצאיו

סריקת postorder

בניגוד למבני נתונים ליניאריים (מערך, רשימה מקושרת, תורים, ערימות וכו') שיש להם רק דרך הגיונית אחת לחצות אותם, ניתן לחצות עצים בדרכים שונות. להלן הדרכים המקובלות למעבר עצים



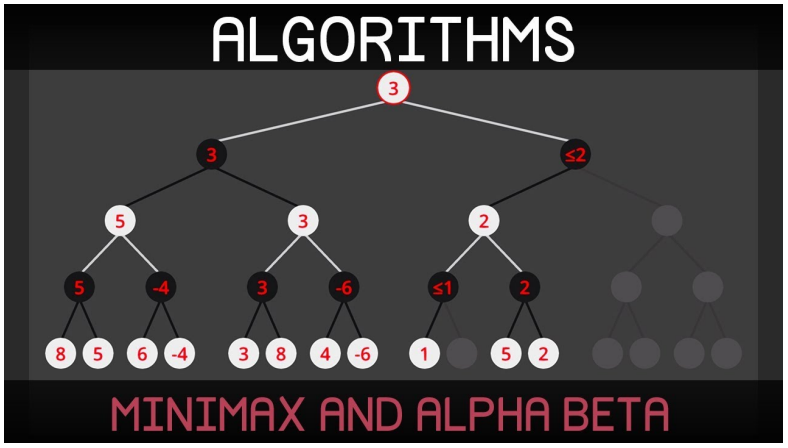
דוגמאות לסוגי סריקות:

(1) Inorder (Left, Root, Right) : 4 2 5 1 3

(2) Preorder (Root, Left, Right) : 1 2 4 5 3

(3) Postorder (Left, Right, Root) : 4 5 2 3 1

האלגוריתם - minimax:



האלגוריתם מינימקס הוא אלגוריתם בתורת המשחקים שמבוסס על סריקה לאחור של עץ המורכב מסיטואציות קבועות בתוך משחק שמבוסס על תורות.
האלגוריתם עובד בצורה הבאה:

- דבר ראשון מייצרים עץ שמורכב שממצבים במשחק.
לדוגמה, כאשר משחקים שחמט יש מספר מוגבל של מהלכים הניתן לבצע. עבור כל מהלך שניתן לבצע יש אחריו כמות מוגבלת של מהלכים שהשחקן האחר יכול לבצע.
- בבניית העץ נוצרים צאצאים אשר בעצם הופכים לצמתי ולתת עצים עד לעומק אשר הגענו לעומק המקסימלי שהגדרנו.

- לאחר בניית העץ מבצעים קריאה ראשונית לאלגוריתם. מטרתו של האלגוריתם היא לבצע את ההחלטה הטובה ביותר בעבור השחקן הנוכחי בקריאה הראשונית.

- עבור כל צומת בעץ נקרא לאלגוריתם מחדש ונבצע סריקת postorder על העץ על מנת להתחיל מהעלים של העץ. האלגוריתם יחזיר את הניקוד רק כאשר הוא בשיא העומק בעץ. עד אז הוא יקרא לעצמו רקורסיבית בעבור כל מהלך פוטנציאלי

- כאשר האלגוריתם הגיע לעלי העץ הוא מחשב את הפוזיציה ומחזיר אותה לפונקציה ממנה הוא נקרא.
- הייחודיות של האלגוריתם מתבטאת במשתנה בוליאני הקובע את התור. בעזרתו, כל רמה בעץ מסמלת בעצם או את התור של השחקן הראשון או השני.
- כאשר נקרא האלגוריתם, השחקן הראשון בעצם "מנסה" למקסם את נקודותיו והשחקן השני מנסה להפחיתן.

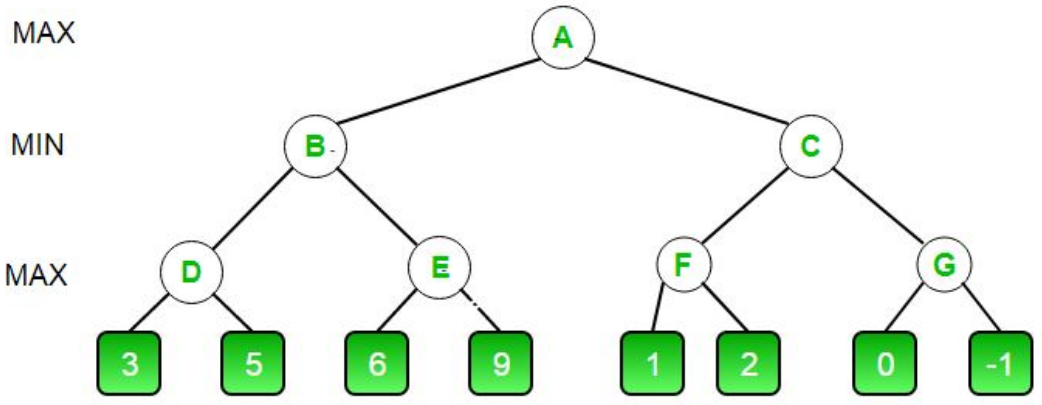
- כאשר הרמה בעץ שייכת לשחקן הראשון הוא האלגוריתם יבחר מבין צאצאי אותה צומת, את הצאצא בעל הניקוד הגבוה ביותר. זאת בניגוד לאם כאשר הרמה בעץ שייכת לשחקן השני, והוא מנסה הרי להפחית את הניקוד.
- כאשר הרמה שייכת לשחקן השני, האלגוריתם יבחר עבורו את הניקוד הנמוך ביותר מבין צאצאיו. (הרי אנו מניחים את המקרה הגרוע ביותר ובו השחקן הנגדי יבחר במהלך הטוב לו ביותר)
- ובדרך הזו אנו עוברים על כל העץ ובסופו של דבר מגיעים לתוצאה ולמהלך רצוי.

גיזום Alpha-beta

גיזום אלפא בטא משמעו להוסיף משתנים שאנו מעבירים באלגוריתם (אלפא ובטא) על מנת לייעל את הקוד. הגיזום חותך ענפים בעץ המשחק שאין צורך לחפש בהם כי כבר קיים מהלך טוב יותר זמין. זה נקרא גיזום אלפא-ביתא מכיוון שהוא מעביר 2 פרמטרים נוספים בפונקציית המינימקס, כלומר אלפא ובטא.
בואו נגדיר את הפרמטרים אלפא ובטא.
אלפא הוא הערך הטוב ביותר שהמקסם יכול להבטיח כרגע ברמה זו ומעלה.
בטא הוא הערך הטוב ביותר שהמזעור יכול להבטיח כרגע ברמה זו ומעלה.

הקריאה הראשונית מתחילה מ-A. הערך של אלפא כאן הוא -INFINITY והערך של בטא הוא +INFINITY. ערכים אלה מועברים לצמתים הבאים בעץ. ב-A השחקן המקסימלי חייב לבחור מקסימום של B ו-C, אז A קורא ל-B ראשון
ברמה של B נמצא השחקן הממזער ועליהם לבחור מינימום של D ו-E ולכן קורא ל-D קודם.
ב-D, הוא מסתכל על הילד השמאלי שלו שהוא צומת עלה. הצומת הזה מחזיר ערך של 3.
כעת הערך של אלפא ב-D הוא max(-INF, 3) שהוא 3.
כדי להחליט אם כדאי להסתכל על הצומת הימני שלו או לא, הוא בודק את התנאי beta<=alpha. זה שקרי מכיוון שבטא = +INF ואלפא = 3. אז זה ממשיך את החיפוש.
D מסתכל כעת על הילד הימני שלו שמחזיר ערך של 5. ב-D, alpha = max(3, 5) שהוא 5. כעת הערך של צומת D הוא 5
D מחזיר ערך של 5 ל-B. ב-B, beta = min(+INF, 5) שהוא 5. לממזער מובטח כעת ערך של 5 או פחות. B מתקשר כעת ל-E כדי לראות אם הוא יכול לקבל ערך נמוך מ-5.
ב-E הערכים של אלפא ובטא אינם -INF ו-+INF אלא במקום -INF ו-5 בהתאמה, כי הערך של בטא שונה ב-B וזה מה ש-B העביר ל-E

כעת E מסתכל על הילד השמאלי שלו שהוא 6. ב-(E, alpha = max(-INF, 6 שהוא 6.
כאן התנאי הופך להיות אמיתי. בטא זה 5 ואלפא זה 6. אז בטא<=אלפא זה נכון. מכאן שהוא נשבר ו-E מחזיר 6 ל-B
שימו לב איך זה לא משנה מה הערך של הילד הנכון של E. זה יכול היה להיות +INF או -INF, זה עדיין לא היה משנה, אף פעם לא היינו צריכים להסתכל על זה כי למזעור הובטח ערך של 5 או פחות. אז ברגע שהמקסם ראה את ה-6 הוא ידע שהמזעור לעולם לא יגיע לכאן כי הוא יכול לקבל 5 בצד שמאל של B. בדרך זו לא נצטרך להסתכל על ה-9 הזה ובכך נחסך זמן חישוב.
E מחזירה ערך של 6 ל-B. ב-(B, beta = min(5, 6 שהוא 5. הערך של צומת B הוא גם 5



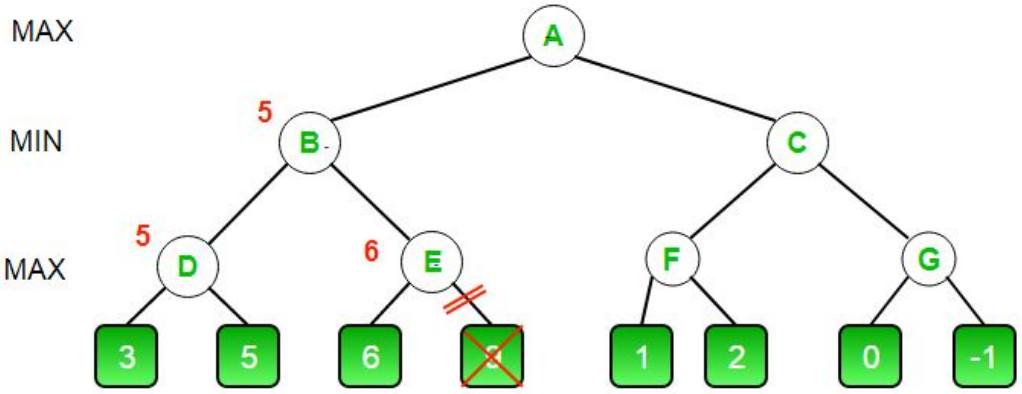
עד כה כך נראה עץ המשחקים שלנו. ה-9 נמחק מכיוון שהוא מעולם לא חושב.
B מחזירה 5 ל-A. ב-(A, alpha = max(-INF, 5 שהוא 5. כעת למקסם מובטח ערך של 5 או יותר. A קורא כעת ל-C כדי לראות אם הוא יכול לקבל ערך גבוה מ-5.
ב-C, אלפא = 5 ובטא = +INF. ג קורא ל-F
ב-F, אלפא = 5 ובטא = +INF. F מסתכל על הילד השמאלי שלו שהוא 1. (alpha = max(5, 1 שהוא עדיין 5.
F מסתכל על הילד הימני שלו שהוא 2. מכאן שהערך הטוב ביותר של הצומת הזה הוא 2. אלפא עדיין נשאר 5
F מחזירה ערך של 2 ל-C. ב-(C, beta = min(+INF, 2. התנאי בטא <= אלפא הופך להיות נכון שכן בטא = 2 ואלפא = 5. אז הוא נשבר והוא אפילו לא צריך לחשב את כל המשנה של G.
האינטואיציה מאחורי הפסקה זו היא שב-C למזעור הובטח ערך של 2 או פחות. אבל למקסם כבר היה מובטח ערך של 5 אם יבחר ב-B. אז למה שהממקסם יבחר אי פעם

ב-C ויקבל ערך פחות מ-2? שוב אתה יכול לראות שזה לא משנה מה היו 2 הערכים האחרונים. חסכנו גם הרבה חישובים על ידי דילוג על תת עץ שלם.

C מחזיר כעת ערך של 2 ל-A. לכן הערך הטוב ביותר ב-A הוא max(5, 2) שהוא 5.

מכאן שהערך האופטימלי שהמקסם יכול לקבל הוא 5

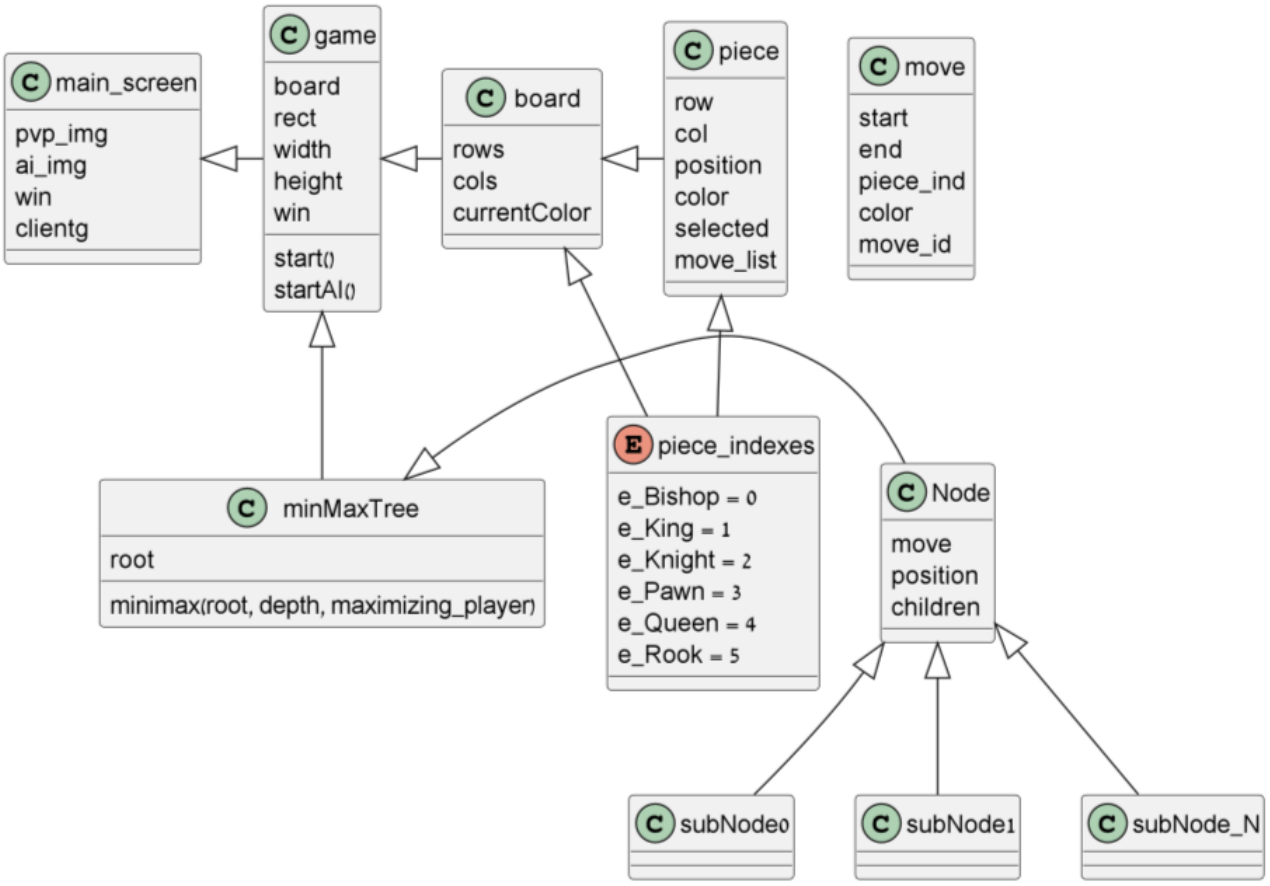
כך נראה עץ המשחקים האחרון שלנו. כפי שאתה יכול לראות G הומחק מכיוון שהוא מעולם לא חושב.



מבנה / ארכיטקטורה

UML

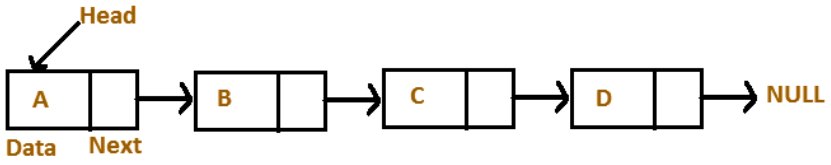
בתמונה ניתן לראות את פריסת המחלקות והמשתנים והזרימה ביניהם.



מבנה נתונים בפרויקט

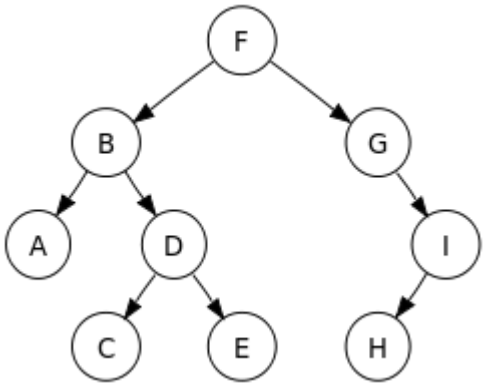
רשימה מקושרת:

רשימה מקושרת היא מבנה נתונים ליניארי, שבו האלמנטים אינם מאוחסנים במיקומי זיכרון רציפים. הרכיבים ברשימה מקושרת מקושרים באמצעות מצביעים. בפייתון המימוש של רשימה דינמאית הוא בעזרת רשימות מקושרות ושימוש מאחורי הקלעים בכתובות ובפוינטרים. בפרויקט



השתמשתי ברשימות על פני מערכים מכיוון שהפייתון פעולת האיטרצה בעבור רשימות נוחה וחוסרת בשורות מיותרות.

X



עצים הבנויים בצמתים:

בניגוד ל-Array ו-Linked List, שהם מבני נתונים ליניאריים, העץ הוא מבנה נתונים היררכי (או לא ליניארי). סיבה אחת להשתמש בעצים עשויה להיות בגלל שאתה רוצה לאחסן מידע שיוצר באופן טבעי היררכיה. לדוגמה, מערכת הקבצים במחשב:

מערכת קבצים. אם נארגן מפתחות בצורה של עץ נוכל לחפש מפתח נתון בזמן מתון (מהיר יותר מ-Linked List ואיטי יותר ממערכים). עצי חיפוש באיזון עצמי כמו AVL ועצים אדום-שחור מבטיחים גבול עליון של O(Logn) לחיפוש.

מימוש הפרויקט

piece.py:

```
import pygame
import os
import constants as c
import enum

b_bishop =  pygame.image.load(os.path.join("Assets",
"black_bishop.png"))
b_king =    pygame.image.load(os.path.join("Assets",
"black_king.png"))
b_knight =  pygame.image.load(os.path.join("Assets",
"black_knight.png"))
b_pawn =    pygame.image.load(os.path.join("Assets",
"black_pawn.png"))
b_queen =   pygame.image.load(os.path.join("Assets",
"black_queen.png"))
b_rook =    pygame.image.load(os.path.join("Assets",
"black_rook.png"))

w_bishop =  pygame.image.load(os.path.join("Assets",
"white_bishop.png"))
w_king =    pygame.image.load(os.path.join("Assets",
"white_king.png"))
w_knight =  pygame.image.load(os.path.join("Assets",
"white_knight.png"))
w_pawn =    pygame.image.load(os.path.join("Assets",
"white_pawn.png"))
w_queen =   pygame.image.load(os.path.join("Assets",
"white_queen.png"))
w_rook =    pygame.image.load(os.path.join("Assets",
"white_rook.png"))

class Piece_Enum(enum.Enum):
   e_Bishop = 0
   e_King = 1
```

```
    e_Knight = 2
    e_Pawn = 3
    e_Queen = 4
    e_Rook = 5

b = [b_bishop, b_king, b_knight, b_pawn, b_queen, b_rook]
w = [w_bishop, w_king, w_knight, w_pawn, w_queen, w_rook]

B = []
W = []

for img in b:
    B.append(pygame.transform.scale(img, (c.SQUARE - c.PADDING,
c.SQUARE - c.PADDING)))

for img in w:
    W.append(pygame.transform.scale(img, (c.SQUARE - c.PADDING,
c.SQUARE - c.PADDING)))

class move:
    def __init__(self, start, end, index, color):
        self.start_row = start[0]
        self.start_col = start[1]
        self.start = start
        self.end_row = end[0]
        self.end_col = end[1]
        self.end = end
        self.piece = index
        self.color = color
        self.move_id = id(self)

class Piece:
    img_ind = -1
    rect = (c.START_X, c.START_Y, c.BOARD_WIDTH, c.BOARD_HEIGHT)
    startX = rect[0]
    startY = rect[1]

    def __init__ (self, row, col, color):
        self.row = row
        self.col = col
        self.position = (row, col)
```

```
        self.color = color
        self.selected = False
        self.move_list = []
        self.inPassing = False
        self.isChecked = False
        self.king = False
        self.pawn = False
        self.moved = False
        self.can_move = True

    def copy(self, board):
        if self.img_ind == Piece_Enum.e_Bishop.value:
            board.board[self.row][self.col] = Bishop(self.row,
self.col, self.color)
        elif self.img_ind == Piece_Enum.e_King.value:
            board.board[self.row][self.col] = King(self.row,
self.col, self.color)
            board.board[self.row][self.col].moved = self.moved
            board.board[self.row][self.col].can_move = self.can_move
        elif self.img_ind == Piece_Enum.e_Knight.value:
            board.board[self.row][self.col] = Knight(self.row,
self.col, self.color)
        elif self.img_ind == Piece_Enum.e_Pawn.value:
            board.board[self.row][self.col] = Pawn(self.row,
self.col, self.color)
            board.board[self.row][self.col].first = self.first
            board.board[self.row][self.col].pawn = self.pawn
        elif self.img_ind == Piece_Enum.e_Queen.value:
            board.board[self.row][self.col] = Queen(self.row,
self.col, self.color)
        elif self.img_ind == Piece_Enum.e_Rook.value:
            board.board[self.row][self.col] = Rook(self.row,
self.col, self.color)
            board.board[self.row][self.col].moved = self.moved

    def UpdateValidMoves(self, board):
        self.move_list = self.valid_moves(board)

    def RemoveMove(self, move):
        for m in self.move_list:
            if m.move_id == move.move_id:
```

```
                self.move_list.remove(m)

    def getPiece(self):
        return (self.row, self.col)

    def IsSelected(self):
        return self.selected

    def UpdatePos(self, pos):
        self.row = pos[0]
        self.col = pos[1]
        self.position = pos
        if self.img_ind == Piece_Enum.e_Pawn.value:
            self.first = False

    def draw(self, win):
        if self.color == "w":
            drawThis = W[self.img_ind]
        else:
            drawThis = B[self.img_ind]

        x = c.PADDING/2 + self.startX + (self.col * self.rect[2]/8)
        y = c.PADDING/2 + self.startY + (self.row * self.rect[3]/8)

        if self.selected:
            pygame.draw.rect(win, (255, 0, 0), (x - c.PADDING/2, y -
c.PADDING/2, c.SQUARE, c.SQUARE), 5)

        win.blit(drawThis, (x, y))

        if self.isChecked:
            checkRect = c.CHECKED
            win.blit(checkRect, (x - c.PADDING/2, y - c.PADDING/2),
None, pygame.BLEND_MIN)

        if self.selected:
            moves = self.move_list

            for move in moves:
                x = 33 + self.startX + (move.end_col *
self.rect[2]/8)
```

```
                y = 33 + self.startY + (move.end_row *
self.rect[3]/8)
                pygame.draw.circle(win, (255, 0 ,0), (x, y), 11)


class Bishop(Piece):
    score = 3
    img_ind = Piece_Enum.e_Bishop.value


    def valid_moves(self, board):
        i = self.row
        j = self.col

        moves = []

        # left up
        count = j
        for y in range(i,-1,-1):
            p = board[y][count]
            if p == 0:
                m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count - 1
            if(count < 0):
                break

        # left down
        count = j
        for y in range(i,8,1):
            p = board[y][count]
            if p == 0:
```

```
                m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count - 1
            if(count < 0):
                break

        # right up
        count = j
        for y in range(i,-1,-1):
            p = board[y][count]
            if p == 0:
                m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count + 1
            if(count > 7):
                break

        # right down
        count = j
        for y in range(i,8,1):
            p = board[y][count]
            if p == 0:
```

```
                m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((self.row, self.col), (y, count),
self.img_ind, self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count + 1
            if(count > 7):
                break


        return moves

class King(Piece):
    score = 50
    img_ind = Piece_Enum.e_King.value

    def __init__(self, row, col, color):
        super().__init__(row, col, color)
        self.king = True
        self.moved = False


    def valid_moves(self, board):
        i = self.row
        j = self.col

        moves = []

        for x in range(-1, 2, 1):
            for y in range(-1, 2, 1):
                canAppend = True
                if 0 <= (i - y) <= 7 and 0 <= (j - x) <= 7:
                    p = board[i - y][j - x]
                    if p == 0:
```

```
                        m = move((i, j), (i - y, j - x),
self.img_ind, self.color)
                        moves.append(m)
                    else:
                        if p.color != self.color:
                            m = move((i, j), (i - y, j - x),
self.img_ind, self.color)
                            moves.append(m)
        if self.color == "w" and self.moved == False:
            if board[i][j-1] == 0 and board[i][j-2] == 0 and
board[i][j-3] == 0:
                m = move((i, j), (i, j-2), self.img_ind, self.color)
                moves.append(m)
            if board[i][j+1] == 0 and board[i][j+2] == 0:
                m = move((i, j), (i, j+2), self.img_ind, self.color)
                moves.append(m)
        if self.color == "b" and self.moved == False:
            if board[i][j-1] == 0 and board[i][j-2] == 0 and
board[i][j-3] == 0:
                m = move((i, j), (i, j-2), self.img_ind, self.color)
                moves.append(m)
            if board[i][j+1] == 0 and board[i][j+2] == 0:
                m = move((i, j), (i, j+2), self.img_ind, self.color)
                moves.append(m)

        return moves


class Knight(Piece):
    score = 3
    img_ind = Piece_Enum.e_Knight.value

    def valid_moves(self, board):
        i = self.row
        j = self.col

        moves = [] # [[pos], is_killing]
        # down left
        if i < 6 and j > 0:
            p = board[i+2][j-1]
            if p == 0:
```

```
                m = move((i, j), (i+2, j-1), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i+2, j-1), self.img_ind,
self.color)
                    moves.append(m)

        # down right
        if i < 6 and j < 7:
            p = board[i+2][j+1]
            if p == 0:
                m = move((i, j), (i+2, j+1), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i+2, j+1), self.img_ind,
self.color)
                    moves.append(m)

        # up left
        if i > 1 and j > 0:
            p = board[i-2][j-1]
            if p == 0:
                m = move((i, j), (i-2, j-1), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i-2, j-1), self.img_ind,
self.color)
                    moves.append(m)

        # up right
        if i > 1 and j < 7:
            p = board[i-2][j+1]
            if p == 0:
                m = move((i, j), (i-2, j+1), self.img_ind,
self.color)
```

```
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i-2, j+1), self.img_ind,
self.color)
                    moves.append(m)

        # left up
        if i > 0 and j > 1:
            p = board[i-1][j-2]
            if p == 0:
                m = move((i, j), (i-1, j-2), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i-1, j-2), self.img_ind,
self.color)
                    moves.append(m)

        # left down
        if i < 7 and j > 1:
            p = board[i+1][j-2]
            if p == 0:
                m = move((i, j), (i+1, j-2), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i+1, j-2), self.img_ind,
self.color)
                    moves.append(m)

        # right up
        if i > 0 and j < 6:
            p = board[i-1][j+2]
            if p == 0:
                m = move((i, j), (i-1, j+2), self.img_ind,
self.color)
                moves.append(m)
            else:
```

```
                if p.color != self.color:
                    m = move((i, j), (i-1, j+2), self.img_ind,
self.color)
                    moves.append(m)

        # right down
        if i < 7 and j < 6:
            p = board[i+1][j+2]
            if p == 0:
                m = move((i, j), (i+1, j+2), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i+1, j+2), self.img_ind,
self.color)
                    moves.append(m)



        return moves

class Pawn(Piece):
    score = 1
    img_ind = Piece_Enum.e_Pawn.value
    def __init__(self, row, col, color):
        super().__init__(row, col, color)
        self.first = True
        self.pawn = True

    def valid_moves(self, board):
        i = self.row
        j = self.col

        moves = []
        if self.color == "b":
            TwoRowsOpening = True
            if i + 1 < 8:
                p = board[i+1][j]
                if p == 0:
```

```
                    m = move((i, j), (i+1, j), self.img_ind,
self.color)
                    moves.append(m)
                else:
                    TwoRowsOpening = False
                if j + 1 < 8 and j - 1 >= 0:
                    p = board[i+1][j+1]
                    if p != 0:
                        if p.color != self.color:
                            m = move((i, j), (i+1, j+1),
self.img_ind, self.color)
                            moves.append(m)
                    p = board[i+1][j-1]
                    if p != 0:
                        if p.color != self.color:
                            if p.color != self.color:
                                m = move((i, j), (i+1, j-1),
self.img_ind, self.color)
                                moves.append(m)
                if j + 1 < 8:
                    p = board[i][j+1]
                    if p != 0:
                        if p.inPassing:
                            if p.color != self.color:
                                m = move((i, j), (i+1, j+1),
self.img_ind, self.color)
                                moves.append(m)
                if j - 1 >= 0:
                    p = board[i][j-1]
                    if p != 0:
                        if p.inPassing:
                            if p.color != self.color:
                                m = move((i, j), (i+1, j-1),
self.img_ind, self.color)
                                moves.append(m)
            if self.first and TwoRowsOpening:
                if i + 2 < 8:
                    p = board[i+2][j]
                    if p == 0:
                        m = move((i, j), (i+2, j), self.img_ind,
self.color)
```

```
                        moves.append(m)
        else:
            TwoRowsOpening = True
            if i >= 0:
                p = board[i-1][j]
                if p == 0:
                    m = move((i, j), (i-1, j), self.img_ind,
self.color)
                    moves.append(m)
                else:
                    TwoRowsOpening = False
                if j + 1 < 8 and j - 1 >= 0:
                    p = board[i-1][j+1]
                    if p != 0:
                        if p.color != self.color:
                            m = move((i, j), (i-1, j+1),
self.img_ind, self.color)
                            moves.append(m)
                    p = board[i-1][j-1]
                    if p != 0:
                        if p.color != self.color:
                            m = move((i, j), (i-1, j-1),
self.img_ind, self.color)
                            moves.append(m)
                if j + 1 < 8:
                    p = board[i][j+1]
                    if p != 0:
                        if p.inPassing:
                            if p.color != self.color:
                                m = move((i, j), (i-1, j+1),
self.img_ind, self.color)
                                moves.append(m)
                if i - 1 >= 0:
                    p = board[i][j-1]
                    if p != 0:
                        if p.inPassing:
                            if p.color != self.color:
                                m = move((i, j), (i-1, j-1),
self.img_ind, self.color)
                                moves.append(m)
            if self.first and TwoRowsOpening:
```

```
                if i > 1:
                    p = board[i-2][j]
                    if p == 0:
                        m = move((i, j), (i-2, j), self.img_ind,
self.color)
                        moves.append(m)




        return moves


class Queen(Piece):
    score = 9
    img_ind = Piece_Enum.e_Queen.value

    def valid_moves(self, board):
        i = self.row
        j = self.col

        moves = []

        # left up
        count = j
        for y in range(i,-1,-1):
            p = board[y][count]
            if p == 0:
                m = move((i, j), (y, count), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (y, count), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count - 1
            if(count < 0):
```

```
                break

        # left down
        count = j
        for y in range(i,8,1):
            p = board[y][count]
            if p == 0:
                m = move((i, j), (y, count), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (y, count), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count - 1
            if(count < 0):
                break

        # right up
        count = j
        for y in range(i,-1,-1):
            p = board[y][count]
            if p == 0:
                m = move((i, j), (y, count), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (y, count), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count + 1
            if(count > 7):
                break
```

```
        # right down
        count = j
        for y in range(i,8,1):
            p = board[y][count]
            if p == 0:
                m = move((i, j), (y, count), self.img_ind,
self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (y, count), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif y != self.row or count != self.col:
                    break
            count = count + 1
            if(count > 7):
                break

        # up
        for x in range(i,-1,-1):
            p = board[x][j]
            if p == 0:
                m = move((i, j), (x, j), self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (x, j), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif x != self.row or j != self.col:
                    break

        # down
        for x in range(i,8,1):
            p = board[x][j]
            if p == 0:
                m = move((i, j), (x, j), self.img_ind, self.color)
```

```
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (x, j), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif x != self.row or j != self.col:
                    break

        # left
        for x in range(j,-1,-1):
            p = board[i][x]
            if p == 0:
                m = move((i, j), (i, x), self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i, x), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif i != self.row or x != self.col:
                    break

        # right
        for x in range(j,8,1):
            p = board[i][x]
            if p == 0:
                m = move((i, j), (i, x), self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i, x), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif i != self.row or x != self.col:
                    break

        return moves
```

```
class Rook(Piece):
    score = 5
    img_ind = Piece_Enum.e_Rook.value

    def __init__(self, row, col, color):
        super().__init__(row, col, color)
        self.moved = False
        self.side = False
        if col > 0:
            self.side = True

    def valid_moves(self, board):
        i = self.row
        j = self.col

        moves = []

        # up
        for x in range(i,-1,-1):
            p = board[x][j]
            if p == 0:
                m = move((i, j), (x, j), self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (x, j), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif x != self.row or j != self.col:
                    break

        # down
        for x in range(i,8,1):
            p = board[x][j]
            if p == 0:
                m = move((i, j), (x, j), self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
```

```
                    m = move((i, j), (x, j), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif x != self.row or j != self.col:
                    break

        # left
        for x in range(j,-1,-1):
            p = board[i][x]
            if p == 0:
                m = move((i, j), (i, x), self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i, x), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif i != self.row or x != self.col:
                    break

        # right
        for x in range(j,8,1):
            p = board[i][x]
            if p == 0:
                m = move((i, j), (i, x), self.img_ind, self.color)
                moves.append(m)
            else:
                if p.color != self.color:
                    m = move((i, j), (i, x), self.img_ind,
self.color)
                    moves.append(m)
                    break
                elif i != self.row or x != self.col:
                    break

        return moves
```

board.py

```
import numpy
import constants as c
from piece import move as mv
from piece import B, W
from piece import Piece_Enum as pe

from piece import Bishop
from piece import King
from piece import Knight
from piece import Pawn
from piece import Queen
from piece import Rook

class Node:
   def __init__(self, _value=0, _move=None, _position=None):
       self.value = _value
       self.move = _move
       self.position = _position
       self.children = []

   def set_value(self, _value):
       self.value = _value

class minMaxTree:
   def __init__(self, board, move=None):
       self.root = Node(board.boardScore, move, board)

   def build_tree(self, move_tree, depth, color):
       other_color = move_tree.position.get_other_color(color)
       if len(move_tree.children) == 0:
           color_moves =
move_tree.position.ai_danger_moves(other_color)
           for _move in color_moves:
               _position = Board(8,8,color)
               move_tree.position.CopyTo(_position)
               _position.move(_move)
               _score = 0
               _child = Node(_score, _move, _position)
```

```
                move_tree.children.append(_child)
        if depth == 0:
            return
        _depth = depth - 1
        for _child in move_tree.children:
            self.build_tree(_child, _depth, other_color)

    def minimax(self, root, depth, maximizing_player):
        root.position.checkMate()
        if depth == 0 or (root.position.is_b_mated or
root.position.is_w_mated):
            return root.position.boardScore

        if maximizing_player:
            maxEval = -numpy.inf
            for _child in root.children:
                _eval = self.minimax(_child, depth - 1, False)
                maxEval = max(maxEval, _eval)
            root.set_value(maxEval)
            return maxEval
        else:
            minEval = numpy.inf
            for _child in root.children:
                _eval = self.minimax(_child, depth - 1, True)
                minEval = min(minEval, _eval)
            root.set_value(minEval)
            return minEval

    def get_move_from_eval(self, root, score):
        for _child in root.children:
            if score == _child.value:
                return root.move
        return None

    def set_new_root(self, score):
        for _child in self.root.children:
            if score == _child.value:
                break
        self.root = _child
```

```
class Board:
    def __init__(self, rows, cols, turn):
        self.rows = rows
        self.cols = cols
        self.currentColor = turn
        self.toolsWin = False
        self.w_tooltip = False
        self.b_tooltip = False
        self.w_tooltip_ind = (-1, -1)
        self.b_tooltip_ind = (-1, -1)
        self.ToggleTurns = True
        self.w_stalemate = False
        self.b_stalemate = False
        self.w_is_mated = False
        self.b_is_mated = False
        self.w_checked = False
        self.b_checked = False
        self.TogglePawns = True
        self.boardScore = 0

        self.b_right_rook_moved = False
        self.b_left_rook_moved = False
        self.b_king_moved = False
        self.w_right_rook_moved = False
        self.w_left_rook_moved = False
        self.w_king_moved = False


        self.board = [[0 for x in range(cols)] for _ in range(rows)]

        self.board[0][0] = Rook(0,0,"b")
        self.board[0][1] = Knight(0,1,"b")
        self.board[0][2] = Bishop(0,2,"b")
        self.board[0][3] = Queen(0,3,"b")
        self.board[0][4] = King(0,4,"b")
        self.board[0][5] = Bishop(0,5,"b")
        self.board[0][6] = Knight(0,6,"b")
        self.board[0][6] = Knight(0,6,"b")
        self.board[0][7] = Rook(0,7,"b")
        if self.TogglePawns:
            for j in range(8):
```

```
            self.board[1][j] = Pawn(1, j, "b")
        self.board[7][0] = Rook(7,0,"w")
        self.board[7][1] = Knight(7,1,"w")
        self.board[7][2] = Bishop(7,2,"w")
        self.board[7][3] = Queen(7,3,"w")
        self.board[7][4] = King(7,4,"w")
        self.board[7][5] = Bishop(7,5,"w")
        self.board[7][6] = Knight(7,6,"w")
        self.board[7][7] = Rook(7,7,"w")
        if self.TogglePawns:
            for j in range(8):
                self.board[6][j] = Pawn(6, j, "w")

        self.update_init_moves()

    def evaluate_move(self, move):
        move_score = 0
        p = self.board[move.end[0]][move.end[1]]
        if p != 0:
            if p.color == "w":
                move_score = p.score
            else:
                move_score = -p.score
        return move_score

    def CopyTo(self, board):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.board[i][j] == 0:
                    board.board[i][j] = 0
                else:
                    self.board[i][j].copy(board)

    def draw(self, win):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.board[i][j] != 0:
                    self.board[i][j].draw(win)
        if self.toolsWin:
            if self.currentColor == "w":
                bishop = W[pe.e_Bishop.value]
```

```
                knight = W[pe.e_Knight.value]
                queen = W[pe.e_Queen.value]
                rook = W[pe.e_Rook.value]
            else:
                bishop = B[pe.e_Bishop.value]
                knight = B[pe.e_Knight.value]
                queen = B[pe.e_Queen.value]
                rook = B[pe.e_Rook.value]
            bishop_des = (c.START_X - c.SQUARE*2 + 20, c.START_Y)
            knight_des = (c.START_X - c.SQUARE, c.START_Y)
            queen_des = (c.START_X - c.SQUARE*2 + 20, c.START_Y +
c.SQUARE)
            rook_des = (c.START_X - c.SQUARE, c.START_Y + c.SQUARE)
            win.blit(bishop, bishop_des)
            win.blit(knight, knight_des)
            win.blit(queen, queen_des)
            win.blit(rook, rook_des)

    def update_init_moves(self):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.board[i][j] != 0:
                    self.board[i][j].UpdateValidMoves(self.board)

    def is_move_a_threat(self, danger_move, my_move):
        if danger_move.end == my_move.end:
            y = danger_move.start_row
            x = danger_move.start_col
            if self.board[y][x]:
                if self.board[y][x].pawn:
                    if danger_move.end_col - my_move.end_col == 0:
                        return False
                    else:
                        return True
                else:
                    return True
            else:
                return False
        return False

    def UpdateKingMoves(self):
```

```
        Wmoves_without, Bmoves_without =
self.get_all_moves_without_kings()
        w_king = self.get_king("w")
        w_king_actual_moves = []
        for _move in w_king.move_list:
            w_king_actual_moves.append(_move)
        for w_move in w_king.move_list:
            for w_secret_move in Bmoves_without:
                if self.is_move_a_threat(w_secret_move, w_move):
                    w_king_actual_moves.remove(w_move)
                    break
        w_king.move_list = w_king_actual_moves
        if len(w_king.move_list) == 0:
            w_king.can_move = False

        b_king = self.get_king("b")
        b_king_actual_moves = []
        for _move in b_king.move_list:
            b_king_actual_moves.append(_move)
        for b_move in b_king.move_list:
            for b_secret_move in Wmoves_without:
                if self.is_move_a_threat(b_secret_move, b_move):
                    b_king_actual_moves.remove(b_move)
                    break
        b_king.move_list = b_king_actual_moves
        if len(b_king.move_list) == 0:
            b_king.can_move = False

    def UpdateBoard(self):
        self.update_init_moves()
        self.update_pinned_moves()
        self.UpdateKingMoves()
        self.is_checked("w")
        self.is_checked("b")

    def get_danger_moves(self, color):
        danger_moves = []
        _pieces =
self.get_all_pieces_by_color(self.get_other_color(color))
        for _piece in _pieces:
            for _move in _piece.move_list:
```

```
                danger_moves.append(_move)
        return danger_moves

    def ai_danger_moves(self, color):
        self.UpdateBoard()
        danger_moves = []
        _pieces =
self.get_all_pieces_by_color(self.get_other_color(color))
        for _piece in _pieces:
            for _move in _piece.move_list:
                danger_moves.append(_move)
        return danger_moves

    def get_all_pieces(self):
        _pieces = []
        for i in range(self.rows):
            for j in range(self.cols):
                if self.board[i][j] != 0:
                    _pieces.append(self.board[i][j])
        return _pieces

    def get_all_pieces_by_color(self, color):
        _pieces = []
        for i in range(self.rows):
            for j in range(self.cols):
                if self.board[i][j] != 0:
                    if self.board[i][j].color == color:
                        _pieces.append(self.board[i][j])
        return _pieces

    def get_danger_moves_by_king(self, king_color):
        _danger_moves = []
        _king_pos = self.GiveKingPos(king_color)
        _pieces = self.get_all_pieces()
        for _piece in _pieces:
            if _piece.color != king_color:
                for _move in _piece.move_list:
                    if _king_pos == _move.end:
                        _danger_moves.append(_move)
        return _danger_moves
```

```
    def get_other_color(self, color):
        _color = "w"
        if color == "w":
            _color = "b"
        return _color

    def get_danger_moves_by_piece(self, by_piece):
        danger_moves = []
        other_color = self.get_other_color(by_piece.color)
        _pieces = self.get_all_pieces_by_color(other_color)
        for _piece in _pieces:
            for _move in _piece.move_list:
                if by_piece.position == _move.end:
                    danger_moves.append(_move)
        return danger_moves

    def get_pinned_pieces(self, color):
        pinned_pieces = []
        _pieces = self.get_all_pieces_by_color(color)
        _king = self.get_king(color)
        for _piece in _pieces:
            secret_danger_moves =
self.get_danger_moves_without_piece(_piece)
            for _move in secret_danger_moves:
                if _king.position == _move.end:
                    pinned_pieces.append(_piece)
        return pinned_pieces

    def is_move_dangerous(self, move, color):
        tmp = Board(8,8,"w")
        self.CopyTo(tmp)
        tmp.move(move)
        tmp.update_init_moves()
        return tmp.is_checked(color)

    def update_pinned_moves(self):
        w_pinned = self.get_pinned_pieces("w")
        for w_piece in w_pinned:
            w_actual_moves = []
            for w_move in w_piece.move_list:
                if not self.is_move_dangerous(w_move, "w"):
```

```
                w_actual_moves.append(w_move)
        w_piece.move_list = w_actual_moves

    b_pinned = self.get_pinned_pieces("b")
    for b_piece in b_pinned:
        b_actual_moves = []
        for b_move in b_piece.move_list:
            if not self.is_move_dangerous(b_move, "b"):
                b_actual_moves.append(b_move)
        b_piece.move_list = b_actual_moves

def get_all_moves(self, color):
    danger_moves = []
    for i in range(self.rows):
        for j in range(self.cols):
            if self.board[i][j] != 0:
                if self.board[i][j].color == color:
                    for move in self.board[i][j].move_list:
                        danger_moves.append(move)
    return danger_moves

def remove_piece_from_board(self, piece):
    if piece.img_ind == pe.e_King:
        return None
    for i in range(self.rows):
        for j in range(self.cols):
            if self.board[i][j] == piece:
                returned_piece = self.board[i][j]
                self.board[i][j] = 0
                return returned_piece
    return None

def get_danger_moves_without_piece(self, by_piece):
    tmp = Board(8,8,self.currentColor)
    self.CopyTo(tmp)
    piece_to_remove = tmp.board[by_piece.row][by_piece.col]
    tmp.remove_piece_from_board(piece_to_remove)
    tmp.update_init_moves()
    moves_without_piece = tmp.get_danger_moves(by_piece.color)
    return moves_without_piece
```

```
    def get_danger_moves_with_piece_in_new_pos(self, by_piece, move):
        tmp = Board(8,8, self.currentColor)
        self.CopyTo(tmp)
        tmp.move(move)
        tmp.update_init_moves()
        new_moves = tmp.get_danger_moves(by_piece.color)
        return new_moves

    def is_checked_with_piece_in_new_pos(self, color, move):
        tmp = Board(8,8, self.currentColor)
        self.CopyTo(tmp)
        tmp.move(move)
        tmp.update_init_moves()
        return tmp.is_checked(color)

    def get_all_moves_without_kings(self):
        tmp = Board(8,8,self.currentColor)
        self.CopyTo(tmp)
        tmp_white_king = tmp.get_king("w")
        tmp_black_king = tmp.get_king("b")
        wy = tmp_white_king.row
        wx = tmp_white_king.col
        by = tmp_black_king.row
        bx = tmp_black_king.col
        tmp.board[wy][wx] = 0
        tmp.board[by][bx] = 0
        tmp.update_init_moves()
        w_moves_without = tmp.get_danger_moves("b")
        b_moves_without = tmp.get_danger_moves("w")

        return w_moves_without, b_moves_without

    def dangerous_pieces(self, piece):
        attacking_pieces = []
        _pieces =
self.get_all_pieces_by_color(self.get_other_color(piece.color))
        for _piece in _pieces:
            for _move in _piece.move_list:
                if _move.end == piece.position:
                    attacking_pieces.append(_piece)
        return attacking_pieces
```

```
    def can_king_be_defended(self, color): # all moves here are legal
        color_pieces = self.get_all_pieces_by_color(color)
        _king = self.get_king(color)
        newer_moves = []
        attacking_pieces = self.dangerous_pieces(_king)
        for _piece in color_pieces:
            for _move in _piece.move_list:
                new_moves =
self.get_danger_moves_with_piece_in_new_pos(_piece, _move)
                for _move in new_moves:
                    newer_moves.append(_move)
                for new_move in new_moves:
                    if _king.position != new_move.end:
                        newer_moves.remove(new_move)
                if len(newer_moves) == 0:
                    return True
        return False

    def checkMate(self):
        black = "b"
        white = "w"
        if self.b_is_mated or self.w_is_mated:
            return True
        w_danger_moves = self.get_danger_moves(white)
        b_danger_moves = self.get_danger_moves(black)
        w_king = self.get_king(white)
        b_king = self.get_king(black)

        w_king_moves = w_king.move_list
        w_king_actual_moves = w_king_moves
        for w_move in w_king_moves:
            for w_danger_move in w_danger_moves:
                if w_move.end == w_danger_move.end:
                    w_king_actual_moves.remove(w_move)
                    break
        w_king.move_list = w_king_actual_moves

        b_king_moves = b_king.move_list
        b_king_actual_moves = b_king_moves
```

```
        for b_move in b_king_moves:
            for b_danger_move in b_danger_moves:
                if b_move.end == b_danger_move.end:
                    b_king_actual_moves.remove(b_move)
                    break
        b_king.move_list = b_king_actual_moves

        if self.w_checked:
            if len(w_king_actual_moves) == 0:
                w_king.can_move = False
                if self.can_king_be_defended(white) == False:
                    self.w_is_mated = True
                    return True

        if self.b_checked:
            if len(b_king_actual_moves) == 0:
                b_king.can_move = False
                if self.can_king_be_defended(black) == False:
                    self.b_is_mated = True
                    return True

        overall_w_moves, overall_b_moves =
self.get_all_moves_without_kings()
        if len(overall_b_moves) == 0:
            if len(b_king_actual_moves) == 0:
                self.b_stalemate = True
                return False

        if len(overall_w_moves) == 0:
            if len(w_king_actual_moves) == 0:
                self.w_stalemate = True
                return False

        return False


    def get_king(self, color):
        _pieces = self.get_all_pieces()
        for _piece in _pieces:
            if _piece.king and _piece.color == color:
                return _piece
```

```
        return None


    def is_checked(self, color):
        danger_moves = self.get_danger_moves(color) # other color all
moves
        _king = self.get_king(color)
        self.w_checked = False
        self.b_checked = False
        _king.isChecked = False
        for _move in danger_moves:
            if _king.position == _move.end:
                _king.isChecked = True
                if color == "w":
                    self.w_checked = True
                else:
                    self.b_checked = True
                return True
        return False


    def select(self, y, x):
        if self.ToggleTurns:
            toselect = False
            if self.board[y][x].selected == False:
                if self.currentColor == self.board[y][x].color:
                    toselect = True
            for i in range(self.rows):
                for j in range(self.cols):
                    if self.board[i][j] != 0:
                        self.board[i][j].selected = False
            if toselect:
                self.board[y][x].selected = True
                self.UpdateBoard()
        else:
            toselect = False
            if(self.board[y][x].selected == False):
                toselect = True
            for i in range(self.rows):
                for j in range(self.cols):
                    if self.board[i][j] != 0:
```

```
                    self.board[i][j].selected = False
            if(toselect):
                self.board[y][x].selected = True
                self.UpdateBoard()

    def castle(self, move, side):
        if side:
            if move.color == "w":
                self.move(move)
                m = mv((7,7), (7,5), pe.e_Rook.value, move.color)
                self.move(m)
            else:
                self.move(move)
                m = mv((0,7), (0,5), pe.e_Rook.value, move.color)
                self.move(m)
        else:
            if move.color == "w":
                self.move(move)
                m = mv((7,0), (7,3), pe.e_Rook.value, move.color)
                self.move(m)
            else:
                self.move(move)
                m = mv((0,0), (0,3), pe.e_Rook.value, move.color)
                self.move(m)

    def authorise_move(self, move, piece):
        if self.checkMate():
            return False
        if self.is_checked_with_piece_in_new_pos(piece.color, move):
            return False
        self.check_tooltip(move)
        if self.b_tooltip or self.w_tooltip:
            return False
        if not self.check_can_castle(move):
            self.move(move)
        self.UpdateBoard()
        self.checkMate()
        self.currentColor = self.get_other_color(piece.color)
        piece.selected = False
        return True
```

```
def ai_move_logic(self, tree, is_maximizing):
    _color = "w"
    if not is_maximizing:
        _color = "b"
    self.UpdateBoard()
    _score = tree.minimax(self.tree.root, 4, is_maximizing)
    _move = tree.get_move_from_eval(self.tree.root, _score)
    _piece = self.board[_move.start_row][_move.start_col]
    if self.authorise_move(_move, _piece):
        tree.set_new_root(_score)
        tree.build_tree(self.tree, 4, _color)


def ai_reaction_move_logic(self, i, j):
    if(0 <= i < 8 and 0 <= j < 8):
        IsSelected = self.IsSelected()
        y = IsSelected[1]
        x = IsSelected[2]
        _piece = self.board[y][x]
        self.UpdateBoard()
        if IsSelected[0]:
            if self.IsValid((i, j), _piece):
                _move = self.get_move((i, j), _piece)
                if self.authorise_move(_move, _piece):
                    return True, _move
            else:
                if(self.board[i][j] != 0):
                    self.select(i,j)
        else:
            if(self.board[i][j] != 0):
                self.select(i,j)
    return False, None

def move_logic(self, i, j):
    change = False
    if(0 <= i < 8 and 0 <= j < 8):
        IsSelected = self.IsSelected()
        y = IsSelected[1]
        x = IsSelected[2]
        _piece = self.board[y][x]
```

```
            self.UpdateBoard()
            if IsSelected[0]:
                if self.IsValid((i, j), _piece):
                    _move = self.get_move((i, j), _piece)
                    if self.authorise_move(_move, _piece):
                        change = True
                else:
                    if(self.board[i][j] != 0):
                        self.select(i,j)
            else:
                if(self.board[i][j] != 0):
                    self.select(i,j)
        return change


    def update_moved_bools(self, p):
        if p.img_ind == pe.e_Rook.value:
            if p.moved == False:
                p.moved = True
                if p.color == "w":
                    if p.side:
                        self.w_right_rook_moved = True
                    else:
                        self.w_left_rook_moved = True
                else:
                    if p.side:
                        self.b_right_rook_moved = True
                    else:
                        self.b_left_rook_moved = True
        if p.img_ind == pe.e_King.value:
            if p.moved == False:
                p.moved = True
                if p.color == "w":
                    self.w_king_moved = True
                else:
                    self.b_king_moved = True

    def check_can_castle(self, move):
        p = self.board[move.start_row][move.start_col]
        if not p.img_ind == pe.e_King.value:
            return False
```

```
        if move.color == "w":
            if not self.w_king_moved:
                if move.start_col - move.end_col < -1:
                    if not self.w_right_rook_moved:
                        if not p.isChecked:
                            self.castle(move, True)
                            return True
                else:
                    if not self.w_left_rook_moved:
                        if not p.isChecked:
                            self.castle(move, False)
                            return True
        else:
            if not self.b_king_moved:
                if move.start_col - move.end_col < -1:
                    if not self.b_right_rook_moved:
                        if not p.isChecked:
                            self.castle(move, True)
                            return True
                else:
                    if not self.b_left_rook_moved:
                        if not p.isChecked:
                            self.castle(move, False)
                            return True
        return False

    def move(self, move):
        self.check_tooltip(move)
        p = self.board[move.start_row][move.start_col]
        self.update_moved_bools(p)
        p = self.board[move.end_row][move.end_col]
        if p != 0:
            if p.color == "b":
                self.boardScore -= p.score
            else:
                self.boardScore += p.score

self.board[move.start_row][move.start_col].UpdatePos((move.end_row,
move.end_col))
        self.board[move.end_row][move.end_col] =
self.board[move.start_row][move.start_col]
```

```
        self.board[move.start_row][move.start_col] = 0


    def get_move(self, end, piece):
        for _move in piece.move_list:
            if end == _move.end:
                return _move
        return None


    def IsValid(self, pos, piece):
        y = pos[0]
        x = pos[1]
        for _move in piece.move_list:
            if y == _move.end_row and x == _move.end_col:
                return True
        return False

    def IsSelected(self):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.board[i][j] != 0:
                    if self.board[i][j].selected:
                     return True, i, j
        return False, -1, -1

    def choose_from_tools(self, tool, color, move):
        self.move(move)
        y = move.end_row
        x = move.end_col
        if tool == "bishop":
            self.board[y][x] = Bishop(y,x, color)
        if tool == "queen":
            self.board[y][x] = Queen(y,x, color)
        if tool == "knight":
            self.board[y][x] = Knight(y,x, color)
        if tool == "rook":
            self.board[y][x] = Rook(y,x, color)
        self.UpdateBoard()
        self.checkMate()
        self.currentColor = self.get_other_color(color)
```

```
        self.board[y][x].selected = False

    def choose_tool_from_pos(self, pos):
        if pos == (-1, -1):
            return False
        if pos == (0,0):
            tool = "bishop"
        if pos == (0,1):
            tool = "knight"
        if pos == (1,0):
            tool = "queen"
        if pos == (1,1):
            tool = "rook"
        if self.currentColor == "w":
            self.choose_from_tools(tool, "w", self.w_tooltip_ind)
        else:
            self.choose_from_tools(tool, "b", self.b_tooltip_ind)
        return True

    def check_tooltip(self, move):
        for _pawn in self.get_all_pieces():
            if _pawn.img_ind == pe.e_Pawn.value:
                _pawn.inPassing = False
        y = move.start_row
        x = move.start_col
        ey = move.end_row
        ex = move.end_col
        self.w_tooltip = False
        self.b_tooltip = False
        if self.board[y][x].img_ind == pe.e_Pawn.value:
            if self.board[y][x].color == "b":
                if ey == 7:
                    self.b_tooltip = True
                    self.b_tooltip_ind = move
                else:
                    if ex != x:
                        p = self.board[ey][ex]
                        if p == 0:
                            r = self.board[ey+1][ex]
                            self.board[ey+1][ex] = 0
                    else:
```

```
                if abs(ey - y) > 1:
                    if self.board[y][x].first:
                        self.board[y][x].inPassing = True
                else:
                    self.board[y][x].inPassing = False
else:
    if ey == 0:
        self.w_tooltip = True
        self.w_tooltip_ind = move
    else:
        if ex != x:
            p = self.board[ey][ex]
            if p == 0:
                r = self.board[ey+1][ex]
                self.board[ey+1][ex] = 0
        else:
            if abs(ey - y) > 1:
                if self.board[y][x].first:
                    self.board[y][x].inPassing = True
                else:
                    self.board[y][x].inPassing = False
```

game.py

```
import pygame
import os
from board import Board
from board import minMaxTree
import constants as c
import time


class game:
   def __init__(self, win):
       self.board =
pygame.transform.scale(pygame.image.load(os.path.join("Assets",
"board_alt.png")), (c.BOARD_ALT_WIDTH, c.BOARD_ALT_HEIGHT))
       self.rect = (c.START_X, c.START_Y, c.BOARD_WIDTH,
c.BOARD_HEIGHT)
       self.width = c.BOARD_ALT_WIDTH
       self.height = c.BOARD_ALT_HEIGHT
       self.win = win
       pygame.display.set_caption("Chess!!")

   def redraw_gamewindow(self, win, bo, p1Time, p2Time):
       win.blit(self.board, (0,0))
       bo.draw(win)
       font = pygame.font.SysFont("Arial", 30)
       p1Timemin = int(p1Time // 60)
       p1Timemsec = int(p1Time % 60)
       p2Timemin = int(p2Time // 60)
       p2Timemsec = int(p2Time % 60)
       txt = font.render("Player 1:
{:02d}:{:02d}".format(p1Timemin,p1Timemsec), 1, (255, 255, 255))
       txt2 = font.render("Player 2:
{:02d}:{:02d}".format(p2Timemin,p2Timemsec), 1, (255, 255, 255))
       win.blit(txt2, (450, 20))
       win.blit(txt, (450, 720))
       pygame.display.update()

   def end_screen(self, win, text):
       pygame.font.init()
```

```
        font = pygame.font.SysFont("helvetic", 80)
        txt = font.render(text, 1, (255, 0,0))
        win.blit(txt, (self.width/2 - txt.get_width() / 2, 300))
        pygame.display.update()

        pygame.time.set_timer(pygame.USEREVENT+1, 1)

        run = True
        br = False
        while run:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                if event.type == pygame.KEYUP:
                    br = True
                    break
            if br:
                break
        self.start()

    def ToolsClick(self, pos):
        rect = (c.START_X - c.SQUARE*2 + 20, c.START_Y, c.SQUARE*2 -
20, c.SQUARE*2)
        x = pos[0]
        y = pos[1]
        if rect[0] < x < rect[0] + rect[2]:
            if rect[1] < y < rect[1] + rect[3]:
                divX = x - rect[0]
                divY = y - rect[1]
                j = int(divX / (rect[2]/2))
                i = int(divY / (rect[3]/2))
                return i, j
        return -1, -1

    def click(self, pos):
        """
        :return: pos  (x, y) in range 0-7 0-7
        """
        x = pos[0]
        y = pos[1]
        if self.rect[0] < x < self.rect[0] + self.rect[2]:
```

```
        if self.rect[1] < y < self.rect[1] + self.rect[3]:
            divX = x - self.rect[0]
            divY = y - self.rect[1]
            j = int(divX / (self.rect[2]/8))
            i = int(divY / (self.rect[3]/8))
            return i, j
    return -1, -1

def start(self):
    p1Time = 60 * 15
    p2Time = 60 * 15
    clock = pygame.time.Clock()
    turn = "w"
    bo = Board(8,8, turn)
    run = True
    startTime = time.time()
    while run:
        clock.tick(30)

        if turn == "w":
            p1Time -= (time.time() - startTime)
        else:
            p2Time -= (time.time() - startTime)

        startTime = time.time()


        self.redraw_gamewindow(self.win, bo, p1Time, p2Time)

        if bo.b_is_mated:
            self.end_screen(self.win, "white won!")
        if bo.w_is_mated:
            self.end_screen(self.win, "black won!")
        if bo.b_stalemate or bo.w_stalemate:
            self.end_screen(self.win, "stalemate!")

        if bo.w_tooltip or bo.b_tooltip:
            while bo.w_tooltip or bo.b_tooltip:
                bo.toolsWin = True
                _change = False
```

```
                    if turn == "w":
                        p1Time -= (time.time() - startTime)
                    else:
                        p2Time -= (time.time() - startTime)

                    startTime = time.time()

                    self.redraw_gamewindow(self.win, bo, p1Time,
p2Time)

                    for event in pygame.event.get():
                        if event.type == pygame.QUIT:
                            run = False
                            quit()
                            pygame.quit()

                        if event.type == pygame.K_ESCAPE:
                            return

                        if event.type == pygame.MOUSEMOTION:
                            pass

                        if event.type == pygame.MOUSEBUTTONDOWN:
                            pos = pygame.mouse.get_pos()
                            i, j = self.ToolsClick(pos)
                            _change = bo.choose_tool_from_pos((i,j))

                    if _change:
                        if turn == "w":
                            turn = "b"
                        else:
                            turn = "w"
                        change = False
                        bo.b_tooltip = False
                        bo.w_tooltip = False
                        bo.toolsWin = False
                        break

                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        run = False
```

```
                quit()
                pygame.quit()

            if event.type == pygame.MOUSEMOTION:
                pass

            if event.type == pygame.K_ESCAPE:
                return

            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                i, j = self.click(pos)
                change = bo.move_logic(i, j)

                if change:
                    if turn == "w":
                        turn = "b"
                    else:
                        turn = "w"
                    change = False

            if event.type == pygame.KEYUP:
                bo = Board(8,8, "w")
                p1Time = 60 * 15
                p2Time = 60 * 15

    def redraw_gamewindow_ai(self, win, bo, p1Time, chosen_color):
        win.blit(self.board, (0,0))
        bo.draw(win)
        font = pygame.font.SysFont("Arial", 30)
        p1Timemin = int(p1Time // 60)
        p1Timemsec = int(p1Time % 60)
        if chosen_color == "w":
            txt = font.render("Player 1:
{:02d}:{:02d}".format(p1Timemin,p1Timemsec), 1, (255, 255, 255))
            win.blit(txt, (450, 720))
        else:
            txt = font.render("Player 2:
{:02d}:{:02d}".format(p1Timemin,p1Timemsec), 1, (255, 255, 255))
            win.blit(txt, (450, 20))
        pygame.display.update()
```

```python
def startAI(self, chosed_color):
    p1Time = 60 * 15
    clock = pygame.time.Clock()
    turn = "w"
    player = chosed_color
    ai_color = "w"
    if player == "w":
        ai_color = "b"
    bo = Board(8,8, turn)
    ai = minMaxTree(bo)
    run = True
    startTime = time.time()
    while run:
        clock.tick(30)

        if turn == "w":
            if player == "w":
                p1Time -= (time.time() - startTime)
            else:
                time.sleep(1)
        else:
            if player == "b":
                p1Time -= (time.time() - startTime)
            else:
                time.sleep(1)

        startTime = time.time()

        self.redraw_gamewindow_ai(self.win, bo, p1Time, player)

        if bo.b_is_mated:
            self.end_screen(self.win, "white won!")
        if bo.w_is_mated:
            self.end_screen(self.win, "black won!")
        if bo.b_stalemate or bo.w_stalemate:
            self.end_screen(self.win, "stalemate!")

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
```

```
                quit()
                pygame.quit()

            if event.type == pygame.MOUSEMOTION:
                pass

            if event.type == pygame.K_ESCAPE:
                return

            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                i, j = self.click(pos)
                if turn == player:
                    change, _move = bo.ai_reaction_move_logic(i,
j)
                    if _move:
                        if ai.root.move:
                            if ai.root.move.end != _move.end:
                                ai.root.move = _move
                                ai.build_tree(ai.root, 3,
ai_color)
                        else:
                            ai.build_tree(ai.root, 3, ai_color)
                else:
                    is_max = True
                    if player == "w":
                        is_max = False
                    bo.ai_move_logic(ai, is_max)
                    change = True

                if change:
                    #startTime = time.time()
                    if turn == "w":
                        turn = "b"
                    else:
                        turn = "w"
                    change = False

            if event.type == pygame.KEYUP:
                bo = Board(8,8, "w")
                p1Time = 60 * 15
```

client.py

```
import pygame
from pygame.locals import *
import game
import constants as c
import os

class main_screen:
   def __init__(self):
       self.pvp_img =
pygame.transform.scale(pygame.image.load(os.path.join("Assets",
"pvp.png")), (c.BOARD_ALT_WIDTH, c.BOARD_ALT_HEIGHT/2))
       self.ai_img =
pygame.transform.scale(pygame.image.load(os.path.join("Assets",
"ai.png")), (c.BOARD_ALT_WIDTH, c.BOARD_ALT_HEIGHT/2))
       self.win = pygame.display.set_mode((c.BOARD_ALT_WIDTH,
c.BOARD_ALT_HEIGHT))
       self.clientg = game.game(self.win)
       self.pvp_hover = False
       self.ai_hover = False
       self.pvp_rect = (0,0,c.BOARD_ALT_WIDTH, c.BOARD_ALT_HEIGHT/2)
       self.ai_rect = (0, c.BOARD_ALT_HEIGHT/2,c.BOARD_ALT_WIDTH,
c.BOARD_ALT_HEIGHT/2)
       self.font = pygame.font.SysFont("Arial", 30)
       self.to_choose = False
       self.choose_txt = self.font.render("choose color!", True,
(255, 255, 255))
       self.black_txt = self.font.render("black", True, (255, 255,
255))
       self.white_txt = self.font.render("white", True, (255, 255,
255))
       self.black_txt_hitbox = (c.BOARD_ALT_WIDTH/2 - 40,
c.BOARD_ALT_HEIGHT/2 - 100, 50, 30)
       self.white_txt_hitbox = (c.BOARD_ALT_WIDTH/2 + 40,
c.BOARD_ALT_HEIGHT/2 - 100, 50, 30)
```

```
    def redraw(self, win):
        win.blit(self.pvp_img, (0,0))
        win.blit(self.ai_img, (0, c.BOARD_ALT_HEIGHT/2))
        if self.pvp_hover:
            pygame.draw.rect(win, [255,255,255], self.pvp_rect, 5)
        if self.ai_hover:
            pygame.draw.rect(win, [255,255,255], self.ai_rect, 5)
        if self.to_choose:
            win.blit(self.choose_txt, (c.BOARD_ALT_WIDTH/2 - 30,
c.BOARD_ALT_HEIGHT/2))
            win.blit(self.black_txt, (c.BOARD_ALT_WIDTH/2 - 70,
c.BOARD_ALT_HEIGHT/2 - 100))
            win.blit(self.white_txt, (c.BOARD_ALT_WIDTH/2 + 10,
c.BOARD_ALT_HEIGHT/2 - 100))
        pygame.display.update()


    def intersects(self, rect, pos):
        if pos[0] >= rect[0] and pos[0] <= rect[2] and pos[1] >=
rect[1] and pos[1] <= rect[3]:
            return True
        return False


    def start(self):
        run = True
        clock = pygame.time.Clock()
        while run:
            clock.tick(30)

            self.redraw(self.win)

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    run = False
                    quit()
                    pygame.quit()

                if event.type == pygame.MOUSEMOTION:
                    if not self.to_choose:
                        pos = pygame.mouse.get_pos()
```

```
                if self.intersects(self.pvp_rect, pos):
                    self.pvp_hover = True
                else:
                    self.pvp_hover = False
                pos = pygame.mouse.get_pos()
                if self.intersects(self.ai_rect, pos):
                    self.ai_hover = True
                else:
                    self.ai_hover = False

            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                if not self.to_choose:
                    if self.intersects(self.pvp_rect, pos):
                        self.clientg.startAI("w")
                    if self.intersects(self.ai_rect, pos):
                        self.to_choose = True
                else:
                    if self.intersects(self.white_txt_hitbox,
pos):
                        self.clientg.startAI("w")
                    if self.intersects(self.black_txt_hitbox,
pos):
                        self.clientg.startAI("b")
                    self.to_choose = False
```

main.py

```python
from client import main_screen
import pygame


def main():
    pygame.init()
    pygame.font.init()
    cl = main_screen()
    cl.start()

main()
```

constants.py

```python
import pygame, os
BOARD_ALT_ORIGINAL_WIDTH = 256
BOARD_ALT_ORIGINAL_HEIGHT = 256
SCALE_FACTOR = 3
BOARD_ALT_WIDTH = BOARD_ALT_ORIGINAL_WIDTH * SCALE_FACTOR
BOARD_ALT_HEIGHT = BOARD_ALT_ORIGINAL_HEIGHT * SCALE_FACTOR
PADDING = 18
SQUARE = 66
START_X = 120
START_Y = 120
ROWS = 8
COLS = 8
BOARD_WIDTH = COLS * SQUARE
BOARD_HEIGHT = ROWS * SQUARE
CHECKED = pygame.image.load(os.path.join("Assets", "checked.png"))
```