**עבודות גמר בjava**

**נושא הפרויקט: "Queah"**

**מגיש: נתנאל חכמון**

**ת.ז:** 324199504

**מנחה: מריו סולאי**

**תאריך הגשה: 1.5.2022**

**תוכן**

[תקציר: 3](#_Toc101822965)

[תיאור הנושא: 3](#_Toc101822966)

[רקע תאורטי: 5](#_Toc101822967)

[תיאור הבעיה האלגוריתמית: 5](#_Toc101822968)

[סקירת אלגוריתמים בתחום הבעיה: 5](#_Toc101822969)

[אסטרטגיה 6](#_Toc101822970)

[מבנה נתונים 8](#_Toc101822971)

[תרשים מחלקותUML: 10](#_Toc101822972)

[ארכיטקטורה של הפתרון בפורמט Top down level design 11](#_Toc101822973)

[תיאור סביבת העבודה ושפת התכנות 12](#_Toc101822974)

[אלגוריתם ראשי 12](#_Toc101822975)

[תיאור ממשקים מחלקות ופונקציות ראשיות בפרויקט 15](#_Toc101822976)

[התוכנית הראשית 19](#_Toc101822977)

[מדריך למשתמש 20](#_Toc101822978)

[סיכום אישי- רפלקציה 21](#_Toc101822979)

[ביבליוגרפיה 21](#_Toc101822980)

[קוד הפרויקט 22](#_Toc101822981)

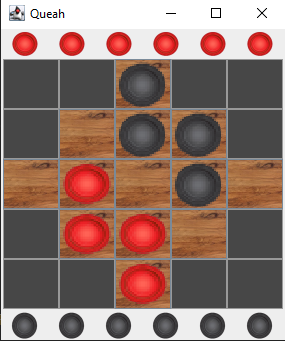
[נספחים 64](#_Toc101822982)

# תקציר:

\*הפרויקט עדין לא סופי ונשאר יעילות של negamax ובאגים.

הפרויקט שבחרתי לעשות הוא queah. זהו משחק בו ניתן לשחק מול המחשב או מול שחקן אחר. בפרויקט זה שילבתי את הנושא תכנות מונחה עצמים ומימשתי אלגוריתמים שונים. בנוסף, המשחק בנוי על אסטרטגיה שיצרתי ומאפשר לשחק בלוחות בגדלים שונים.

# תיאור הנושא:

**לוח המשחק**: המשחק משוחק על לוח מרובע משופע או אלכסוני עם 13 רווחים בלבד.

**כללי משחק הבסיסיים:**

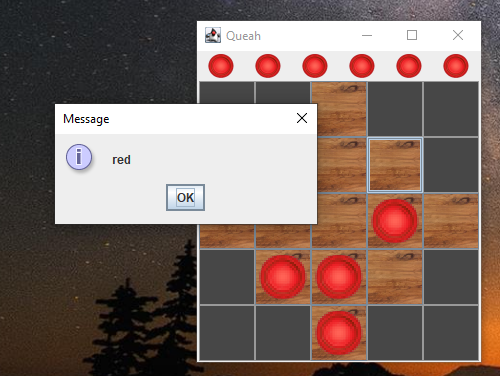
משחק ל – 2 משתתפים, 10 אבני משחק לכל משתתף 4 אבנים על הלוח.

מטרה: "לאכול" את כול האבנים של המשתתף השני.

תנועה על-גבי הלוח: כול שחקן בתור שלו יכול להזיז אבן למקום הצמוד הפנוי אל אותו אבן. שחקן יכול לאכול חתיכת אויב בקפיצה קצרה בגודל של שתיים. הכלי של השחקן חייב להיות צמוד לכלי האויב, ולנחות על שטח פנוי בצד השני. הלכידה חייבת להיעשות בכיוון אורתוגונלי בהתאם לעיצוב המלוכסן או האלכסוני של הלוח. ניתן ללכוד רק חתיכת אויב אחת בכל תור. חתיכה שנתפסה מוסרת מהלוח.

מהלך המשחק:

שחקנים מחליטים באילו צבעים לשחק, ומי מתחיל ראשון. אם כלי של שחקן נתפס, אז השחקן בתחילת התור הבא שלו חייב לקחת חתיכה אחת מהרזרבה שלו, ולשחרר אותו על כל מקום פנוי. יש להחזיר תמיד את מספר הכלים של שחקן על הלוח לארבעה, אלא אם כן השחקן מיצה את הרזרבה שלו. ששחקן יכול להוריד רק חתיכה מהרזרבה שלו, אם אחד הכלים שלהם נכבש בתורו האחרון של היריב.

**סיום המשחק:**

שחקן אדום ניצח

בגלל שלשחור לא

נישאר חיילים-

 שחקן שחור ניצח

בגלל שלאדום לא

נישאר חיילים-

# רקע תאורטי:

Liberian Queah הוא משחק אסטרטגיה מופשט לשני שחקנים מליבריה. השם הרשמי של המשחק הזה אינו ידוע, מכיוון שהוא לא צוין כאשר המשחק נכתב לראשונה. המשחק הוקלט לראשונה בשנת 1882, שחקניו היו חברים בשבט הקואה בליבריה. הלוח המסורתי עשוי מסורג של זרדים, ומקלות יוצרים את החלקים. את החלק העליון של המקלות חותכים בצורה מלוכסנת מצד אחד, ונקראים "גברים", ואילו המקלות של הצד השני נחתכים ישר ונקראים "נשים".

# תיאור הבעיה האלגוריתמית:

במהלך העבודה על הפרויקט נתקלתי במספר בעיות אלגוריתמיות אותן הייתי צריך לפתור כדי שהמערכת תעבוד כמו שצריך:

בדיקת תקינות של המהלך - כאשר המשתמש מבצעה מהלך המחשב בודק אם הוא תקין.

בניית אלגוריתם לשחקן ממוחשב – ליצור שחקן אשר פועל לפי המהלכים של היריב. יש למחשב אלגוריתם אשר סורק את הלוח שומר את כול המהלכים האפשריים לכול חייל וגם נותן משקל לכול אחד מהחלקים שלו.

מציאת כל המהלכים האפשריים לחייל מסוים- מציאת כל האפשרויות למהלכים עבור אחד החיילים ממצב לוח מסוים.

# סקירת אלגוריתמים בתחום הבעיה:

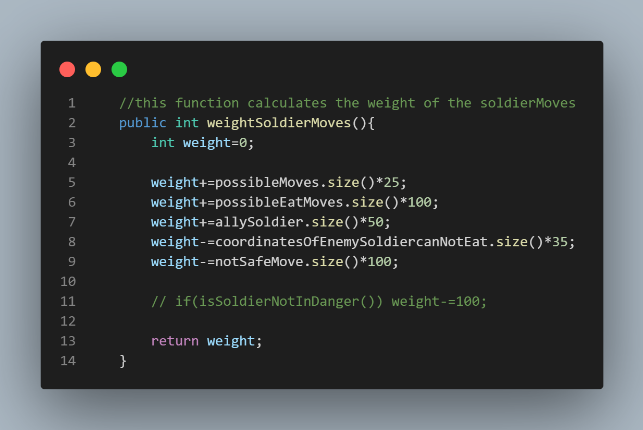
בדיקת תקינות של המהלך - הבדיקות תקינות שהיו הם: אם השחקן מזיז את החייל שלו או שם חייל חדש. אם הוא מזיז אז אם הוא ביצעה אכילה או הזזה רגילה. אם זה אכילה(כאשר יש עד 4 אפשרויות של אכילה. אכילה מתבצעת בקפיצה קצרה בגודל של שתיים ובאמצע יש את היריב) אז האם האכילה הייתה חוקית ואם זה הזזה(כאשר יש עד 4 אפשרויות של הזזה. הזזה מתבצעת בכך שצריך להזיז אבן למקום הצמוד הפנוי אל אותו אבן) אז האם הזזה הייתה חוקית. השחקן שם חייל חדש(בכול מקום רק בלוח) רק אם חייל שלך נאכל בתור הקודם וגם אם נישאר לך חיילים שאתה יכול לשים.

בניית אלגוריתם לשחקן ממוחשב – למחשב יש 3 אפשריות רמה שונות בכול האפשרויות האופציה הראשונה תהייה תמיד לאכול אחר כך אם זה רמה קלה אז הוא יבדוק אם יש חיילים שעומדים להיאכל ואם כן אז הוא יזיז את החייל לאחד מהאפשרויות הזזה הרנדומליות שיש לחייל אם לא אז הוא יבחר חייל בצורה רנדומלית ומזיז אותו לאחד מהאפשרויות הרנדומליות. ברמה בינונית הוא יבדוק אם יש חיילים שעומדים להיאכל ואם כן אז הוא יזיז את החייל לאפשרות הזזה אם המשקל הגבוה ביותר אם אין חייל שעומד להיאכל אז הוא מחשב לכול אחד מהחיילים את המשקל שלו ולוקח את החייל אם המשקל הגבוה ביותר ומזיז אותו לאפשרות אם המשקל הגבוה ביותר. המשקל מתחשב במה שקורה מסביב לחייל ב 2 בלוקים לכול כיוון ( למעלה למטה ימינה ושמאלה ). כאשר זה הרמה הקשה החייל משתמש באלגוריתם Negamax ובוחר את ההזזה לפי אלגוריתם זה.

מציאת כל המהלכים האפשריים לחייל מסוים - מציאת כל האפשרויות למהלכים עבור אחד החיילים ממצב לוח מסוים בכך שמקבליים את קואורדינטות של חייל וזה והאלגוריתם מסורק 2 בלוקים לכול כיוון ( למעלה למטה ימינה ושמאלה ) ובודק אם יש אכילה , הזזה או אין כלום. הוא שומר את כול ההזזות האפשריים ברשימה וכך גם האכילות האפשריים. וזה בודק גם אם החייל יכול להיאכל על ידי היריב.

# אסטרטגיה

האסטרטגיה מתחלקת לכמה חלקים שהם:

1. דירוג החייל של השחקן – דירוג זה מתבצע על מתן נקודות לפי פרמטרים וחישובם כך. כמות אפשרויות הזזה של החייל כפול 25 + כמות חיילים של היריב אשר צמודים לחייל ואפשר לאכול אותם כפול 100 + כמות החיילי בירית הצמודים לחייל כפול 50 – כמות חיילים של היריב אשר צמודים לחייל ואי אפשר לאכול אותם כפול 35 – כמות המהלכים של החייל שיגרמו לו להיאכל על ידי היריב.
2. דירוג הלוח של שחקן – הדירוג מתבצע על ידי חישוב וחיבור הדירוג של כול אחד מהחיילים של השחקן.



1. דירוג הלוח עצמו - הדירוג מתבצע על ידי מתן נקודות לשחקן וליריב שלו על ידי פרמטרים זהים ובסוף להחזיר את ההפרש. פרמטרים הם הדירוג של הלוח השחקן ,כמות הכלים של השחקן , הדירוג של הלוח של היריב ו כמות הכלים של היריב.



1. הוספת חייל חדש – כול אחד מהמקומות בלוח מדורג אם ניקוד קבועה וכאשר מוסיפים חייל חדש עדיף לשים את החייל במיקום הפנוי אם הניקוד הגבוה ביותר.



# מבנה נתונים

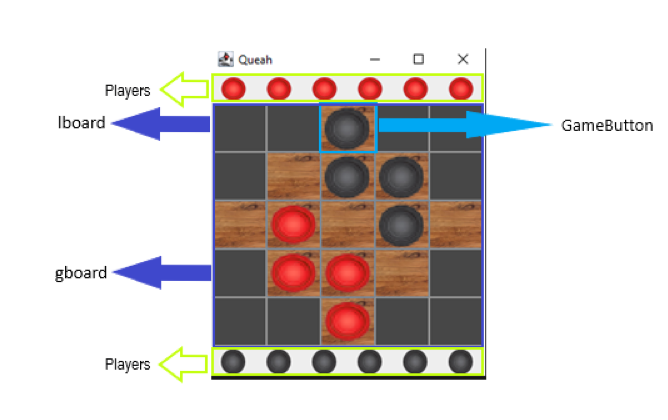
בפרויקט יש כמה מבני נתונים שונים ולכול אחד יש מטרה שונה:

1. יש מטריצה דו ממדית דינמית(שלוש אופציות: קטן ,בינוני וגדול) של int בשם lboard והיא שומרת בכול אחד מהמשתנים במערך את האופציות האלה(1- אם זה לא פעיל/לא קשור ללוח , 0 משבצת ריקה , 1 משבצת אם חייל אדום ו 2 משבצת אם חייל שחור)
2. יש מטריצה דו ממדית דינמית(שלוש אופציות: קטן ,בינוני וגדול) של GameButton(מחזיק תמונה וכפתור) בשם gboard שהיא אחראית לשמירה והצגה גרפית של הלוח.

1. מערך חד ממדי בגודל 8 של int בשם test שימושו היא בשליחת מידע ועדכון מידה כאשר המחשב מבצעה מהלך. כול תא אומר כך.

[newRow][ newColumn][ previsRow][ previsColumn][ eatRow][ eatColum][ isEat][ isSoldierLeft]

1. מחלקה בשם Coordinate אשר שומרת שלוש משנים של int בשם row, column, value. והמטרה שלה היא לשמור את השורה, עמודה ומידע.
2. יש מחלקה בשם SoldierMoves אשר ובתוכו יש מבנה נתונים הללו
   * 1. רשימה של Coordinate בשם possibleMoves ששומרת את כול ההזזות החוקיות של החייל.
     2. רשימה של Coordinate בשם possibleEatMoves ששומרת את כול האכילות החוקיות של החייל.
     3. רשימה של Coordinate בשם notSafeMove ששומרת את כול ההזזות החוקיות אך יגרמו לחייל להיאכל בתור הבא.
     4. רשימה של Coordinate בשם allySoldier ששומרת את כול חיילי ברית אשר צמודים לחייל.
     5. רשימה של Coordinate בשם coordinatesOfEnemySoldiercanNotEat ששומרת את כול החיילים של היריב הצמודים לחייל ואי אפשר לאכול אותם.

כך מתבטאים מבני הנתונים השייכים לגרפיקה.

# תמונה שמכילה טקסט, צילום מסך, מקורה התיאור נוצר באופן אוטומטיתרשים מחלקותUML:

# ארכיטקטורה של הפתרון בפורמט Top down level design

* התוכנית הראשית Game מכילה 3 פאנלים: Bottom, Center, Top .
* Top ו Bottom מכילים את הפאנל Players/ Computer.
  + המחלקה Computer יורשת ממחלקה Players ומוסיפה פונקציות אשר מאפשרות למחשב לשחק ולחשוב. המחלקה משתמשת במחלקות הללו.
    - soldierMovesStack - זהו מחסנית של SoldierMovesשימוש המחלקה היא לשמור ולסרוק את המהלכים של חייל היא משתמשת במשתנים והפונקציות ההלו.
      * SoldierMoves משתמש במחלקה Coordinate אשר מוזכרת בסעיף "מבנה נתונים" מס' 4.
      * המשנים מוסברים בהרחבה בסעיף "מבנה נתונים" מס' 5.
      * הפונקציות מוסברות בהרחבה בסעיף "הפונקציות/ המחלקות הראשיות בפרויקט".
    - Computer משתמש גם במחלקה Negamax. הפונקציות והמשתנים מוסברים בהרחבה בסעיף "הפונקציות/ המחלקות הראשיות בפרויקט".
    - שאר הפונקציות והמשתנים מוסברים בהרחבה בסעיף "הפונקציות/ המחלקות הראשיות בפרויקט".
  + המחלקה והפאנל Players(תפקידה של המחלקה היא לשמור את כמות החיילים של השחקן על הלוח וגם את כמות החיילים שנמצאים ברזרבה) מכילה את המשתנים והפונקציות הללו.
    - soldier\_left – מספר החיילים שנשארו.
    - soldier\_on\_board – מספר החיילים בלוח.
    - player\_color – צבע השחקן.
    - Copy – פונקציה שמעתיקה שחקן.
    - removeSoldierFromStack – פונקציה שמורידה חייל מהרזרבה גם בצורה גרפית
    - removeSoldierFromBoard – פונקציה שמורידה מכמות החיילים בלוח.
    - addSoldierToBoard – הפונקציה מוספיה שחקן לכמות השחקנים שבלוח.
* Center מכיל את הפאנל QueahBoard שמטרתו היא להיות לוח המשחק. המחלקה מכילה מספר פונקציות ומשתנים.
  + lBoard – מערך דו ממדי של מספרים שלמים ומטרתו לשמור בכול אחד מהמשתנים במערך את האופציות האלה (1- אם זה לא פעיל/לא קשור ללוח , 0 משבצת ריקה , 1 משבצת אם חייל אדום ו 2 משבצת אם חייל שחור).
  + gboard - מערך דו ממדית דינמית של GameButton שמטרתו להיות אחראית לשמירה והצגה גרפית של הלוח.
    - GameButton זה מחלקה אשר משמשת ככפתור. המחלקה משתמשת במשתנים הללו.
      * img – זהו תמונה של הכפתור.
      * weight – זהו המשקל של הכפתור (בשביל המחשב).
      * soldier – זהו מחלקה בשם Soldier שמטרתה לייצג חייל. היא משתמשת במשתנים הללו.
        + img – זה התמונה של החייל.
        + color – זה מספר אשר מיצג את הצבע של החייל.
  + שאר הפונקציות והמשתנים מוסברות בהרחבה בסעיף "הפונקציות/ המחלקות הראשיות בפרויקט".

# תיאור סביבת העבודה ושפת התכנות

שפת התכנות היא: .java

שימוש מינימלי בשפת תגיות: .html

ספריות: util, awt, swing ,file.

סביבת העבודה היא: .visual studio code

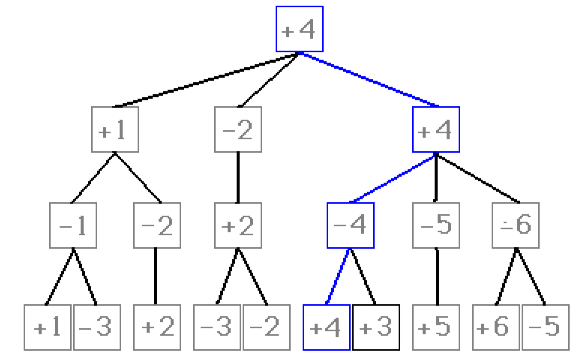
מפרט תכני:

* מעבד: .Intel(R) Core(TM) i5-5300U CPU 2.30GHz
* זיכרון: 8 GB RAM.
* סביבת עבודה: windows 10.

# אלגוריתם ראשי

האלגוריתם הראשי הוא האלגוריתם למציאת המהלך הטוב ביותר, בעזרת האלגוריתם negamax. האלגוריתם מוצא את התור הבא הטוב ביותר בעזרת מציאת כל התורות הבאים האפשריים ומוצא את התוצאה הטובה ביותר. בעזרת האלגוריתם אפשר גם לבדוק את כמה הצעדים הבאים על ידי מציאת את כל הצעדים האפשריים עבור הלוחות שמתקבלים לאחר חישוב כל הצעדים האפשריים בלוח הקודם.

האלגוריתם Negamax מיוצג בעזרת עץ משחק. האלגוריתם סורק את כל המצבים האפשריים, עד לעומק מסוים אותו נבחר. שורש העץ מייצג את הלוח הקיים, לפני ביצוע מהלך כלשהו. ברמה הבאה יופיעו כל מצבי הלוח, עליו בוצע מהלך אחד קדימה, וכך הלאה. כל עלי העץ יקבלו ציון, לפי הפעולה, evaluate שנותנת ציון ללוח. ברמה שבודקת מהלכים של השחקן הממוחשב, נבחר באופציה בעל הציון הגבוה ביותר, המהלך הטוב ביותר עבור המחשב. ורמה שבודקת מהלכים עבור השחקן האנושי, נבחר את האופציה בעל הדירוג הנמוך ביותר, משמע האופציה הכי פחות טובה לשחקן הממוחשב.

כדי להשתמש באלגוריתם זה, נכפול במינוס אחד (1-), את התוצאה שחוזרת ברקורסיה מה- negamax .תמיד נחזיר את הציון בעל הערך הגבוה ביותר, וכך בתור השחקן האנושי נחזיר את התוצאה בעל הערך המוחלט הקטן ביותר.

דוגמא לעץ negamax:

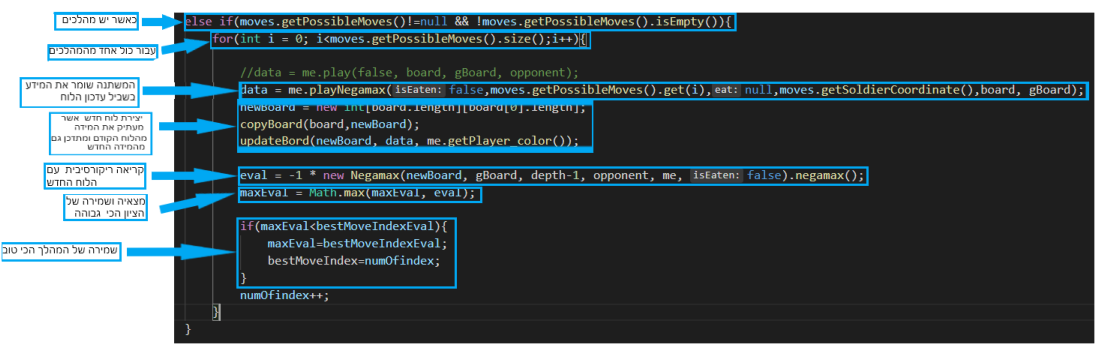
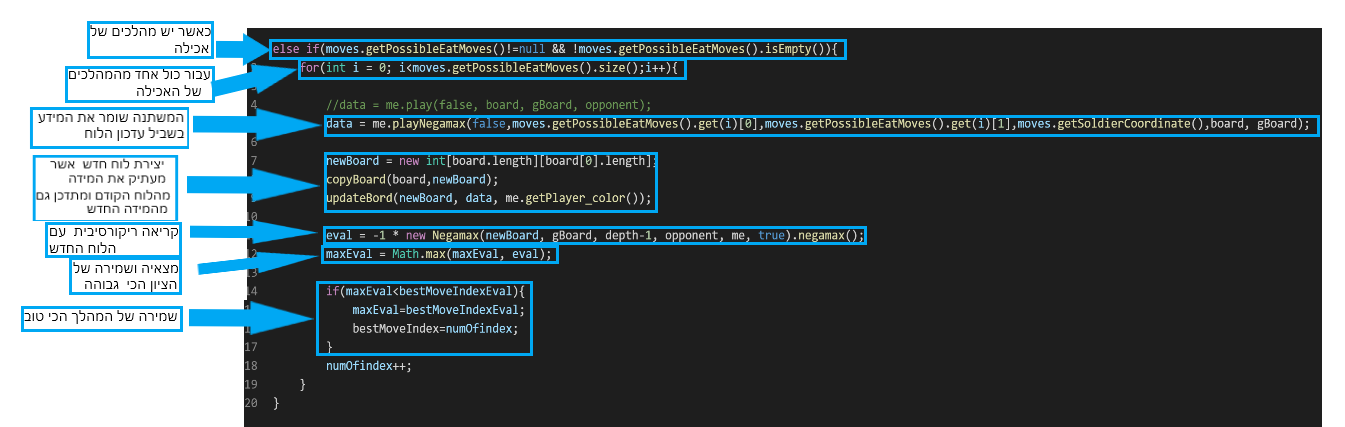
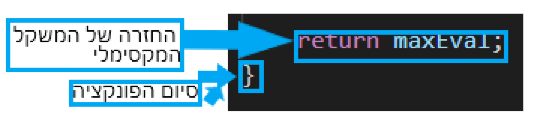
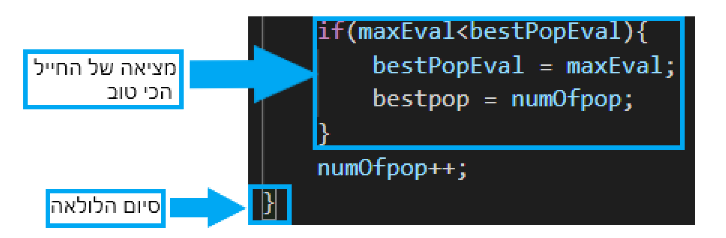
הפונקציה negamax בפרויקט:



תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט, צילום מסך, צג, מסך

התיאור נוצר באופן אוטומטי 

# תיאור ממשקים מחלקות ופונקציות ראשיות בפרויקט

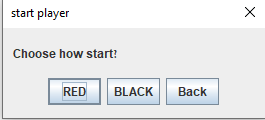
|  |  |  |  |
| --- | --- | --- | --- |
| מחלקה | משתנה/פעולה | הסבר | יעילות |
| Game | int turn | שומר את הטור(1 אדום 2 שחור) | - |
| int gameMode | שומר את סוג המשחק (0 שחקן נגד שחקן 1 שחקן נגד מחשב 2 מחשב נגד מחשב) | - |
| int gameStart | שומר איזה שחקן מתחיל | - |
| int difficultyComputer1  int difficultyComputer2 | שומר את הרמת קושי של המחשבים | - |
| Players playerRed  Players playerBlack | שומר את השחקן האדום והשחור | - |
| Computer computerRed  Computer computerBlack | שומר את המחשב האדום והשחור | - |
| QueahBoard board | הלוח משחק | - |
| String map | שומרת את סוג המפה של המשחק | - |
| public Game() | פעולה בונה שמתחלה את המשחק ומשתמשת בפונקציות אחרות כדי לאתחל ממשתנים | O(1) |
| private static String getFileInfo() | פונקציה שקוראת את הקובץ של החוקים ומחזירה אותו. | O(1) |
| private void menuMap() | פונקציה שמציגה את האופציות של המפות ומאתחלת את | O(1) |
| private void startMap() | פונקציה שמציגה את האופציות של איזה מהשחקנים מתחיל | O(1) |
| private void gameModeMenu() | פונקציה שמציגה את האופציות של איזה סוג משחק. | O(1) |
| private void difficultyComputer1()  private void difficultyComputer2() | פונקציות שמציגות את האופציות של איזה קושי כול מחשב יכול להיות | O(1) |
| private void mapSolid(QueahBoard board) | פונקציה שמקבלת את הלוח ומעדכנת לפי המפה את הלוח. | O(1) |
| QueahBoard | int [][]lBoard | מערך דו מימדי של מספרים שלמים אשר מיצג את הלוח הלוגי | - |
| GameButton [][]gBoard | מערך דו מימדי של כפתורים אשר מיצגת את הלוח הגרפי של המשחק | - |
| public QueahBoard(Game game) | פונקציה בונה שמקבלת את המחלקה game ומאתחלת את הנתונים | O(n)  n=row\*column |
| public void initBoard() | פונקציה מאפסת את הלוחות(הגרפי והלוגי)  ומוסיפה במקום המתאים את החיילים בצורה לוגית וגרפית  וגם מוסיפה משקל. | O(n)  n=row\*column |
| private void addWeight() | הפונקציה מוסיפה את המשקלים ללוח. | O(n)  n=row\*column |
| public void victory(int player) | הפונקציה מקבלת את השחקן ומציגה הודעה שאומרת שהשחקן ניצח | O(1) |
| public void moveSoldier() | הפונקציה מזיזה חייל | O(1) |
| public void removeSoldier(int row, int column) | הפונקציה מקבלת את השורה והעמודה ומסירה חייל מהלוח | O(1) |
| public void addSoldierToBoard() | הפונקציה מוספיה חייל ללוח | O(1) |
| public void actionPerformed(ActionEvent e) | הפונקציה מקבלת ActionEvent ומחכה ללחיצת כפתור כאשר נלחץ הכפתור היא בודרת מי לחץ ובאיזה סוג משחק אנכנו אם אנכנו בשחקן נגד שחקן היא מפעילה את הפונקציה של השחקן אם זה שחקן נגד מחשב היא בודקת טור מי זה אם זה טור השחקן היא מפעילה את הפונקציה של השחקן אם זה טור של המחשב אז היא מזיזה אותו | O(1) |
| private void HumanMove() | הפונקציה אחראית לזה לבדיקת הפעולה של השחקן אם היא חוקית היא משנה אותו בלוח הלוגי והגרפי | O(1) |
| private void ComputerMove(boolean isEaten) | הפונקציה אחראית לזה שהמחשב יזוז גרפית ולוגית והיא מקבלת משתנה בוליאני שאומר אם נאכל חייל של המחשב | O(1) |
| Computer | public Computer(int player\_color, String map,int difficulty) | הפונקציה הבונה מאתחלת מישתנים.  היא מקבלת את הצבע של המחשב את סוג המפה ואת קושי של המחשב | O(1) |
| public int[] play(boolean isEaten,int [][]lBoard,GameButton [][]gBoard, Computer enamy) | הפונקציה מחשבת ומחזירה מידה שאומר אך למחשב רוצה להזיז את החייל.  היא מקבל משתנה בוליאני שאומר אם האחל למחשב חייל הוא מקבל גם את הלוח הלוגי והגרפי וגם הוא מקבל את היריב שלו. | O(n)  n= row\*column |
| private int[] addNewSolid() | הפונקציה מחשבת ומחזירה למחשב את המקום הטוב ביותר לשים שחקן חדש | O(n)  n= row\*column |
| private int[] move() | הפונקציה מחשבת לפי הרמת קושי של המחשב את המהלך שהוא צריך לעשות ומחזיר את המהלך הזה. | O(n)  n= all item in negamax tree |
| private SoldierMoves findBestMove(Stack<SoldierMoves> soldierMovesStack)  private SoldierMoves findBestEat(Stack<SoldierMoves> eatSoldierMovesStack) | הפונקציות מקבלות את המחסנית של האכילה/הזזה ומחזירות את החייל אם הזזות הטובות ביות | O(n)  n= all item in the stack |
| private int indexOfBestMove(List<Coordinate> possibleMoves)  private int indexOfBestEat(List<Coordinate[]> possibleEatMoves) | הפונקציה מחזירה את המיקום ברשימה של האכילה/הזזה של החייל שבוא המשקל הוא הטוב ביותר. | O(n)  n= all item in the list |
| private void findAllPossibleSoldier() | הפונקציה מחפשת את כול החיילים של השחקן שנמצאים על הלוח. | O(n)  n= row\*column |
| private int[] findMostWeightBlock() | הפונקציה מחזירה את המיקום אם המשקל הכי גבוה בשביל לשים חייל חדש. | O(n)  n= row\*column |
| SoldierMoves | public SoldierMoves(int [][]lBoard,GameButton [][]gBoard,Coordinate soldierCoordinate) | הפונקציה בונה ומאפסת את כול המשתנים. | O(1) |
| public void scannMap() | הפונקציה סורקת את הלוח ומכניסה למישתנים את הנתונים שלהם. (המישתנים מתוארים בסעיף "מיבנה נתונים" מס' 5 ) | O(1) |
| public void findPossibleMoves()  public void findPossibleEatMoves()  public void findCoordinatesOfEnemySoldiercanNotEat()  public void findAllySoldier()  public void findMoveNotSafe() | הפונקציות מחפשות לפי המישתנים של הסריקה של המפה את(לפי שם בפונקציה)  ומעכן את הרשימה המתאימה) | O(1) |
| public boolean isSoldierStuck() | בודק אם חייל תקועה ומחזיר את התשובה. | O(1) |
| public boolean isSoldierNotInDanger() | בודק אם חייל לא נימצה בסכנה. ומחזיר את התשובה | O(1) |
| public int weightSoldierMoves() | הפונקציה מחשבת ומחזירה את המישקל של החייל. | O(1) |
| Negamax | public Negamax(int[][] board,GameButton [][]gBoard, int depth,Computer me, Computer opponent, boolean isEaten) | הפונקציה בנוה מקבלת ממשתנים ומעדכנת אותם במחלקה. | O(n)  n=row\*column |
| public boolean gameOver() | הפונקציה מחזירה משתנה בוליאני אם נגמר המשחק. | O(1) |
| private void updateBord(int[][] board,int[] data, int player\_color) | הפונקציה מקבלת את הלוח,מידע,את צבע השחקן ומעדכנת את הלוח שקיבלה. | O(1) |
| private int evaluateBordByPlayer(Computer computer) | הפונקציה מחזירה את המשקל של המחשב שהיא קיבלה. | O(n)  n=the number of item in the stack |
| private int evaluate() | הפונקציה מחזירה את המשקל של הלוח. | O(n)  n=the number of item in the stack |
| public int negamax() | פונקציה רקורסיבית שעובדת לפי(ההסברת בסעיף האלגוריתם הראשי) ומחזירה את המשקל. | O(n)  n=מספר האיברים בעץ |

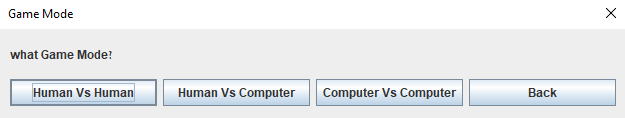
# התוכנית הראשית

התוכנית הראשית מייצרת משחק חדש. ואחראית לכול האופציות לפני המשחק עצמו.

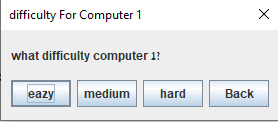
# מדריך למשתמש

הלוח הראשון שנפתח עם הרצת התוכנית הוא לוח בחירת המפה והחוקים של המשחק.

כאשר לוחצים על אחד מהמפות נפתח לוח של בחירת השחקן שמתחיל.

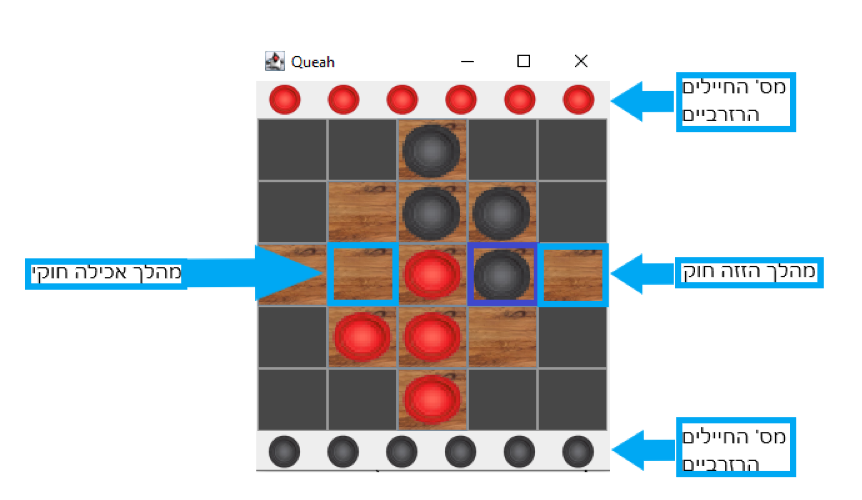
כאשר לוחצים על אחד השחקנים נפתח לוח של בחירת סוג המשחק.

כאשר לוחצים על שחקן נגד שחקן מתחיל המשחק. אם נלחץ הכפתור של שחקן נגד מחשב או מחשב נגד מחשב נפתח חלון של בחירת רמת הקושי של המחשב.



כאשר בוחרים את הרמת קושי נפתח המשחק.

בשביל להזיז חייל השחקן שתורו בוחר את החייל שהוא רוצה להזיז בלחיצה עליו. ואז בשביל להזיז אותו השחקן לוחץ על המקום שהוא רוצה שהחייל יזוז. אם זה לא חוקי השחקן יצטרך ללחוץ שוב פעם על החייל ואז אז מקום אחר.(אותו דבר גם באכילה).



# סיכום אישי- רפלקציה

העבודה על הפרויקט הייתה מסע ארוך מעניין וכיף במיוחד עבורי. במהלך העבודה הייתי צריך לפתור המון בעיות, לבצע ניסיונות ובדיקות ופשוט לעבוד בצורה רפטטיבית ומעמיקה כדי להגשים את החזון שתכננתי. אני מרגיש שקיבלתי מהעבודה על הפרויקט הזה המון כלים, רובם בתחום התכנות כמו הרחבת הידע שלי, כתיבה מסודרת, תכנון, יעילות, שימוש ב git לגיבוי, עמידה בזמנים, תיעוד קוד, אלגוריתמים חדשים כמו negamax ושימוש בגרפיקה ב java. היה הרבה קשיים כמו לכתוב מחלקה ואלגוריתם מחדש בגלל שהוא היה לא קריא ולא יעיל, למצוא באגים וקריסות קוד במשך שעות. אם הייתי עושה את הפרויקט היום הייתי משקיעה יותר חשיבה בלתכנן ולשמור אל עיצוב תבניות בקוד ועקרונות עיצוב בקוד. אני חושב גם שהייתי משקיעה יותר מחשבה בגרפיקה מכיוון שהגרפיקה לא כזה יפה וגם אן אנימציות. גרפיקה זה הדבר הראשון שמושך אנשים למשחק כי לפני עוד שמשחקים אנשים רואים ושופטים לפי הגרפיקה. המשחק שעשיתי היה מענין אך גיליתי שהוא לא מוכר ואן הרבה מקורות מידע עליו ולכן גם יש הרבה בעיות עם החוקים כמו שאן תיקו והמשחק יכול להימשך לניצח וגם האסטרטגיה לא מסובכת יחסית למשחקים אחרים בגלל שהוא פשוט ולא אופטימלי. אני גם חושב ש java לא אופטימלית ליציקת משחקים הללו כי חסר לה הרבה אפשרויות מכון שהספרייה מותאמת לאפליקציות ולא למשחק ולכן היה לפי דעתי לעשות את המשחק אם ספריה/שפה אחרת ולהשתמש גם במנועה משחק אשר מאפשר הרבה אופציות הקשורות למשחקים.

# ביבליוגרפיה

1. <https://www.javatpoint.com/java-swing>
2. <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>
3. <https://www.javatpoint.com/java-awt>
4. <https://stackoverflow.com/questions/65750233/what-is-the-difference-between-minimax-and-negamax>
5. <https://www.youtube.com/watch?v=l-hh51ncgDI&t=3s>

# קוד הפרויקט

package code**;**

**import** java**.**awt**.\*;**

**import** java**.**io**.**IOException**;**

**import** java**.**nio**.**file**.\*;**

**import** javax**.**swing**.**JFrame**;**

**import** javax**.**swing**.**JOptionPane**;**

**import** javax**.**swing**.**SwingUtilities**;**

public class Game **extends** JFrame **{**

public String map**;**

public int turn**;**

public int gameMode**=**0**;**

public int gameStart**=**1**;**

private int difficultyComputer1**;** // 0 easy | 1 medium | 2 hard

private int difficultyComputer2**;** // 0 easy | 1 medium | 2 hard

public Players playerRed**;**

public Players playerBlack**;**

public Computer computerRed**;**

public Computer computerBlack**;**

private QueahBoard board**;**

public Game**()** **{**

menuMap**();**

mapSolid**(**board**);**

add**(**board**,**BorderLayout**.**CENTER**);**

setTitle**(**"Queah"**);**

setDefaultCloseOperation**(**EXIT\_ON\_CLOSE**);**

setSize**(**300**,**350**);**

setVisible**(true);**

setLocationRelativeTo**(null);**

**}**

private static String getFileInfo**(){**

String path **=** "././files/Game\_Play\_and\_Rules.txt"**;**

**try** **{**

String content **=** Files**.**readString**(**Paths**.**get**(**path**));**

**return** content**;**

**}** **catch** **(**IOException e**)** **{**

e**.**printStackTrace**();**

**}**

**return** **null;**

**}**

private void menuMap**(){**

String**[]** options **=** **{**"small"**,** "mid"**,** "large"**,**"Rules"**,**"Exit"**};**

int response **=** JOptionPane**.**showOptionDialog**(null,** "Choose Type map"**,**

"Starting map Options"**,**

JOptionPane**.**DEFAULT\_OPTION**,** JOptionPane**.**PLAIN\_MESSAGE**,**

**null,** options**,** options**[**0**]);**

**switch(**response**)**

**{**

**case** **-**1**:**

System**.**out**.**println**(**"map Dialog Window Was Closed"**);**

System**.**exit**(**0**);**

**case** 0**:**

map **=** "small"**;**

startMap**();**

**break;**

**case** 1**:**

map **=** "mid"**;**

startMap**();**

**break;**

**case** 2**:**

map **=** "large"**;**

startMap**();**

**break;**

**case** 3**:**

Runnable rule **=** **()** **->** **{**String html **=(**"<html><body width='%1s'><h1>Rules</h1><p>"**+**getFileInfo**());**

JOptionPane**.**showMessageDialog**(**Game**.this,** String**.**format**(**html**,** 500**,** 500**));**

**};**

SwingUtilities**.**invokeLater**(**rule**);**

menuMap**();**

**break;**

**case** 4**:**

System**.**exit**(**0**);**

**default:**

**break;**

**}**

**}**

private void startMap**(){**

String**[]** options **=** **{**"RED"**,** "BLACK"**,**"Back"**};**

int response **=** JOptionPane**.**showOptionDialog**(null,** "Choose how start?"**,**

"start player"**,**

JOptionPane**.**DEFAULT\_OPTION**,** JOptionPane**.**PLAIN\_MESSAGE**,**

**null,** options**,** options**[**0**]);**

**switch(**response**)**

**{**

**case** **-**1**:**

System**.**out**.**println**(**"map Dialog Window Was Closed"**);**

System**.**exit**(**0**);**

**case** 0**:**

turn **=** 1**;**

gameModeMenu**();**

**break;**

**case** 1**:**

turn **=** 2**;**

gameModeMenu**();**

**break;**

**case** 2**:**

menuMap**();**

**default:**

**break;**

**}**

**}**

private void gameModeMenu**(){**

String**[]** options **=** **{**"Human Vs Human"**,** "Human Vs Computer"**,**"Computer Vs Computer"**,**"Back"**};**

int response **=** JOptionPane**.**showOptionDialog**(null,** "what Game Mode?"**,**

"Game Mode"**,**

JOptionPane**.**DEFAULT\_OPTION**,** JOptionPane**.**PLAIN\_MESSAGE**,**

**null,** options**,** options**[**0**]);**

**switch(**response**)**

**{**

**case** **-**1**:**

System**.**out**.**println**(**"gameModeMenu Dialog Window Was Closed"**);**

System**.**exit**(**0**);**

**case** 0**:**

gameMode**=**0**;**

playerRed **=** **new** Players**(**1**,**map**);**

playerBlack **=** **new** Players**(**2**,**map**);**

board **=** **new** QueahBoard**(**Game**.this);**

add**(**playerRed**,**BorderLayout**.**NORTH**);**

add**(**playerBlack**,**BorderLayout**.**SOUTH**);**

**break;**

**case** 1**:**

gameMode **=** 1**;**

difficultyComputer2**();**

playerRed **=** **new** Players**(**1**,**map**);**

computerBlack **=new** Computer**(**2**,**map**,**difficultyComputer2**);**

board **=** **new** QueahBoard**(**Game**.this);**

add**(**playerRed**,**BorderLayout**.**NORTH**);**

add**(**computerBlack**,**BorderLayout**.**SOUTH**);**

**break;**

**case** 2**:**

gameMode **=** 2**;**

difficultyComputer1**();**

computerRed **=** **new** Computer**(**1**,**map**,**difficultyComputer1**);**

computerBlack **=** **new** Computer**(**2**,**map**,**difficultyComputer2**);**

board **=** **new** QueahBoard**(**Game**.this);**

add**(**computerRed**,**BorderLayout**.**NORTH**);**

add**(**computerBlack**,**BorderLayout**.**SOUTH**);**

**break;**

**case** 3**:**

startMap**();**

**break;**

**default:**

**break;**

**}**

**}**

private void difficultyComputer1**(){**

String**[]** options **=** **{**"eazy"**,** "medium"**,**"hard"**,**"Back"**};**

int response **=** JOptionPane**.**showOptionDialog**(null,** "what difficulty computer 1?"**,**

"difficulty For Computer 1"**,**

JOptionPane**.**DEFAULT\_OPTION**,** JOptionPane**.**PLAIN\_MESSAGE**,**

**null,** options**,** options**[**0**]);**

**switch(**response**)**

**{**

**case** **-**1**:**

System**.**out**.**println**(**"difficulty Dialog Window Was Closed"**);**

System**.**exit**(**0**);**

**case** 0**:**

difficultyComputer1 **=** 0**;**

difficultyComputer2**();**

**break;**

**case** 1**:**

difficultyComputer1 **=** 1**;**

difficultyComputer2**();**

**break;**

**case** 2**:**

difficultyComputer1 **=** 2**;**

difficultyComputer2**();**

**break;**

**case** 3**:**

gameModeMenu**();**

**break;**

**default:**

**break;**

**}**

**}**

private void difficultyComputer2**(){**

String**[]** options **=** **{**"eazy"**,** "medium"**,**"hard"**,**"Back"**};**

int response **=** JOptionPane**.**showOptionDialog**(null,** "what difficulty computer 2?"**,**

"difficulty For Computer 2"**,**

JOptionPane**.**DEFAULT\_OPTION**,** JOptionPane**.**PLAIN\_MESSAGE**,**

**null,** options**,** options**[**0**]);**

**switch(**response**)**

**{**

**case** **-**1**:**

System**.**out**.**println**(**"difficulty Dialog Window Was Closed"**);**

System**.**exit**(**0**);**

**case** 0**:**

difficultyComputer2 **=** 0**;**

**break;**

**case** 1**:**

difficultyComputer2 **=** 1**;**

**break;**

**case** 2**:**

difficultyComputer2 **=** 2**;**

**break;**

**case** 3**:**

**if(**gameMode **==** 1**)** gameModeMenu**();**

**else** difficultyComputer1**();**

**break;**

**default:**

**break;**

**}**

**}**

private void mapSolid**(**QueahBoard board**){**

**switch** **(**map**)** **{**

**case** "small"**:**

board**.**smallMapSolid**();**

**break;**

**case** "mid"**:**

board**.**midMapSolid**();**

**break;**

**case** "large"**:**

board**.**largeMapSolid**();**

**break;**

**default:**

board**.**smallMapSolid**();**

**break;**

**}**

**}**

public static void main**(**String**[]** args**)**

**{**

**new** Game**();**

**}**

**}**

package code**;**

**import** java**.**awt**.\*;**

**import** java**.**awt**.**event**.\*;**

**import** java**.**io**.**IOException**;**

**import** java**.**nio**.**file**.**Files**;**

**import** java**.**nio**.**file**.**Paths**;**

**import** javax**.**swing**.\*;**

public class QueahBoard **extends** JPanel **{**

//img

private static final String IMG\_BOARD**=**"././img/Wood.png"**;**

ImageIcon icon **=** **new** ImageIcon**(**IMG\_BOARD**);**

Image img **=** icon**.**getImage**();**

private static int startPlayer**=**1**;**

private boolean isFirstCOMvsCOM**=true;**

private static final int THINKING\_TIME**=**500**;**

private static int sizeOfboard**=**5**;**

private static int heightOfboard**=**2**;**

private static int max\_Player\_soldiers\_on\_board**=**4**;**

private static int turn**;**// 1 red 2 black

private GameButton **[][]**gBoard**;** //graphic board

private int **[][]**lBoard**;** //logic board, 0 free 1 red player 2 black Player

private Soldier redSoldier **=** **new** Soldier**(**"red"**);**

private Soldier blackSoldier **=** **new** Soldier**(**"black"**);**

private Players playerRed**;**

private Players playerBlack**;**

private Computer computerRed**;**

private Computer computerBlack**;**

private Game game**;**

public QueahBoard**(**Game game**)** **{**

**this.**game**=** game**;**

startPlayer**=**game**.**turn**;**

constrictGamMode**();**

constrictorMap**(**game**.**map**);**

initBoard**();**

**}**

//if game mode is 0 red plater and black player is Human

//if game mode is 1 red player is Human and black player is computer

//if game mod is 2 red player and black player is computer

private void constrictGamMode**(){**

**switch** **(**game**.**gameMode**)** **{**

**case** 1**:**

computerBlack **=** game**.**computerBlack**;**

playerBlack **=** game**.**computerBlack**;**

playerRed **=** game**.**playerRed**;**

computerRed **=** **new** Computer**(**playerRed**.**getPlayer\_color**(),**playerRed**.**getMap**(),**computerBlack**.**getDifficulty**());**

computerRed.copy(playerRed);

break;

case 2:

computerRed = game.computerRed;

computerBlack = game.computerBlack;

playerRed = game.computerRed;

playerBlack = game.computerBlack;

break;

default:

playerRed=game.playerRed;

playerBlack = game.playerBlack;

break;

}

}

//Manege map Size and Heigh

private void constrictorMap(String map){

if(map.equals("small")){

QueahBoard.sizeOfboard = 5;

QueahBoard.heightOfboard = 2;

}

if(map.equals("mid")){

QueahBoard.sizeOfboard = 7;

QueahBoard.heightOfboard = 3;

}

if(map.equals("large")){

QueahBoard.sizeOfboard = 9;

QueahBoard.heightOfboard = 4;

}

}

//Build the map

public void initBoard()

{

gBoard = new GameButton[sizeOfboard][sizeOfboard];

lBoard = new int[sizeOfboard][sizeOfboard];

setLayout(new GridLayout(sizeOfboard,sizeOfboard));

//up

for (int row = 0; row < heightOfboard; row ++) {

for (int column = 0; column < sizeOfboard; column ++) {

if(Math.abs(column - heightOfboard) <= row){

lBoard[row][column]=0;

gBoard[row][column]= new GameButton(img,null);

gBoard[row][column].addActionListener(new AL(row,column));

}

else{

lBoard[row][column]=-1;

gBoard[row][column] = new GameButton();

gBoard[row][column].setEnabled(false);

gBoard[row][column].setBackground(new Color(0,0,0,0.7f));

}

}

for (int column = 0; column < sizeOfboard; column ++) {

add(gBoard[row][column]);

}

}

//mid

for( int row=heightOfboard; row<(sizeOfboard-heightOfboard); row++){

for(int column=0; column<sizeOfboard; column++){

lBoard[row][column]=0;

gBoard[row][column]= new GameButton(img,null);

gBoard[row][column].addActionListener(new AL(row,column));

add(gBoard[row][column]);

}

}

//down

for (int row = sizeOfboard-heightOfboard,k=heightOfboard-1; row < sizeOfboard; row ++,k--) {

for (int column = 0; column < sizeOfboard; column ++) {

if((lBoard[k][column]==0)){

lBoard[row][column]=0;

gBoard[row][column]= new GameButton(img,null);

gBoard[row][column].addActionListener(new AL(row,column));

}

else{

lBoard[row][column]=-1;

gBoard[row][column] = new GameButton();

gBoard[row][column].setEnabled(false);

gBoard[row][column].setBackground(new Color(0,0,0,0.7f));

}

}

for (int column = 0; column < sizeOfboard; column ++) {

add(gBoard[row][column]);

}

}

addWeight();

turn=startPlayer;

}

public static void setTurn(int start){

turn=start;

}

// function's to set app the map

public void smallMapSolid(){

max\_Player\_soldiers\_on\_board=4;

gBoard[0][2].setSoldier(blackSoldier);

lBoard[0][2] = 2;

gBoard[1][2].setSoldier(blackSoldier);

lBoard[1][2] = 2;

gBoard[1][3].setSoldier(blackSoldier);

lBoard[1][3] = 2;

gBoard[2][3].setSoldier(blackSoldier);

lBoard[2][3] = 2;

gBoard[4][2].setSoldier(redSoldier);

lBoard[4][2] = 1;

gBoard[3][2].setSoldier(redSoldier);

lBoard[3][2] = 1;

gBoard[3][1].setSoldier(redSoldier);

lBoard[3][1] = 1;

gBoard[2][1].setSoldier(redSoldier);

lBoard[2][1] = 1;

}

public void midMapSolid(){

max\_Player\_soldiers\_on\_board=6;

gBoard[0][3].setSoldier(blackSoldier);

lBoard[0][3] = 2;

gBoard[1][3].setSoldier(blackSoldier);

lBoard[1][3] = 2;

gBoard[1][4].setSoldier(blackSoldier);

lBoard[1][4] = 2;

gBoard[2][4].setSoldier(blackSoldier);

lBoard[2][4] = 2;

gBoard[2][5].setSoldier(blackSoldier);

lBoard[2][5] = 2;

gBoard[3][5].setSoldier(blackSoldier);

lBoard[3][5] = 2;

gBoard[6][3].setSoldier(redSoldier);

lBoard[6][3] = 1;

gBoard[5][3].setSoldier(redSoldier);

lBoard[5][3] = 1;

gBoard[5][2].setSoldier(redSoldier);

lBoard[5][2] = 1;

gBoard[4][2].setSoldier(redSoldier);

lBoard[4][2] = 1;

gBoard[4][1].setSoldier(redSoldier);

lBoard[4][1] = 1;

gBoard[3][1].setSoldier(redSoldier);

lBoard[3][1] = 1;

}

public void largeMapSolid(){

max\_Player\_soldiers\_on\_board=8;

gBoard[0][4].setSoldier(blackSoldier);

lBoard[0][4] = 2;

gBoard[1][4].setSoldier(blackSoldier);

lBoard[1][4] = 2;

gBoard[1][5].setSoldier(blackSoldier);

lBoard[1][5] = 2;

gBoard[2][5].setSoldier(blackSoldier);

lBoard[2][5] = 2;

gBoard[2][6].setSoldier(blackSoldier);

lBoard[2][6] = 2;

gBoard[3][6].setSoldier(blackSoldier);

lBoard[3][6] = 2;

gBoard[3][7].setSoldier(blackSoldier);

lBoard[3][7] = 2;

gBoard[4][7].setSoldier(blackSoldier);

lBoard[4][7] = 2;

gBoard[8][4].setSoldier(redSoldier);

lBoard[8][4] = 1;

gBoard[7][4].setSoldier(redSoldier);

lBoard[7][4] = 1;

gBoard[7][3].setSoldier(redSoldier);

lBoard[7][3] = 1;

gBoard[6][3].setSoldier(redSoldier);

lBoard[6][3] = 1;

gBoard[6][2].setSoldier(redSoldier);

lBoard[6][2] = 1;

gBoard[5][2].setSoldier(redSoldier);

lBoard[5][2] = 1;

gBoard[5][1].setSoldier(redSoldier);

lBoard[5][1] = 1;

gBoard[4][1].setSoldier(redSoldier);

lBoard[4][1] = 1;

}

private String getFileInfoMap(){

String path;

switch (game.map) {

case "small":

path = "././files/smallMap.txt";

break;

case "mid":

path = "././files/midMap.txt";

break;

case "large":

path = "././files/largeMap.txt";

break;

default:

path = "././files/smallMap.txt";

break;

}

try {

String content = Files.readString(Paths.get(path));

return content;

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

private int[][] readWeightFile(){

int[][] weight=new int[sizeOfboard][sizeOfboard];

String content = getFileInfoMap();

String[] num = content.split(" ");

int count = 0;

for(int r=0;r<sizeOfboard;r++){

for(int c=0;c<sizeOfboard;c++){

weight[r][c]= (int) Integer.parseInt(num[count++]);

// System.out.print(weight[r][c]+" ");

}

// System.out.println();

}

return weight;

}

private void addWeight(){

int[][] weight=readWeightFile();

for(int i=0; i<sizeOfboard; i++){

for(int j=0; j<sizeOfboard; j++){

gBoard[i][j].setWeight(weight[i][j]);

}

}

}

class AL implements ActionListener{

private int row,column;

private static int previsRow,previsColumn;

private static boolean isSoldiersEaten = false;

private static boolean previsButtonPressed = false;

public AL(int row , int column){

this.row=row;

this.column=column;

}

//if player is victory display it then dispose of old game and create new game

public void victory(int player){

String playerColor;

if(player==1) playerColor ="red";

else playerColor = "black";

javax.swing.JOptionPane.showMessageDialog(game,playerColor);

game.dispose();

new Game();

}

//move soldier in lBoard and in gBoard and switch turn

public void moveSoldier(){

lBoard[row][column]=lBoard[previsRow][previsColumn];

lBoard[previsRow][previsColumn] = 0;

gBoard[row][column].setSoldier(gBoard[previsRow][previsColumn].getSoldier());

gBoard[previsRow][previsColumn].setSoldier(null);

repaint();

if(turn==1) turn=2;

else turn=1;

}

// remove soldier in lBoard and in gBoard and switch turn and remove from Stack

public void removeSoldier(int row, int column){

boolean isRemoveSoldierFromStack;

if(turn==1){

isRemoveSoldierFromStack = playerBlack.removeSoldierFromStack();

playerBlack.removeSoldierFromBoard();

if(!isRemoveSoldierFromStack)

{

lBoard[row][column] = 0;

gBoard[row][column].setSoldier(null);

}

else

{

lBoard[row][column] = 0;

gBoard[row][column].setSoldier(null);

isSoldiersEaten = true;

}

}

else

{

isRemoveSoldierFromStack = playerRed.removeSoldierFromStack();

playerRed.removeSoldierFromBoard();

if(!isRemoveSoldierFromStack)

{

lBoard[row][column] = 0;

gBoard[row][column].setSoldier(null);

}

else

{

lBoard[row][column] = 0;

gBoard[row][column].setSoldier(null);

isSoldiersEaten = true;

}

}

}

// add soldier to board and switch turn

public void addSoldierToBoard(){

if(turn==1){

playerRed.addSoldierToBoard(max\_Player\_soldiers\_on\_board);

if(lBoard[row][column]==0){

lBoard[row][column]=1;

gBoard[row][column].setSoldier(redSoldier);

isSoldiersEaten=false;

turn=2;

}

}

else

{

playerBlack.addSoldierToBoard(max\_Player\_soldiers\_on\_board);

if(lBoard[row][column]==0){

lBoard[row][column]=2;

gBoard[row][column].setSoldier(blackSoldier);

isSoldiersEaten=false;

turn=1;

}

}

}

// if gamemode is 0/default then the gamemode is HumanVSHuman

//if gamemode is 1 then the gamemode is HumanVSComputer

//if gamemode is 2 then the gamemode is ComputerVSComputer

public void actionPerformed(ActionEvent e){

switch (game.gameMode) {

case 1:

if(game.gameStart==1){

ComputerMove(false);

game.gameStart=0;

}

HumanMove();

break;

case 2:

if(isFirstCOMvsCOM){

ComputerMove(false);

isFirstCOMvsCOM=false;

}

break;

default:

HumanMove();

break;

}

}

private void HumanMove(){

//check if the player is computer and this is His turn if it is then return void

if(turn == playerRed.getPlayer\_color() && (!playerRed.IsHuman())) return;

if(turn == playerBlack.getPlayer\_color() && (!playerBlack.IsHuman())) return;

//check if the player Won and if it is call the function victory

if(playerRed.getSoldier\_on\_board() == 0) victory(2);

if(playerBlack.getSoldier\_on\_board() == 0) victory(1);

/\*if soldier was Eaten then call the function addSoldierToBoard.

else check If the button is pressed a previous time if pressed the check if

the move is valid and if the move was to eat the other player soldier and set the previsButtonPressed to false.

if The button was not pressed once the previous time then check if the button is populated by soldier if it populated then

set previsButtonPressed to true and previsRow=row and previsColumn=column if it not populated set previsButtonPressed to false\*/

if(isSoldiersEaten){

addSoldierToBoard();

new Thread(new Runnable(){

public void run(){

try{

Thread.sleep(THINKING\_TIME);

ComputerMove(false);

}

catch(InterruptedException ex) {}

}

}).start();

}

else if(previsButtonPressed){

if(((previsRow == row+1 || previsRow == row-1) && previsColumn == column) || ((previsColumn == column+1 || previsColumn == column-1) && previsRow == row )){

if(lBoard[row][column] == 0 && lBoard[previsRow][previsColumn] == turn){

if(!(previsRow == row && previsColumn == column)){

System.out.println("1");

moveSoldier();

if(game.gameMode==1){

new Thread(new Runnable(){

public void run(){

try{

Thread.sleep(THINKING\_TIME);

ComputerMove(false);

}

catch(InterruptedException ex) {}

}

}).start();

}

}

}

}

if(lBoard[row][column] == 0 && lBoard[previsRow][previsColumn] == turn){

if(!(previsRow == row && previsColumn == column)){

if(previsColumn == column){

if((previsRow == row+2 && (lBoard[row+1][column] !=turn && lBoard[row+1][column] !=0 ))){

System.out.println("2");

removeSoldier(row+1, column);

moveSoldier();

if(game.gameMode==1) ComputerMove(true);

}

else if((previsRow == row-2 && (lBoard[row-1][column] !=turn && lBoard[row-1][column] !=0 ))){

System.out.println("3");

removeSoldier(row-1,column);

moveSoldier();

if(game.gameMode==1){

ComputerMove(true);

}

}

}

else if(previsRow == row){

if((previsColumn == column+2 && (lBoard[row][column+1] !=turn && lBoard[row][column+1] !=0 ))){

System.out.println("4");

removeSoldier(row, column+1);

moveSoldier();

if(game.gameMode==1) ComputerMove(true);

}

else if((previsColumn == column-2 && (lBoard[row][column-1] !=turn && lBoard[row][column-1] !=0 ))){

System.out.println("5");

removeSoldier(row, column-1);

moveSoldier();

if(game.gameMode==1) ComputerMove(true);

}

}

}

}

previsButtonPressed = false;

}

else if(lBoard[row][column] == 0 ){

System.out.println("7");

previsButtonPressed = false;

}

else

{

System.out.println("8");

previsButtonPressed = true;

previsRow=row;

previsColumn=column;

}

//debag:

//System.out.println("previsRow:"+previsRow+" "+"previsColumn:"+previsColumn+" "+"previsButtonPressed:"+previsButtonPressed+" "+"row:"+row+" "+"column:"+column+" "+"player:"+lBoard[row][column]+" "+"turn:"+turn);

}

private void ComputerMove(boolean isEaten){

// int isEaten 0|1 to check if the player ate the Computer

int tempRow, tempColumn,tempPrevisRow,tempPrevisColumn;

//check if the player Won and if it is call the function victory

if(playerRed.getSoldier\_on\_board() == 0) victory(2);

if(playerBlack.getSoldier\_on\_board() == 0) victory(1);

int data[];

//check if the player is computer and this is His turn if it is then call the function play

if(turn == playerRed.getPlayer\_color() && !playerRed.IsHuman()) data = computerRed.play(isEaten,lBoard,gBoard,computerBlack);

else if(turn == playerBlack.getPlayer\_color() && !playerBlack.IsHuman()) data = computerBlack.play(isEaten,lBoard,gBoard,computerRed);

else{

//System.out.println("dont move computer");

return;

}

//temperary store row and column in tempRow and tempColumn so we can use row and column it in the moveSoldier function

//and not destroyed the value of the button

tempRow =row;

tempColumn = column;

tempPrevisRow = previsRow;

tempPrevisColumn = previsColumn;

row = data[0];

column = data[1];

previsRow = data[2];

previsColumn = data[3];

if(data[6]==0 && (!isEaten||data[7]==0)){

System.out.println("move soldier");

moveSoldier();

if(game.gameMode==2){

new Thread(new Runnable(){

public void run(){

try{

Thread.sleep(THINKING\_TIME);

ComputerMove(false);

}

catch(InterruptedException ex) {}

}

}).start();

}

}

else if(isEaten && data[7]==1){

System.out.println("addSoldierToBoard");

addSoldierToBoard();

if(game.gameMode==2){

new Thread(new Runnable(){

public void run(){

try{

Thread.sleep(THINKING\_TIME);

ComputerMove(false);

}

catch(InterruptedException ex) {}

}

}).start();

}

}

else

{

System.out.println("eat soldier");

removeSoldier(data[4], data[5]);

moveSoldier();

if(game.gameMode==2){

ComputerMove(true);

}

}

row=tempRow;

column=tempColumn;

previsRow=tempPrevisRow;

previsColumn=tempPrevisColumn;

//check if the player Won and if it is call the function victory

if(playerRed.getSoldier\_on\_board() == 0) victory(2);

if(playerBlack.getSoldier\_on\_board() == 0) victory(1);

}

}

}

package code**;**

**import** java**.**awt**.\*;**

**import** javax**.**swing**.\*;**

**import** javax**.**swing**.**JPanel**;**

public class Players **extends** JPanel **{**

protected int soldier\_left**;**

protected int soldier\_on\_board**;**

protected final int player\_color**;** // 1 red | 2 black

protected String map**;**

private static final ImageIcon RedSoldier **=** **new** ImageIcon**(**"././img/Soldier\_red\_New.png"**);**

private static final ImageIcon BlackSoldier **=** **new** ImageIcon**(**"././img/Soldier\_black\_New.png"**);**

private JLabel img**;**

public Players**(**int player\_color**,**String map**){**

**this.**player\_color **=** player\_color**;**

**this.**map **=** map**;**

int soldiers**;**

**if(**map**.**equals**(**"small"**)){**

soldier\_on\_board**=**4**;**

soldiers**=**10**;**

**}**

**else** **if(**map**.**equals**(**"mid"**)){**

soldier\_on\_board**=**6**;**

soldiers**=**14**;**

**}**

**else** **if(**map**.**equals**(**"large"**))**

**{**

soldier\_on\_board**=**8**;**

soldiers**=**18**;**

**}**

**else**

**{**

soldier\_on\_board**=**4**;**

soldiers**=**10**;**

**}**

**this.**soldier\_left**=**soldiers**-**soldier\_on\_board**;**

setLayout**(new** GridLayout**(**1**,**soldier\_left**));**

drawSoldier\_left**();**

**}**

public int getPlayer\_color**()** **{**

**return** player\_color**;**

**}**

public int getSoldier\_on\_board**()** **{**

**return** soldier\_on\_board**;**

**}**

public int getSoldierLeft**()** **{**

**return** soldier\_left**;**

**}**

public String getMap**()** **{**

**return** map**;**

**}**

public void copy**(**Players player**){**

**this.**soldier\_left**=**player**.**soldier\_left**;**

**this.**soldier\_on\_board**=**player**.**soldier\_on\_board**;**

**}**

// 0 no soldier left | 1 soldier on board is 4 and thir is soldier\_left

public boolean removeSoldierFromStack**()** **{**

**if(**soldier\_left **<=** 0**)** **return** **false;**

soldier\_left **-=** 1**;**

remove**(**soldier\_left**);**

revalidate**();**

**return** **true;**

**}**

public boolean removeSoldierFromBoard**()**

**{**

**if(**soldier\_on\_board **<=** 0**)** **return** **false;**

soldier\_on\_board **-=** 1**;**

**return** **true;**

**}**

public boolean addSoldierToBoard**(**int max\_soldier\_on\_board**){**

**if(**max\_soldier\_on\_board**<**soldier\_on\_board**+**1**)** **return** **false;**

soldier\_on\_board **+=** 1**;**

**return** **true;**

**}**

public boolean isSoldiersLeft**()** **{**

**if** **(**soldier\_left **==** 0 **&&** soldier\_on\_board **==**0**)** **return** **true;**

**return** **false;**

**}**

public void drawSoldier\_left**(){**

**for(**int i **=** 0**;** i **<**soldier\_left**;**i**++){**

**if(**player\_color**==**1**)** img **=** **new** JLabel**(**RedSoldier**);**

**else** img **=** **new** JLabel**(**BlackSoldier**);**

add**(**img**,**i**);**

**}**

**}**

public boolean IsHuman**()** **{return** **true;}**

// public int getWeight(int [][]lBoard){

// }

**}**

package code**;**

**import** java**.**util**.**List**;**

**import** java**.**util**.**Stack**;**

public class Computer **extends** Players **{**

private int difficulty**;**

private int player\_color**;** //1 red 2 black

private int isSoldierNotLeftFirstTime**;**

private int sameMoveCount**;**

private int depth**=**3**;**

private Coordinate lestPos**=null;**

private int **[][]**lBoard**;**

private GameButton **[][]**gBoard**;**

private Stack**<**SoldierMoves**>** soldierMovesStack**;**

private Computer enamy**;**

public Computer**(**int player\_color**,** String map**,**int difficulty**)** **{**

**super(**player\_color**,** map**);**

**this.**player\_color **=** player\_color**;**

**this.**difficulty**=**difficulty**;**

isSoldierNotLeftFirstTime **=**0**;**

soldierMovesStack **=** **new** Stack**<**SoldierMoves**>();**

**}**

//this function is to manage the computer,isSoldierLeft-1(yes)/0(no)

public int**[]** play**(**boolean isEaten**,**int **[][]**lBoard**,**GameButton **[][]**gBoard**,** Computer enamy**){**

int test**[]=new** int**[**8**];**

**this.**lBoard**=**lBoard**;**

**this.**gBoard**=**gBoard**;**

**this.**enamy**=**enamy**;**

//System.out.println(enamy.getLboard());

**if(**getSoldierLeft**()==**0**)** isSoldierNotLeftFirstTime**++;**

**if(**isEaten **&&** isSoldierNotLeftFirstTime**<=**1**)** test**=**addNewSolid**();**

**else** test**=**move**();**

soldierMovesStack**.**clear**();**

**return** test**;**

**}**

//NOT FOR QUEAHBOARD ONLY MINMAX CAN USE THIS FUNCTION!!!!!!!!!!

public int**[]** playNegamax**(**boolean isEaten**,**Coordinate moveCoordinate**,**Coordinate eat**,**Coordinate soldierCoordinates **,**int **[][]**lBoard**,**GameButton **[][]**gBoard**){**

int test**[]=new** int**[**8**];**

**this.**lBoard**=**lBoard**;**

**this.**gBoard**=**gBoard**;**

**if(**getSoldierLeft**()==**0**)** isSoldierNotLeftFirstTime**++;**

**if(**isEaten **&&** isSoldierNotLeftFirstTime**<=**1**)** test**=**addNewSolid**();**

**else{**

**if(**isSoldierNotLeftFirstTime**<=**1**)** test**[**7**]=**1**;**

else test[7]=0;

if(eat !=null){

test[0] = moveCoordinate.getRow();

test[1] = moveCoordinate.getColumn();

test[2] = soldierCoordinates.getRow();

test[3] = soldierCoordinates.getColumn();

test[4] = eat.getRow();

test[5] = eat.getColumn();

test[6] = 1;

}

else{

test[0] = moveCoordinate.getRow();

test[1] = moveCoordinate.getColumn();

test[2] = soldierCoordinates.getRow();

test[3] = soldierCoordinates.getColumn();

test[4] = 0;

test[5] = 0;

test[6] = 0;

}

}

return test;

}

//return newRow,newColumn,0,0,0,0,0,0

//this function is for the computer to add new soldier if soldier is eaten

private int[] addNewSolid(){

int test[]=new int[8];

int data[]=findMostWeightBlock();

test[0]=data[0];

test[1]=data[1];

if(isSoldierNotLeftFirstTime<=1) test[7]=1;

else test[7]=0;

for(int i=2;i<7;i++) test[i]= 0;

return test;

}

//return newRow,newColumn,previsRow,previsColumn, eatRow,eatColum, isEat- 1(yes)/0(no), isSoldierLeft-1(yes)/0(no)

//this function is for the computer to move the soldier

private int[] move(){

int test[]=new int[8];

int index;

int size;

Coordinate soldierCoordinate;

Stack<SoldierMoves> copySoldierMovesStack=new Stack<SoldierMoves>();

Stack<SoldierMoves> eatSoldierMovesStack=new Stack<SoldierMoves>();

Stack<SoldierMoves> notSafeSoldierMovesStack=new Stack<SoldierMoves>();

Negamax negamax;

findAllPossibleSoldier();

copyStack(copySoldierMovesStack,soldierMovesStack);

while(!copySoldierMovesStack.isEmpty()){

if(!copySoldierMovesStack.peek().getPossibleEatMoves().isEmpty()) eatSoldierMovesStack.push(copySoldierMovesStack.peek());

copySoldierMovesStack.pop();

}

copySoldierMovesStack.clear();

copyStack(copySoldierMovesStack,soldierMovesStack);

while(!copySoldierMovesStack.isEmpty()){

if(!copySoldierMovesStack.peek().isSoldierNotInDanger()) notSafeSoldierMovesStack.push(copySoldierMovesStack.peek());

copySoldierMovesStack.pop();

}

if(!eatSoldierMovesStack.isEmpty()){

System.out.println("eat");

List<Coordinate[]> possibleEatMoves;

if(difficulty==2){

negamax = new Negamax(lBoard,gBoard,depth,Computer.this,enamy,false);

negamax.negamax();

}

else negamax=null;

SoldierMoves bestEatMoves;

switch(difficulty){

case 0:

popRandom(eatSoldierMovesStack);

possibleEatMoves=eatSoldierMovesStack.peek().getPossibleEatMoves();

soldierCoordinate=eatSoldierMovesStack.peek().getSoldierCoordinate();

break;

case 1:

bestEatMoves = findBestEat(eatSoldierMovesStack);

possibleEatMoves=bestEatMoves.getPossibleEatMoves();

soldierCoordinate=bestEatMoves.getSoldierCoordinate();

break;

case 2:

for(int i=0 ;i<negamax.bestpop;i++){

eatSoldierMovesStack.pop();

}

bestEatMoves=eatSoldierMovesStack.peek();

possibleEatMoves=bestEatMoves.getPossibleEatMoves();

soldierCoordinate=bestEatMoves.getSoldierCoordinate();

break;

default:

popRandom(eatSoldierMovesStack);

possibleEatMoves=eatSoldierMovesStack.peek().getPossibleEatMoves();

soldierCoordinate=eatSoldierMovesStack.peek().getSoldierCoordinate();

break;

}

size = possibleEatMoves.size();

switch(difficulty){

case 0:

index=(int)(Math.random()\*(size-1));

break;

case 1:

index=indexOfBestEat(possibleEatMoves);

break;

case 2:

index=negamax.bestMoveIndex;

break;

default:

index=(int)(Math.random()\*(size-1));

break;

}

test[0] = possibleEatMoves.get(index)[0].getRow();

test[1] = possibleEatMoves.get(index)[0].getColumn();

test[2] = soldierCoordinate.getRow();

test[3] = soldierCoordinate.getColumn();

test[4] = possibleEatMoves.get(index)[1].getRow();

test[5] = possibleEatMoves.get(index)[1].getColumn();

test[6] = 1;

}

else if(!notSafeSoldierMovesStack.isEmpty()){

System.out.println("move denger Soldier");

List<Coordinate> possibleMoves;

if(difficulty==2){

negamax = new Negamax(lBoard,gBoard,depth,Computer.this,enamy,false);

negamax.negamax();

}

else negamax=null;

SoldierMoves bestMoves;

switch(difficulty){

case 0:

popRandom(notSafeSoldierMovesStack);

possibleMoves=notSafeSoldierMovesStack.peek().getPossibleMoves();

soldierCoordinate=notSafeSoldierMovesStack.peek().getSoldierCoordinate();

break;

case 1:

bestMoves = findBestMove(notSafeSoldierMovesStack);

possibleMoves=bestMoves.getPossibleMoves();

soldierCoordinate=bestMoves.getSoldierCoordinate();

break;

case 2:

for(int i=0 ;i<negamax.bestpop;i++){

soldierMovesStack.pop();

}

bestMoves=soldierMovesStack.peek();

possibleMoves=bestMoves.getPossibleMoves();

soldierCoordinate=bestMoves.getSoldierCoordinate();

break;

default:

popRandom(notSafeSoldierMovesStack);

possibleMoves=notSafeSoldierMovesStack.peek().getPossibleMoves();

soldierCoordinate=notSafeSoldierMovesStack.peek().getSoldierCoordinate();

break;

}

size = notSafeSoldierMovesStack.peek().getPossibleMoves().size();

switch(difficulty){

case 0:

index=(int)(Math.random()\*(size-1));

break;

case 1:

index=indexOfBestMove(possibleMoves);

break;

case 2:

index=negamax.bestMoveIndex;

break;

default:

index=(int)(Math.random()\*(size-1));

break;

}

//fixs loop infanetly problem

if(lestPos==null)lestPos=possibleMoves.get(index);

else if(lestPos.equals(possibleMoves.get(index))){

sameMoveCount++;

if(sameMoveCount>5){

index = 0;

while(lestPos.equals(possibleMoves.get(index)))

{

index++;

if(index>=size) break;

}

}

}

else if(possibleMoves.size()>1){

sameMoveCount=0;

lestPos=possibleMoves.get(index);

}

test[0] = possibleMoves.get(index).getRow();

test[1] = possibleMoves.get(index).getColumn();

test[2] = soldierCoordinate.getRow();

test[3] = soldierCoordinate.getColumn();

test[4] = 0;

test[5] = 0;

test[6] = 0;

if(isSoldierNotLeftFirstTime<=1) test[7]=1;

else test[7]=0;

return test;

}

else{

System.out.println("move");

List<Coordinate> possibleMoves;

if(difficulty==2){

negamax = new Negamax(lBoard,gBoard,depth,Computer.this,enamy,false);

negamax.negamax();

}

else negamax=null;

SoldierMoves bestMoves;

switch(difficulty){

case 0:

popRandom(soldierMovesStack);

possibleMoves=soldierMovesStack.peek().getPossibleMoves();

soldierCoordinate=soldierMovesStack.peek().getSoldierCoordinate();

break;

case 1:

bestMoves = findBestMove(soldierMovesStack);

possibleMoves=bestMoves.getPossibleMoves();

soldierCoordinate=bestMoves.getSoldierCoordinate();

break;

case 2:

for(int i=0 ;i<negamax.bestpop;i++){

soldierMovesStack.pop();

}

bestMoves=soldierMovesStack.peek();

possibleMoves=bestMoves.getPossibleMoves();

soldierCoordinate=bestMoves.getSoldierCoordinate();

break;

default:

popRandom(soldierMovesStack);

possibleMoves=soldierMovesStack.peek().getPossibleMoves();

soldierCoordinate=soldierMovesStack.peek().getSoldierCoordinate();

break;

}

size = soldierMovesStack.peek().getPossibleMoves().size();

switch(difficulty){

case 0:

index=(int)(Math.random()\*(size-1));

break;

case 1:

index=indexOfBestMove(possibleMoves);

break;

case 2:

index=negamax.bestMoveIndex;

break;

default:

index=(int)(Math.random()\*(size-1));

break;

}

//fixs loop infanetly problem

if(lestPos==null)lestPos=possibleMoves.get(index);

else if(lestPos.equals(possibleMoves.get(index))){

sameMoveCount++;

if(sameMoveCount>5){

index = 0;

while(lestPos.equals(possibleMoves.get(index)))

{

index++;

if(index>=size) break;

}

}

}

else if(possibleMoves.size()>1){

sameMoveCount=0;

lestPos=possibleMoves.get(index);

}

test[0] = possibleMoves.get(index).getRow();

test[1] = possibleMoves.get(index).getColumn();

test[2] = soldierCoordinate.getRow();

test[3] = soldierCoordinate.getColumn();

test[4] = 0;

test[5] = 0;

test[6] = 0;

}

if(isSoldierNotLeftFirstTime<=1) test[7]=1;

else test[7]=0;

return test;

}

//this function pop the best move from the stack

private SoldierMoves findBestMove(Stack<SoldierMoves> soldierMovesStack){

Stack<SoldierMoves> copySoldierMovesStack=new Stack<SoldierMoves>();

SoldierMoves bestMove=null;

copyStack(copySoldierMovesStack, soldierMovesStack);

while (!copySoldierMovesStack.isEmpty()){

if(bestMove==null) bestMove = copySoldierMovesStack.pop();

else if(bestMove.weightSoldierMoves()<copySoldierMovesStack.peek().weightSoldierMoves()) bestMove = copySoldierMovesStack.pop();

else copySoldierMovesStack.pop();

}

System.out.println("\nfindBestMove - bestMove: "+bestMove+"\nWeight: "+bestMove.weightSoldierMoves()+"\n");

return bestMove;

}

//this function return the index of the best move in the list

private int indexOfBestMove(List<Coordinate> possibleMoves){

boolean isSafe=false;

int index=0;

int numOfindex=0;

int bestWeight = Integer.MIN\_VALUE;

SoldierMoves bestMove=null;

SoldierMoves Move=null;

for (Coordinate coordinate : possibleMoves){

int weight;

Move = new SoldierMoves(lBoard,gBoard,coordinate);

weight=Move.weightSoldierMoves();

if(!isSafe && Move.isSoldierNotInDanger()){

System.out.println("indexOfBestMove - 1| "+" weight: "+weight+" isSafe: "+Move.isSoldierNotInDanger()+" CoordinateMove: "+Move.getSoldierCoordinate());

isSafe=true;

bestWeight=weight;

bestMove=Move;

index=numOfindex;

}

else if(weight>bestWeight && Move.isSoldierNotInDanger()){

System.out.println("indexOfBestMove - 2| "+" weight: "+weight+" isSafe: "+Move.isSoldierNotInDanger()+" CoordinateMove: "+Move.getSoldierCoordinate());

isSafe=Move.isSoldierNotInDanger();

bestWeight=weight;

bestMove=Move;

index=numOfindex;

}

else if(weight>bestWeight && !isSafe){

System.out.println("indexOfBestMove - 3| "+" weight: "+weight+" isSafe: "+Move.isSoldierNotInDanger()+" CoordinateMove: "+Move.getSoldierCoordinate());

isSafe=Move.isSoldierNotInDanger();

bestWeight=weight;

bestMove=Move;

index=numOfindex;

}

else System.out.println("indexOfBestMove - 4| "+" weight: "+weight+" isSafe: "+Move.isSoldierNotInDanger()+" CoordinateMove: "+Move.getSoldierCoordinate());

numOfindex++;

}

System.out.println("\nindexOfBestMove - bestMove: "+bestMove+"\nWeight: "+bestMove.weightSoldierMoves()+"\nisSafe: "+isSafe+"\nindex: "+index+"\n");

System.out.println();

return index;

}

//this function pop the best eat move from the stack

private SoldierMoves findBestEat(Stack<SoldierMoves> eatSoldierMovesStack){

Stack<SoldierMoves> copyEatSoldierMovesStack=new Stack<SoldierMoves>();

SoldierMoves bestEatMoves=null;

copyStack(copyEatSoldierMovesStack, eatSoldierMovesStack);

while (!copyEatSoldierMovesStack.isEmpty()){

if(bestEatMoves==null) bestEatMoves = copyEatSoldierMovesStack.pop();

else if(bestEatMoves.weightSoldierMoves()<copyEatSoldierMovesStack.peek().weightSoldierMoves()) bestEatMoves = copyEatSoldierMovesStack.pop();

else copyEatSoldierMovesStack.pop();

}

System.out.println("\nfindBestEat - bestEatMoves: "+bestEatMoves+"\nWeight: "+bestEatMoves.weightSoldierMoves()+"\n");

return bestEatMoves;

}

//this function retun the best index of the possible Eat moves in the list

private int indexOfBestEat(List<Coordinate[]> possibleEatMoves){

int index=0;

int numOfindex=0;

int bestWeight = Integer.MIN\_VALUE;

SoldierMoves bestEatMoves=null;

SoldierMoves EatMoves=null;

for (Coordinate[] coordinate : possibleEatMoves){

int weight;

EatMoves = new SoldierMoves(lBoard,gBoard,coordinate[0]);

weight=EatMoves.weightSoldierMoves();

if(weight>bestWeight){

bestWeight=weight;

index=numOfindex;

bestEatMoves=EatMoves;

}

numOfindex++;

}

System.out.println("\nindexOfBestEat - bestEatMoves: "+bestEatMoves+"\nWeight: "+bestEatMoves.weightSoldierMoves()+"\n");

return index;

}

//this function is pop random soldierMoves from stack

private void popRandom(Stack<SoldierMoves> stack){

int numOfpop=(int)(Math.random()\*(stack.size()));

while(numOfpop>0){

stack.pop();

numOfpop--;

}

}

//this function is copy stack from one stack to another

private void copyStack(Stack<SoldierMoves> copySoldierMovesStack,Stack<SoldierMoves> soldierMovesStack){

Stack<SoldierMoves> copySoldierMovesStack2=new Stack<SoldierMoves>();

while(!soldierMovesStack.isEmpty()){

copySoldierMovesStack.push(soldierMovesStack.peek());

copySoldierMovesStack2.push(soldierMovesStack.pop());

}

while(!copySoldierMovesStack2.isEmpty()) soldierMovesStack.push(copySoldierMovesStack2.pop());

}

//find all the soldier of the computer that is not stuck

public void findAllPossibleSoldier(){

soldierMovesStack.clear();

for(int i=0;i<lBoard.length;i++){

for(int j=0;j<lBoard.length;j++){

if(lBoard[i][j]==player\_color){

soldierMovesStack.push(new SoldierMoves(lBoard,gBoard,new Coordinate(i,j,player\_color)));

if(soldierMovesStack.peek().isSoldierStuck()) soldierMovesStack.pop();

}

}

}

}

//this function is to find the max weight coordinate

private int[] findMostWeightBlock(){

int data[]=new int[3];

int weight=0;

for(int i=0;i<lBoard.length;i++){

for(int j=0;j<lBoard.length;j++){

if(gBoard[i][j].getWeight()>weight && lBoard[i][j]==0){

weight = gBoard[i][j].getWeight();

data[0]=i;

data[1]=j;

data[2]=weight;

}

}

}

return data;

}

//this function is printing test

private void printTest(int[] test){

for (int i : test) {

System.out.print(i+" ");

}

}

public int getDifficulty(){

return difficulty;

}

public Stack<SoldierMoves> getSoldierMovesStack(){

return soldierMovesStack;

}

public void copy(Computer computer){

super.copy(computer);

this.map=computer.map;

//copyBoard(computer.lBoard,lBoard );

//copyGBoard(computer.gBoard,gBoard);

this.difficulty=computer.difficulty;

this.soldierMovesStack=computer.soldierMovesStack;

this.isSoldierNotLeftFirstTime=computer.isSoldierNotLeftFirstTime;

}

public void copyBoard(int[][] board, int[][] newBoard) {

for (int i = 0; i < board.length; i++) {

for (int j = 0; j < board[i].length; j++) {

newBoard[i][j] = board[i][j];

}

}

}

public void copyGBoard(GameButton [][]gBoard, GameButton [][]newGBoard) {

if(newGBoard==null) newGBoard=new GameButton[gBoard.length][gBoard.length];

for (int i = 0; i < gBoard.length; i++) {

for (int j = 0; j < gBoard[i].length; j++) {

newGBoard[i][j] = gBoard[i][j];

}

}

}

public void setBoard(int[][] board){

this.lBoard=board;

}

public int[][] getLboard(){

return lBoard;

}

public void setGBoard(GameButton[][] gBoard){

this.gBoard=gBoard;

}

public GameButton[][] getGBoard(){

return gBoard;

}

public int getIsSoldierNotLeftFirstTime(){

return isSoldierNotLeftFirstTime;

}

@Override

public boolean IsHuman() {return false;}

}

package code**;**

**import** java**.**util**.\*;**

public class SoldierMoves **{**

private boolean isSafe**;**

private int player\_color**;**

private int opponent\_color**;**

private int **[][]**lBoard**;**

private GameButton **[][]**gBoard**;**

private Coordinate up**,** down**,** left**,** right**;**

private Coordinate**[]** twoUp**,** twoDown**,** twoLeft**,** twoRight**;**

private Coordinate soldierCoordinate**;**

private List**<**Coordinate**>** possibleMoves**;**

private List**<**Coordinate**>** allySoldier**;**

private List**<**Coordinate**[]>** possibleEatMoves**;**

private List**<**Coordinate**>** coordinatesOfEnemySoldiercanNotEat**;**

private List**<**Coordinate**>** notSafeMove**;**

//this function is a constructor

public SoldierMoves**(**int **[][]**lBoard**,**GameButton **[][]**gBoard**,**Coordinate soldierCoordinate**){**

**this.**lBoard**=**lBoard**;**

**this.**gBoard**=**gBoard**;**

**this.**soldierCoordinate**=**soldierCoordinate**;**

**this.**player\_color**=**soldierCoordinate**.**getValue**();** // 0 empty 1 red 2 black

**if(**player\_color**==**1**){**

opponent\_color**=**2**;**

**}**

**else{**

opponent\_color**=**1**;**

**}**

possibleMoves **=** **new** ArrayList**<**Coordinate**>();**

possibleEatMoves **=** **new** ArrayList**<**Coordinate**[]>();**

coordinatesOfEnemySoldiercanNotEat **=** **new** ArrayList**<**Coordinate**>();**

allySoldier **=** **new** ArrayList**<**Coordinate**>();**

notSafeMove **=** **new** ArrayList**<**Coordinate**>();**

scannMap**();**

isSoldierNotInDanger**();**

findPossibleMoves**();**

findPossibleEatMoves**();**

findCoordinatesOfEnemySoldiercanNotEat**();**

findAllySoldier**();**

findMoveNotSafe**();**

**}**

//this function scann the map and update the directions and the Two\_directions

public void scannMap**(){**

**if(**soldierCoordinate**.**getRow**()+**1**>=**lBoard**.**length**)** up**=new** Coordinate**(-**1**,-**1**,-**1**);**

**else** up**=new** Coordinate**(**soldierCoordinate**.**getRow**()+**1**,**soldierCoordinate**.**getColumn**(),**lBoard**[**soldierCoordinate**.**getRow**()+**1**][**soldierCoordinate**.**getColumn**()]);**

**if(**soldierCoordinate**.**getRow**()-**1**<**0**)** down**=new** Coordinate**(-**1**,-**1**,-**1**);**

**else** down**=new** Coordinate**(**soldierCoordinate**.**getRow**()-**1**,**soldierCoordinate**.**getColumn**(),**lBoard**[**soldierCoordinate**.**getRow**()-**1**][**soldierCoordinate**.**getColumn**()]);**

**if(**soldierCoordinate**.**getColumn**()+**1**>=**lBoard**.**length**)** right**=new** Coordinate**(-**1**,-**1**,-**1**);**

**else** right**=new** Coordinate**(**soldierCoordinate**.**getRow**(),**soldierCoordinate**.**getColumn**()+**1**,**lBoard**[**soldierCoordinate**.**getRow**()][**soldierCoordinate**.**getColumn**()+**1**]);**

**if(**soldierCoordinate**.**getColumn**()-**1**<**0**)** left**=new** Coordinate**(-**1**,-**1**,-**1**);**

**else** left**=new** Coordinate**(**soldierCoordinate**.**getRow**(),**soldierCoordinate**.**getColumn**()-**1**,**lBoard**[**soldierCoordinate**.**getRow**()][**soldierCoordinate**.**getColumn**()-**1**]);**

**if(**soldierCoordinate**.**getRow**()+**2**>=**lBoard**.**length**){**

twoUp**=new** Coordinate**[**2**];**

twoUp**[**0**]=new** Coordinate**(-**1**,-**1**,-**1**);**

twoUp**[**1**]=new** Coordinate**(-**1**,-**1**,-**1**);**

**}**

**else{**

twoUp**=new** Coordinate**[**2**];**

twoUp**[**0**]=new** Coordinate**(**soldierCoordinate**.**getRow**()+**2**,**soldierCoordinate**.**getColumn**(),**lBoard**[**soldierCoordinate**.**getRow**()+**2**][**soldierCoordinate**.**getColumn**()]);**

twoUp**[**1**]=**up**;**

**}**

**if(**soldierCoordinate**.**getRow**()-**2**<**0**){**

twoDown**=new** Coordinate**[**2**];**

twoDown**[**0**]=new** Coordinate**(-**1**,-**1**,-**1**);**

twoDown**[**1**]=new** Coordinate**(-**1**,-**1**,-**1**);**

**}**

**else{**

twoDown**=new** Coordinate**[**2**];**

twoDown**[**0**]=new** Coordinate**(**soldierCoordinate**.**getRow**()-**2**,**soldierCoordinate**.**getColumn**(),**lBoard**[**soldierCoordinate**.**getRow**()-**2**][**soldierCoordinate**.**getColumn**()]);**

twoDown**[**1**]=**down**;**

}

if(soldierCoordinate.getColumn()+2>=lBoard.length){

twoRight=new Coordinate[2];

twoRight[0]=new Coordinate(-1,-1,-1);

twoRight[1]=new Coordinate(-1,-1,-1);

}

else{

twoRight=new Coordinate[2];

twoRight[0]=new Coordinate(soldierCoordinate.getRow(),soldierCoordinate.getColumn()+2,lBoard[soldierCoordinate.getRow()][soldierCoordinate.getColumn()+2]);

twoRight[1]=right;

}

if(soldierCoordinate.getColumn()-2<0){

twoLeft=new Coordinate[2];

twoLeft[0]=new Coordinate(-1,-1,-1);

twoLeft[1]=new Coordinate(-1,-1,-1);

}

else{

twoLeft=new Coordinate[2];

twoLeft[0]=new Coordinate(soldierCoordinate.getRow(),soldierCoordinate.getColumn()-2,lBoard[soldierCoordinate.getRow()][soldierCoordinate.getColumn()-2]);

twoLeft[1]=left;

}

}

//updates the list of possible moves

public void findPossibleMoves(){

if(up.getValue()==0){

possibleMoves.add(up);

}

if(down.getValue()==0){

possibleMoves.add(down);

}

if(right.getValue()==0){

possibleMoves.add(right);

}

if(left.getValue()==0){

possibleMoves.add(left);

}

}

//updates the list of possible eat moves

public void findPossibleEatMoves(){

if(twoUp[0].getValue()==0 && twoUp[1].getValue() != player\_color && twoUp[1].getValue() != 0){

possibleEatMoves.add(twoUp);

}

if(twoDown[0].getValue()==0 && twoDown[1].getValue() != player\_color && twoDown[1].getValue() != 0){

possibleEatMoves.add(twoDown);

}

if(twoRight[0].getValue()==0 && twoRight[1].getValue() != player\_color && twoRight[1].getValue() != 0){

possibleEatMoves.add(twoRight);

}

if(twoLeft[0].getValue()==0 && twoLeft[1].getValue() != player\_color && twoLeft[1].getValue() != 0){

possibleEatMoves.add(twoLeft);

}

}

//updates the list of coordinates of enemy soldier that can not be eaten

public void findCoordinatesOfEnemySoldiercanNotEat(){

if(up.getValue()!=0 && up.getValue() !=-1 && up.getValue()!=player\_color && twoUp[0].getValue()!=0 && twoUp[0].getValue()!=-1){

coordinatesOfEnemySoldiercanNotEat.add(up);

}

if(down.getValue()!=0 && down.getValue() !=-1 && down.getValue()!=player\_color && twoDown[0].getValue()!=0 && twoDown[0].getValue()!=-1){

coordinatesOfEnemySoldiercanNotEat.add(down);

}

if(right.getValue()!=0 && right.getValue() !=-1 && right.getValue()!=player\_color && twoRight[0].getValue()!=0 && twoRight[0].getValue()!=-1){

coordinatesOfEnemySoldiercanNotEat.add(right);

}

if(left.getValue()!=0 && left.getValue() !=-1 && left.getValue()!=player\_color && twoLeft[0].getValue()!=0 && twoLeft[0].getValue()!=-1){

coordinatesOfEnemySoldiercanNotEat.add(left);

}

}

//updates the list of ally soldier

public void findAllySoldier(){

if(up.getValue()==player\_color){

allySoldier.add(up);

}

if(down.getValue()==player\_color){

allySoldier.add(down);

}

if(right.getValue()==player\_color){

allySoldier.add(right);

}

if(left.getValue()==player\_color){

allySoldier.add(left);

}

}

//updates the list of notSafeMove

public void findMoveNotSafe(){

if(up.getValue()==0 && (twoUp[0].getValue()!=0 && twoUp[0].getValue()!=-1 && twoUp[0].getValue()!=player\_color)){

notSafeMove.add(up);

}

if(down.getValue()==0 && (twoDown[0].getValue()!=0 && twoDown[0].getValue()!=-1 && twoDown[0].getValue()!=player\_color)){

notSafeMove.add(down);

}

if(right.getValue()==0 && (twoRight[0].getValue()!=0 && twoRight[0].getValue()!=-1 && twoRight[0].getValue()!=player\_color)){

notSafeMove.add(right);

}

if(left.getValue()==0 && (twoLeft[0].getValue()!=0 && twoLeft[0].getValue()!=-1 && twoLeft[0].getValue()!=player\_color)){

notSafeMove.add(left);

}

}

//this function is to find if the soldier is stuck

public boolean isSoldierStuck(){

if(possibleMoves.isEmpty() && possibleEatMoves.isEmpty()){

return true;

}

return false;

}

//this function is to find if the soldier is not in danger

public boolean isSoldierNotInDanger(){

if((up.getValue()==0 && down.getValue()==opponent\_color) || (down.getValue()==0 && up.getValue()==opponent\_color) || (right.getValue()==0 && left.getValue()==opponent\_color) || (left.getValue()==0 && right.getValue()==opponent\_color)){

return false;

}

return true;

}

//this function calculates the weight of the soldierMoves

public int weightSoldierMoves(){

int weight=0;

weight+=possibleMoves.size()\*25;

weight+=possibleEatMoves.size()\*100;

weight+=allySoldier.size()\*50;

weight-=coordinatesOfEnemySoldiercanNotEat.size()\*35;

weight-=notSafeMove.size()\*100;

// if(isSoldierNotInDanger()) weight-=100;

return weight;

}

public List<Coordinate> getCoordinatesOfEnemySoldier() {

return coordinatesOfEnemySoldiercanNotEat;

}

public List<Coordinate[]> getPossibleEatMoves() {

return possibleEatMoves;

}

public List<Coordinate> getPossibleMoves() {

return possibleMoves;

}

public Coordinate getSoldierCoordinate() {

return soldierCoordinate;

}

public List<Coordinate> getAllySoldier() {

return allySoldier;

}

@Override

public String toString(){

return("\n"+"up: "+up+" down: "+down+" left: "+left+" right: "+right+"\n"+"twoUp: "+twoUp[0]+" twoDown: "+twoDown[0]+" twoLeft: "+twoLeft[0]+" twoRight: "+twoRight[0]+"\n"+"possibleMoves: "+possibleMoves+"\nnotSafeMove: "+notSafeMove+"\n"+"possibleEatMoves: "+possibleEatMoves+"\n"+"coordinatesOfEnemySoldiercanNotEat: "+coordinatesOfEnemySoldiercanNotEat+"\nallySoldier: "+allySoldier+"\n"+"soldierCoordinate: "+soldierCoordinate);

}

}

package code**;**

**import** java**.**util**.**Stack**;**

public class Negamax **{**

public static int numofRecursion **=** **-**1**;**

public int bestMoveIndex**=**0**;**

public int bestpop**=**0**;**

private int**[][]** board**;**

private int depth**;**

private Computer me**;**

private Computer opponent**;**

private GameButton **[][]**gBoard**;**

private boolean isEaten**;**

public Negamax**(**int**[][]** board**,**GameButton **[][]**gBoard**,** int depth**,**Computer me**,** Computer opponent**,** boolean isEaten**)** **{**

**this.**depth **=** depth**;**

**this.**isEaten **=** isEaten**;**

**this.**board **=** **new** int**[**board**.**length**][**board**[**0**].**length**];**

copyBoard**(**board**,this.**board**);**

**this.**gBoard **=** **new** GameButton**[**gBoard**.**length**][**gBoard**[**0**].**length**];**

copyGBoard**(**gBoard**);**

**this.**me **=** **new** Computer**(**me**.**getPlayer\_color**(),** me**.**getMap**(),** me**.**getDifficulty**());**

**this.**me**.**copy**(**me**);**

**this.**me**.**setBoard**(**board**);**

**this.**me**.**setGBoard**(**gBoard**);**

**this.**opponent **=** **new** Computer**(**opponent**.**getPlayer\_color**(),** opponent**.**getMap**(),** opponent**.**getDifficulty**());**

**this.**opponent**.**copy**(**opponent**);**

**this.**opponent**.**setBoard**(**board**);**

**this.**opponent**.**setGBoard**(**gBoard**);**

numofRecursion**++;**

//System.out.println("numofRecursion: "+numofRecursion);

**}**

public int negamax**(){**

**if** **(**depth **==** 0 **||** gameOver**())** **return** evaluate**();**

int maxEval **=** Integer**.**MIN\_VALUE**;**

int bestPopEval **=** Integer**.**MIN\_VALUE**;**

int numOfpop **=** 0**;**

int**[]** data**;**

Stack**<**SoldierMoves**>** soldier **=** **new** Stack**<>();**

me**.**findAllPossibleSoldier**();**

soldier **=** **(**Stack**<**SoldierMoves**>)**me**.**getSoldierMovesStack**().**clone**();**

**while(!**soldier**.**isEmpty**()){**

SoldierMoves moves **=** soldier**.**pop**();**

int eval**=**Integer**.**MIN\_VALUE**;**

int numOfindex **=** 0**;**

int bestMoveIndexEval **=** Integer**.**MIN\_VALUE**;**

int[][] newBoard;

if(isEaten && me.getIsSoldierNotLeftFirstTime()<=1){

//data = me.play(isEaten, board, gBoard, opponent);

data = me.playNegamax(true,null,null,null,board, gBoard);

newBoard = new int[board.length][board[0].length];

copyBoard(board,newBoard);

updateBord(newBoard, data, me.getPlayer\_color());

eval = -1 \* new Negamax(newBoard, gBoard, depth-1,opponent, me, false).negamax();

maxEval = Math.max(maxEval, eval);

}

else if(moves.getPossibleEatMoves()!=null && !moves.getPossibleEatMoves().isEmpty()){

for(int i = 0; i<moves.getPossibleEatMoves().size();i++){

//data = me.play(false, board, gBoard, opponent);

data = me.playNegamax(false,moves.getPossibleEatMoves().get(i)[0],moves.getPossibleEatMoves().get(i)[1],moves.getSoldierCoordinate(),board, gBoard);

newBoard = new int[board.length][board[0].length];

copyBoard(board,newBoard);

updateBord(newBoard, data, me.getPlayer\_color());

eval = -1 \* new Negamax(newBoard, gBoard, depth-1, opponent, me, true).negamax();

maxEval = Math.max(maxEval, eval);

if(maxEval<bestMoveIndexEval){

maxEval=bestMoveIndexEval;

bestMoveIndex=numOfindex;

}

numOfindex++;

}

}

else if(moves.getPossibleMoves()!=null && !moves.getPossibleMoves().isEmpty()){

for(int i = 0; i<moves.getPossibleMoves().size();i++){

//data = me.play(false, board, gBoard, opponent);

data = me.playNegamax(false,moves.getPossibleMoves().get(i),null,moves.getSoldierCoordinate(),board, gBoard);

newBoard = new int[board.length][board[0].length];

copyBoard(board,newBoard);

updateBord(newBoard, data, me.getPlayer\_color());

eval = -1 \* new Negamax(newBoard, gBoard, depth-1, opponent, me, false).negamax();

maxEval = Math.max(maxEval, eval);

if(maxEval<bestMoveIndexEval){

maxEval=bestMoveIndexEval;

bestMoveIndex=numOfindex;

}

numOfindex++;

}

}

if(maxEval<bestPopEval){

bestPopEval = maxEval;

bestpop = numOfpop;

}

numOfpop++;

}

return maxEval;

}

private int evaluate() {

int meSoldiers = (me.getSoldierLeft()+me.getSoldier\_on\_board())\*10;

int opponentSoldiers = (opponent.getSoldierLeft()+opponent.getSoldier\_on\_board())\*10;

// if (meSoldiers == 0) return Integer.MIN\_VALUE;

// if (opponentSoldiers == 0) return Integer.MAX\_VALUE;

int meEval = evaluateBordByPlayer(me);

int opponentEval = evaluateBordByPlayer(opponent);

return meSoldiers+meEval - opponentSoldiers+opponentEval;

}

private int evaluateBordByPlayer(Computer computer){

int eval = 0;

Stack<SoldierMoves> soldiers;

if(computer.getSoldierMovesStack()==null || computer.getSoldierMovesStack().isEmpty()) return 0;

soldiers = computer.getSoldierMovesStack();

while(!soldiers.isEmpty()){

eval += soldiers.pop().weightSoldierMoves();

}

return eval;

}

private void updateBord(int[][] board,int[] data, int player\_color) {

if(data[6]==0 && data[2]!=0){

board[data[0]][data[1]]=board[data[2]][data[3]];

board[data[2]][data[3]]=0;

}

else if(data[6]==1){

board[data[0]][data[1]]=board[data[2]][data[3]];

board[data[2]][data[3]]=0;

board[data[4]][data[5]]=0;

}

else{

board[data[0]][data[1]] = player\_color;

}

}

public void copyBoard(int[][] board, int[][] newBoard) {

for (int i = 0; i < board.length; i++) {

for (int j = 0; j < board[i].length; j++) {

newBoard[i][j] = board[i][j];

}

}

}

public void copyGBoard(GameButton [][]gBoard) {

for (int i = 0; i < gBoard.length; i++) {

for (int j = 0; j < gBoard[i].length; j++) {

this.gBoard[i][j] = gBoard[i][j];

}

}

}

public boolean gameOver() {

return me.isSoldiersLeft() || opponent.isSoldiersLeft();

}

}

package code**;**

**import** java**.**awt**.\*;**

**import** javax**.**swing**.**ImageIcon**;**

public class Soldier**{**

private Image img**;**

private final String color**;**

private int x**,**y**;**

Image redSoldier **=** **new** ImageIcon**(**"././img/Soldier\_red\_New.png"**).**getImage**();**

Image blackSoldier **=** **new** ImageIcon**(**"././img/Soldier\_black\_New.png"**).**getImage**();**

public Soldier**(**Image img**,**String color**,**int x**,**int y**)** **{**

**this.**img**=**img**;**

**this.**color **=** color**;**

**this.**setX**(**x**);**

**this.**setY**(**y**);**

**}**

public Soldier**(**String color**)** **{**

**this.**color **=** color**;**

**if(**color**.**equals**(**"red"**))** img **=** redSoldier**;**

**else** img **=** blackSoldier**;**

**}**

public Image getImg**()** **{**

**return** img**;**

**}**

public void setImg**(**Image img**)** **{**

**this.**img **=** img**;**

**}**

public String getColor**()** **{**

**return** color**;**

**}**

public int getX**()** **{**

**return** x**;**

**}**

public void setX**(**int x**)** **{**

**this.**x **=** x**;**

**}**

public int getY**()** **{**

**return** y**;**

**}**

public void setY**(**int y**)** **{**

**this.**y **=** y**;**

**}**

**}**

package code**;**

public class Coordinate **{**

private int row**,**column**,**value**;**

public Coordinate**(**int row**,** int column**,**int value**)** **{**

**this.**row **=** row**;**

**this.**column **=** column**;**

**this.**value**=**value**;** //-1 null 0 empty 1 red 2 black

**}**

public void setValue**(**int value**)** **{**

**this.**value **=** value**;**

**}**

public void setRow**(**int row**){**

**this.**row**=**row**;**

**}**

public void setColumn**(**int column**){**

**this.**column**=**column**;**

**}**

public int getRow**(){**

**return** row**;**

**}**

public int getColumn**(){**

**return** column**;**

**}**

public int getValue**()** **{**

**return** value**;**

**}**

@Override

public boolean equals**(**Object other**){**

**if(this.**row **==** **((**Coordinate**)**other**).**row **&&** **this.**column **==** **((**Coordinate**)**other**).**column**){**

**return** **true;**

**}**

**return** **false;**

**}**

@Override

public String toString**(){**

**return** **(**"row:"**+**row**+**" column:"**+**column**+**" value:"**+**value**);**

**}**

**}**

package code**;**

**import** java**.**awt**.\*;**

**import** javax**.**swing**.\*;**

public class GameButton **extends** JButton **{**

private Image img**;**

private Soldier soldier**=null;**

private int weight**=**0**;**

public GameButton**(**Image img**,**Soldier soldier**)** **{**

**this.**img**=**img**;**

**this.**soldier**=**soldier**;**

**}**

public GameButton**(**Image img**){**

**this.**img**=**img**;**

**}**

public GameButton**(**Soldier soldier**){**

**this.**soldier**=**soldier**;**

**this.**img**=**soldier**.**getImg**();**

**}**

public GameButton**(){}**

public Image getImg**()** **{**

**return** img**;**

**}**

public void setImg**(**Image img**){**

**this.**img **=** img**;**

**}**

public int getWeight**()** **{**

**return** weight**;**

**}**

public void setWeight**(**int weight**)** **{**

**this.**weight **=** weight**;**

**}**

public Soldier getSoldier**(){**

**return** soldier**;**

**}**

public void setSoldier**(**Soldier soldier**){**

**this.**soldier**=**soldier**;**

**}**

public void paintComponent**(**Graphics g**){**

**super.**paintComponent**(**g**);**

g**.**drawImage**(**img**,** 0**,** 0**,** getWidth**(),** getHeight**(),** **null);**

**if(!(**soldier**==null)){**

g**.**drawImage**(**soldier**.**getImg**(),** 0**,** 0**,** getWidth**(),** getHeight**(),** **null);**

**}**

**}**

**}**

# נספחים

GitHub של הפקוירט:

<https://github.com/netnis22/java-queah-game>