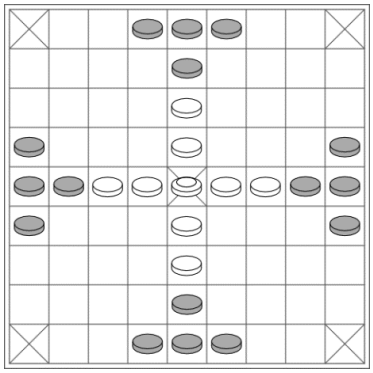
עבודת גמר

לקבלת תואר

טכנאי תוכנה

הנושא: פיתוח משחק Tablut AI



המגיש : חודרה אופיר

ת.ז. המגיש: 325187284

שמות המנחים: מיכאל צ'רנלובסקי

תאריך הגשה: 25/04/2021

תוכן עניינים

תקציר

התוצר הסופי של פרויקט זה הוא לבנות desktop application בעל ממשק נוח ומעוצב למשתמש למשחק Tablut המפורסם.

מטרת הפרויקט היא לבחון ולאתגר את כישורי התכנות שלי באמצעות הצבת יעדים תכנותיים ודרכי התמודדות כדי להגיע ליעדים האלה, דרכים אשר לא התנסיתי בהם בעבר.

אחד האתגרים בפרויקט היה לפתח שחקן ממוחשב אשר יודע "לחשוב" בצורה מלאכותית על מנת

לבחור מהלכים לפי חוקי המשחק אשר יובילו אותו לניצחון במשחק. מעולם לא התנסיתי

בכתיבת אלגוריתם הפותר בעיה כזו, ועל כן אחת ממטרות העל הייתה להתמודד עם כתיבת אלגוריתם זה.

באפליקציה זאת יהיו 2 מצבי משחק:

א. לשחק Tablut שני שחקנים אנושיים.
ב. לשחק נגד בוט דמוי שחקן שאותו אפתח בעזרת בינה מלאכותית.

כמו כן, במהלך הפרויקט איישם את החומר הנלמד במהלך שנת הלימודים:

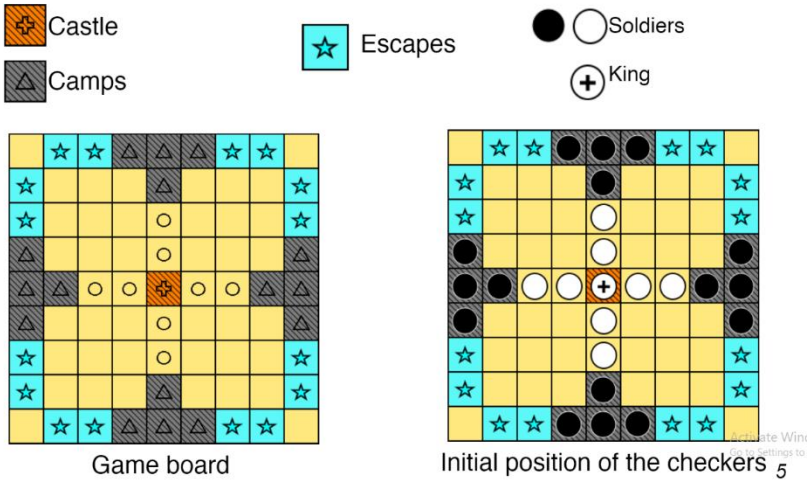
- תכנות בשפה מונחית עצמים
- בחירת מבני נתונים
- מימוש אלגוריתמיקה יעילה
- פיתוח תוכנה עם ארכיטקטורות מוכרות של הנדסת תוכנה

תיאור נושא הפרויקט

המשחק הנורדי טאבלוט הוא ממשפחת המשחקים hnefatafl games. משחק זה הומצא בשנת 1732 ע"י ילד שוודי בשם קארל לינאוס.

לוח המשחק הוא לוח עץ בצורת ריבוע המורכב מ81 משבצות 9X9.

כלי המשחק כוללים: מלך אחד , שמונה מגנים בצבע לבן ו16 תוקפים בצבע שחור.



- המשבצות הכחולות נקראות "תאי בריחה".
- המשבצות השחורות נקראת מחנות.
- המשבצת הכתומה נקראת "הטירה".

המשחק משוחק ע"י שני שחקנים ולכל שחקן דרך ניצחון שונה:

א. <u>שחקן מגן:</u> הוא השחקן עם הכלים הלבנים ומטרתו היא לקדם את המלך שלו לאחד מקודקודי הלוח. לשחקן זה 8 כלים שנקראים "מגנים" שמטרתם לשמור על המלך מפני "לכידה" על ידי כלי השחקן השחור הנקראים "תוקפים". שחקן זה מנצח כאשר המלך הגיע לאחד מצדדי הלוח.

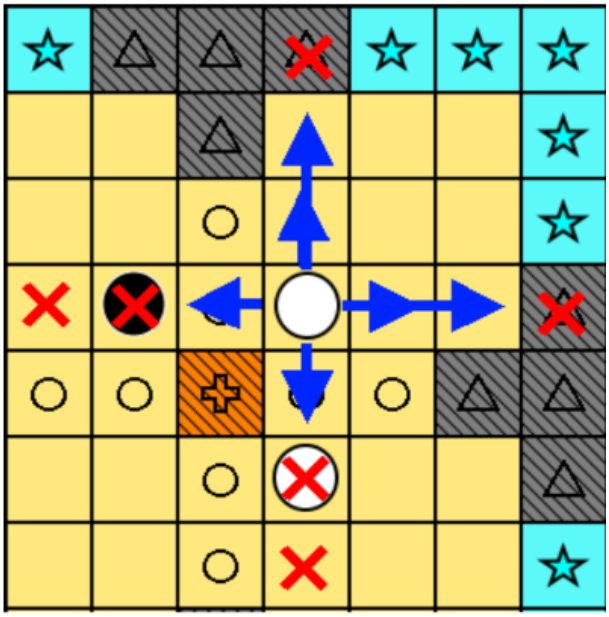
ב. <u>שחקן תוקף:</u> דרך שחקן זה לנצח היא ללכוד את המלך של השחקן היריב ולמנוע ממנו להגיע לאחד מהתאים במסגרת הלוח.

חוקי המשחק

השחקן התוקף מבצע את המהלך הראשון במשחק.

הזזת כלי המשחק

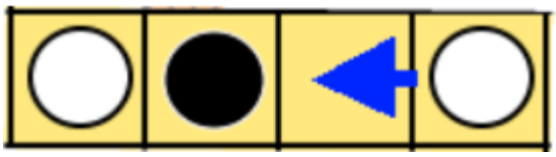
1. כל הכלים במשחק יכולים לנוע מס' משבצות לא מוגבל ימינה, שמאלה למעלה ולמטה (לא באלכסון). אך אסור "לדלג" על כלי משחק אחרים.
2. אסור לאף כלי משחק לנוע למשבצת המרכזית הנקראת "הטירה" – משבצת המוצא של המלך. אפילו למלך כאשר לאחר שעזב אותה לראשונה.
3. לתוקפים מותר לנוע במחנות ההתחלה שלהם עד שהם עוזבים אותם, לאחר מכן הם לא יכולים לחזור אליהם.



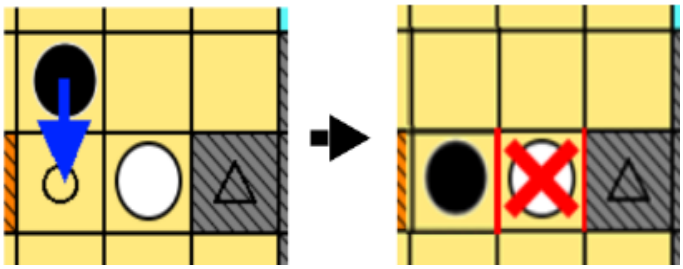
לכידות

1. תוקפים ומגנים נלכדים כאשר הם מוקפים ע"י כלי 2 מכלי היריב בשני משבצות סמוכות מנוגדות. שיטה נוספת היא להקיף אותם ע"י שחקן יריב אחד ובצד השני "טירה" או "מחנה".
2. לכידה חייבת להיות "אקטיבית" כלומר אם חייל מביא את עצמו למצב של לכידה בין שני חייליו יריב הוא נשאר בחיים.

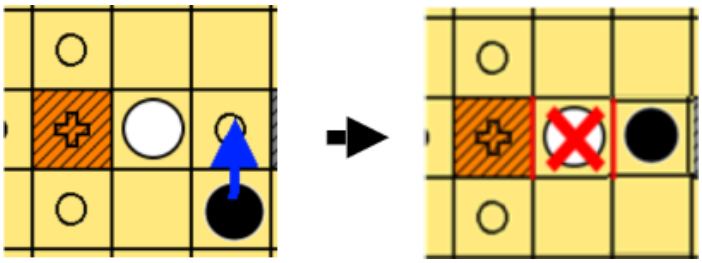
לכידה פשוטה



לכידה בעזרת מחנה

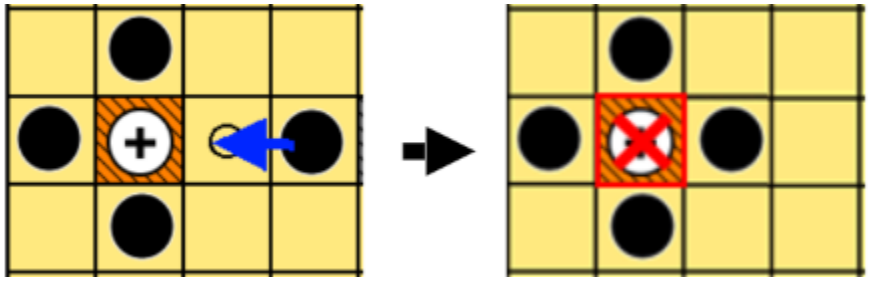


לכידה בעזרת הטירה

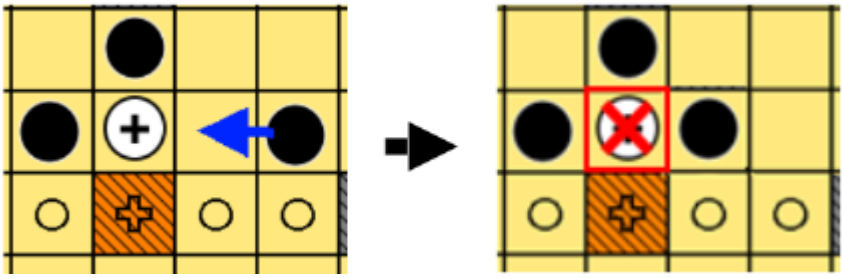


לכידות המלך

א. כאשר המלך לא נמצא ב"טירה" הוא נלכד כמו שאר כלי המשחק.
ב. כאשר המלך ממוקם במשבצת המרכזית "הטירה" הוא נלכד כאשר הוא מוקף בארבעה תוקפי היריב.



ג. כאשר המלך ממוקם בתא הסמוך לטירה, בכדי ללכוד אותו דרוש להקיפו בשלושה "תוקפים" בשלושת התאים הסמוכים אליו שהם לא הטירה.



סיום המשחק

א. המגנים והמלך מנצחים במשחק כאשר המלך מגיע לאחת מהמשבצות במסגרת הלוח. לעומתם התוקפים מנצחים כאשר מצליחים ללכוד את המלך.
ב. אם אחד מהשחקנים לא יכול לבצע מהלך, הוא מפסיד.
ג. לאחר 100 מהלכים מוכרז תיקו.

רקע תאורטי

משחקי טאפל (הידוע גם בשם משחקי hnefatafl) הם משפחה משחקי אסטרטגיה אשר משוחקים על לוח משחק משובץ או סריג עם שני צבאות של מספרים לא אחידים. ככל הנראה הם מבוססים על המשחק הרומי Ludus latrunculorum. משחקים במשפחת tafl נערכו ב נורבגיה , שבדיה , דנמרק , איסלנד , בריטניה , אירלנד , ו לפלנד . המשחק של טאפל הוחלף בסופו של דבר על ידי שחמט במאה ה -12, אך גרסת הטפל של טבלוט, הייתה במשחק עד לפחות 1700.

הכללים לטבלוט נכתבו על ידי חוקר הטבע השוודי לינאוס בשנת 1732, ואלה תורגמו מלטינית לאנגלית בשנת 1811. כל משחקי הטפל המודרניים מבוססים על תרגום 1811, שהיה בו שגיאות רבות. כללים חדשים נוספו לתיקון הבעיות הנובעות משגיאות אלה, מה שהוביל ליצירת משפחה מודרנית של משחקי טאפל אשר משוחקת כיום ברחבי העולם.

סיבה לבחירת נושא הפרויקט:

במהלך תהליך בחירת נושא הפרויקט חיפשתי במשך ימים משחק אשר מסקרן אותי לכתוב לו את הבינה המלאכותית. כאשר נחשפתי למשחק הTablut, הורדתי אפליקציה לנייד שלי ושיחקתי בו מספר משחקים ביום לאורך שבועות כדי ללמוד את חוקי המשחק ואת האסטרטגיות האנושיות לניצחון במשחק. למשחק זה המון טכניקות ותכסיסים והסקתי שבניית בוט ממוחשב יהיה אתגר קשה ומעניין.

לכן לקחתי על עצמי את האתגר והתחלתי בבניית הפרויקט.

תהליכים עיקריים בפרויקט:

א. בניית ממשק משתמש למשחק בין 2 משתמשים אנושיים.
ב. חקר אלגוריתמים לבינה מלאכותית.
ג. בחירת מבני נתונים מתאימים למימוש הבינה מלאכותית.
ד. יישום הכלים שרכשתי בחקר – בניית הבינה לתוקפים ומגנים.
ה. סיום בניית ממשק משתמש גרפי למשחק נגד המחשב.
ו. ניתוח ביצועיו של התוצר הסופי ושיפורו בדרכים שונות.
ז. תיקון תקלות ובאגים

תיאור הבעיה האלגוריתמית

במהלך תכנון וכתיבת הפרויקט נתקלתי בכמה בעיות אלגוריתמיות, להן הייתי צריך למצוא פתרונות אלגוריתמיים. בעמודים הבאים אפרט על בעיות אלה ובהמשך אתאר את הפתרונות בהם השתמשתי.

מימוש חוקי המשחק

בפרויקט שלי אממש את כל חוקי המשחק באופן ממוחשב.

מימוש זה כולל: בדיקת מהלכים חוקיים, מצבי ניצחון שונים ותיקו.

למשחק Tablut המון חוקים לכן אשקיע מאמץ רב בשלב הראשון של פיתוח המשחק למשחק בין אנשים.

על המערכת לדעת כיצד לבדוק את תקינותם של המהלכים בהתאם לחוקים הנ"ל על מנת לאפשר משחק תקין וגם הכרת החוקים תשמש את חישובי המהלכים של השחקן הממוחשב.

ייצוג לוח המשחק

במהלך כתיבת ותכנון הפרויקט עלתה הבעיה כיצד נכון לייצג את לוח המשחק והשחקנים בשפת התכנות. לדוגמה, באיזה מבנה נתונים עדיף להשתמש כדי לייצג את הלוח? האם מבני הנתונים יעילים וחסכוניים מספיק כדי שיתאפשר לשחקן הממוחשב לבצע את פעולותיו במהירות הנדרשת ? באיזה דרך לממש את ייצוג הלוח בצד שרת ובצד לוקח? בעיה זאת משמעותית כדי ליצור מערכת שעובדת באופן מהיר, יעיל וחסכוני

מימוש מצב של שחקן נגד מחשב

על המערכת לאפשר משחק של אדם נגד שחקן ממוחשב. לכן, כמובן, אחת מהבעיות הכי משמעותיות של הפרויקט היא לכתוב אלגוריתם מתאים לשחקן ממוחשב. כלומר, יש לחשוב על אלגוריתם אשר יאפשר למערכת "לחשוב" מהו המהלך הטוב ביותר שניתן לבצע – המהלך שיקדם את הצד של השחקן הממוחשב לניצחון המשחק .על האלגוריתם להיות יעיל וחסכוני במשאבים שכן עליו לספק לשרת את המהלך הנבחר באופן מהיר מאוד – על מנת לספק למשתמש חוויה טובה מהשימוש במערכת, שהרי משתמשים יעדיפו משחק מהיר ומעניין נגד AI מהיר וחכם.

כעת אחשב את מספר מצבי הלוח האפשריים בדרך נאיבית של המשחק Tablut:

סוג התא	חייל לבן "מגן"	חייל שחור "תוקף"	מלך	כמות משבצות
הטירה	-	-	✓	1
מחנה	-	✓	-	16
משבצת רגילה	✓	✓	✓	44

מטבלה זאת נוכל להסיק לחשב את כמות הלוחות האפשריים:

$$UB_{naive} = 2 \cdot 2^{16} \cdot 3^{20} \cdot 4^{44} \approx 10^{41}$$

כעת נשווה את כמות לוחות האפשריים ביחס למשחקים שונים:

Nine Men's Morris	English Draughts	International Draughts	Othello	Chess	Go
3×10^{11}	5×10^{20}	10^{30}	10^{28}	10^{43}, 10^{50}	2×10^{170}

מהשוואה זאת ניתן להסיק שהמשחק הוא משחק מורכב כתוצאה מכמות מהלכים רבים לכל כלי בכל שלב של המשחק

- מספר המהלכים האפשריים המקסימלי לכלי משחק הוא 16 מהלכים
- מספר המהלכים האפשריים הממוצע לשחקן בתור במשחק הוא 80

לכן מבחינתי לבנות בינה אשר תשחק בחוכמה ובמהירות זה אתגר גדול ואני שמח על תהליך הלמידה הנרחב ומתוצאות הפרויקט שאסקור בהמשך הספר.

סקירת אלגוריתמים בתחום הבעיה

במהלך החקר נחשפתי לשלושה אלגוריתמים עיקריים אשר מתאימים לצרכי הפרויקט שלי, בניית בינה מלאכותית למשחק קופסא.

בפרק זה אפרט בקצרה על שלושתם ובהמשך אדגים והסביר בהרחבה את האלגוריתם שבחרתי ואת הסיבות לבחירתו.

אלגוריתם מינימקס

עבור כל צומת בעץ המשחק באלגוריתם זה ישנו ערך שמייצג את הערכת הלוח לפי פונקציית הערכה אשר מעריכה מספרית את מצב הלוח עבור השחקן הנוכחי בהתאם לחוקי המשחק .
הרעיון המרכזי באלגוריתם זה בא לידי ביטוי לאחר שנבנה עץ עבור המקרים השונים שבהם הלוח יכול להימצא מהמצב הנוכחי, כל רמה בתורה, כאשר כל רמה מייצגת שחקן שונה (השחקן שעבורו מחשבים את המהלך הטוב ביותר והשחקן הנגדי), "בוחרת" במצב הלוח הטוב ביותר עבורה. לכן, רמה שמייצגת את השחקן הנוכחי תבחר בערך המקסימלי מבניה, ורמה שמייצגת את השחקן הנגדי תבחר בערך המינימלי מבניה – שכן, השחקן היריב ירצה את המצב הטוב בשבילו, כלומר המצב הרע בשביל השחקן הנוכחי.

אלגוריתם מונטה קרלו

אלגוריתם זה הוא אלגוריתם אשר הומצא עבור משחקים מורכבים אשר מכילים כמות מהלכים אפשריים גדולה מאוד לכל מצב בלוח. הרעיון העיקרי באלגוריתם זה הוא רעיון הרנדומליות. עץ כזה, אם נכתב נכון, לא זקוק אפילו לפונקציית הערכה כלשהי משון שהוא נסמך על רנדומליות של מצבי משחק שהשחקנים יכולים להגיע אליהם. אלגוריתם Carlo Monte משתמש גם הוא בעץ חיפוש, אך המבנה שלו שונה מהעצים שהאלגוריתמים הקודמים.

Reinforcement machine learning

Learning Reinforcement או למידת חיזוק בעברית , הוא תחום בלמידת מכונה בו האלגוריתם מנסה לקבל את התגמול המרבי עבור מעשיו ע"י נקיטת פעולות בסביבה נתונה.
המתכנת נותן משימה אותה האלגוריתם צריך לבצע, אך הוא לא נותן לאלגוריתם ידע מוקדם או רמזים כיצד לפתור את המשימה שהוקצבה לו, רק בעזרת ניסוי וטעייה ומערכת של תגמולים ועונשים אותה המתכנת יוצר. האלגוריתם מצליח לפענח כיצד הוא צריך לפעול בסביבה בה הוא נמצא. מערכת התגמולים והעונשים של האלגוריתם עובדת כך שאם האלגוריתם מצליח לבצע את המשימה שהמתכנת נתן לו או אם הוא מצליח להתקדם במשימה, האלגוריתם מקבל תגמול חיובי עבור מעשיו , אך אם האלגוריתם אינו מצליח במשימה שלו הוא מקבל עונש, וכך האלגוריתם לומד באיזה התנהגות כדאי לו לנקוט כדי למקסם את התגמול שלו .
האלגוריתם מתחיל בהתנהגות רנדומלית אבל בעזרת מערכת זו של תגמולים ועונשים האלגוריתם יכול להגיע לרמות גבוהות של תחכום ואף ליכולות על אנושיות.

מושגים

מושגים בתחום המשחקים

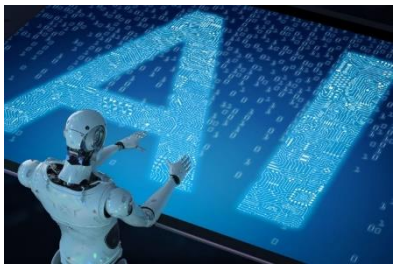
- **משחק מחשב** הוא תוכנה המהווה משחק, ומקיימת אינטראקציה מתמדת עם השחקן. משחק מחשב מופעל על-גבי מחשב אישי. משחק זה נשלט באמצעות ממשק משתמש.

- **משחק לוח** הוא משחק קופסה המיועד לשחקן אחד או יותר, המתנהל בעיקר, או באופן בלעדי, על-גבי לוח שסומן מראש לצורך זה, בהתאם לכללים קבועים מראש. בדרך כלל נעשה שימוש בכלי משחק שונים, כגון חיילי משחק וקוביות משחק.

- **אסטרטגיה (תכסיס)** היא תוכנית פעולה שמורה לשחקן איך לפעול בכל מצב אפשרי של המשחק. כאשר אין מעורב מזל במשחק, תוצאת המשחק נקבעת בצורה בלעדית על פי בחירת התכסיסים על ידי השחקנים .

מושגים בתחום התוכנה

- **ממשק משתמש** או באנגלית (Interface User (UI זהו החלק המוצג למשתמש בתוכנה, הדרך של המשתמש ליצור קשר עם התוכנה או האלגוריתם. הממשק בדרך כלל מכיל בתוכו כפתורים או שדות טקסט המאפשרים למשתמש לתקשר עם התוכנה באמצעות העכבר או מקשי המקלדת.



- **בינה מלאכותית Artificial intelligence AI** היא ענף של מדעי המחשב העוסק ביכולת לתכנת מחשבים לפעול באופן המציג יכולות המאפיינות את הבינה האנושית.
הגדרה רחבה יותר לתחום זה היא לגרום למכונה להתנהג בדרך שהייתה נחשבת לאינטליגנטית לו אדם התנהג כן.

- **DESKTOP APPLICATION** תוכנות אשר רצות באופן עצמאי על המחשב נייח או נייד. בניגוד לאתרי אינטרנט שרצות על דַּפְדְּפָן. לדוגמא: Skype or visual Studio.

- **פסאודו קוד** הוא תיאור מצומצם ולא רשמי לאלגוריתם של תוכנית מחשב. פסאודו קוד משתמש בקונבנציות של שפות תכנות, אך מיועד לקריאה של בני אדם ולא לקריאה על ידי מחשב.

- **BOARD** לוח המשחק.

- **MASKS** – ערכים בעלי סיביות לפי סידור מסוים, המיועדים להשוואה /השמה עם /על ערכים אחרים.

- **HASHMAP** – מבנה נתונים, שבה לכל מפתח כניסה ערך ייחודי.

- **OPPONENT-PLAYER** – שחקן נוכחי/יריב, בהתאם.

- **TREE GAME** – עץ המשחק

- **BRANCH** – ענף משחק מצד שחקן בעץ משחק.

- **EVALUATE/SCORE** – ציון שניתן באמצעות פונקציית הערכה הקובע עד כמה מצב במשחק טוב עבור אחד השחקנים

- **COORDINATES** – קורדינטות לוח.

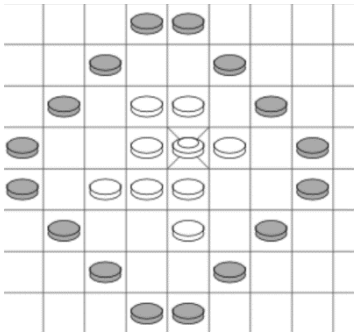
- **MOVES** – מהלכים חוקיים של השחקן

אסטרטגיה אנושית לניצחון במשחק

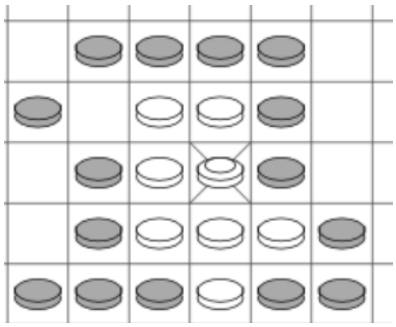
בפרק זה אסקור אסטרטגיות אנושיות אשר רכשתי בשלב הראשוני של כתיבת הפרויקט. בתור כותב העבודה היה לי חשוב לדעת מה הן האסטרטגיות הנפוצות של שחקנים מקצוענים על מנת לכתוב פונקציית הערכה טובה ביותר עבור ציון לוח משחק. אני מדגיש, בחלק זה לא הרחבתי על ה"תכסיסים" שאני השתמשתי בהם באלגוריתם הראשי אלא רק אסטרטגיות אנושיות.
פירוט פונקציות הערכה יהיו בפרק של האלגוריתם הראשי.

אסטרטגיות אנושיות עבור השחקן השחור

- **סגירת שרשרת סביב המלך:**
המטרה הראשונה של השחקן השחור היא לסגור שרשרת בלי ניתנת לשבירה מסביב למלך והכוחות הלבנים שלו. ככל שהשרשרת נשארת שלמה, המלך אינו יכול לברוח.
כל שנשאר לו לעשות זה להגיע לתוצאת תיקו או לחכות שהכלי השחור יעשה מהלך לא נבון. דוגמא לשרשרת זאת:



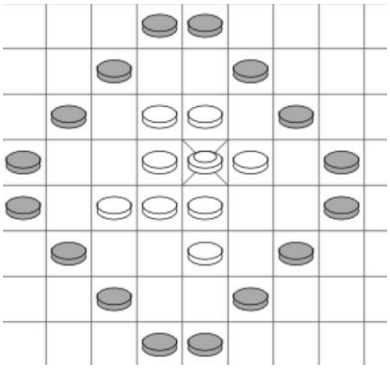
- **תקיפת המלך:**
לאחר סגירת השרשרת. השלב העיקרי הוא ללכוד את החיילים הלבנים עד שהמלך נשאר לבדו ללא הגנה. שלב זה הוא מסובך ומורכב מכיוון שבכל שלב יש להבטיח את שלמותה של מרבית מהשרשרת סגורה. כמו כן, אסטרטגיה מוכרת היא באמצעות יכולת הפיתוי לגרום לשחקן הלבן לנסות לפרוץ את המחסום אך לטמון לו מלכודת.



אסטרטגיות אנושיות עבור השחקן הלבן

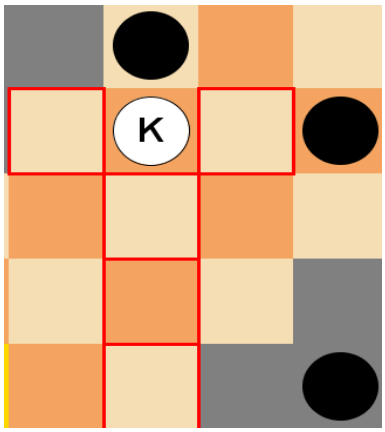
- **<u>כיבוש טריטוריה:</u>**

בתחילת המשחק, השחקן הלבן צריך למנוע בכל הכוח מן השחור לסגור סביבו שרשרת אשר תוביל לניצחון השחקן השחור. הדרך לכיבוש הטריטוריה החשוב עבור השחקן הלבן היא בשלבי המשחק הראשונים להגיע למשבצות ליד המחנות אשר יגבילו את תנועת היריב.



- **<u>תנועת המלך:</u>**

כתוצאה מהיתרון המספרי של החיילים השחורים. קיים ניסיון מצד אחד להגן ללא ערף על המהלך באמצעות הקפתו והקרבת כלים מגנים "רגילים" להילכד בהגנה על המלך.
מצד השני, יש לדאוג לאכילת הכלים של היריב על מנת להבטיח מרחב תנועתיות במרחב הלוח. מרחב התנועתיות של המלך חשוב, הסבר: נתאר מצב שמלך ניסה להגיע לאחת ממשבצות הבריחה ברביע הראשון של הלוח מצד ימין למעלה. ולאחר כישלון השחקן השחור יכול למנוע מהמלך לצאת מרביע זה. לכן כאשר המלך נע יש להבטיח לו "דרך מילוט" בחזרה למקום מבטחים – אזור בעל שליטה לבנה.



מבנה נתונים

רקע תיאורטי

מבנה הנתונים שבו בחרתי לממש את לוח המשחק במודל הלוגי הוא bit Board.

לוח ביטים הוא מבנה נתונים הנפוץ במערכות מחשב שמשחקות משחקי קופסא , כאשר כל סיבית מתאימה למרחב או לחתיכת לוח המשחק. זה מאפשר לפעולות סיביות מקבילות לקבוע או לשאול על מצב המשחק, או לקבוע מהלכים או משחקים במשחק.

לוחות סיביות יעילים במיוחד כאשר הסיביות המשויכות של מצבים קשורים שונים על הלוח משתלבות במילה אחת או במילה כפולה של ארכיטקטורת המעבד, כך שניתן להשתמש במפעילים סיביים בודדים כמו AND ו- OR לבניית או שאילתה מצבי משחק.

מימוש בפרויקט

לוח הTablut מורכב בפרויקט שלי ע"י 4 לוחות ביטים:

א. לוח הכלים הלבנים BitSet whitesPawns

ב. לוח הכלים השחורים BitSet blackPawns

ג. לוח כלי המלך BitSet kingPawn

ד. לוח כל כלי המשחק. BitSet board

גודל כל אחד הוא כמספר התאים בלוח המשחק – 81. כל ביט – ספרה בינארית תייצג משבצת ותהיה דלוקה אם חייל עומד על המשבצת וכבויה אם לא.

ייצוג בינארי של לוחות הסיביות במצב ההתחלתי של המשחק:

לוח שחקן לבן כולל המלך

```
100000001
000000000
000010000
000010000
001111100
000010000
000010000
000000000
100000001
```

לוח שחקן שחור

```
100111001
000010000
000000000
100000001
110010011
100000001
000000000
000010000
100111001
```

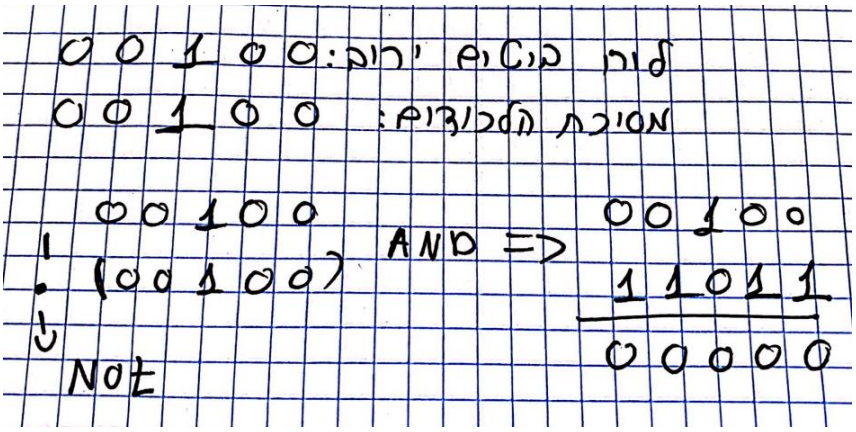
סיבות לבחירת מבנה הנתונים

1. כאשר מחשבים את כמות הסיביות שנדרשות על מנת לייצג את לוח המשחק מגיעים לנוסחה: 4*81 כמות סיביות בלוח סיביות * מספר לוחות הסיביות = שהם 324 ביטים בלבד (שווה ערך של 10 integers בלבד). כאשר חקרתי מבנה נתונים שונים לדוגמא כמו מטריצה, גיליתי שכמות הסיביות שנדרשות במימושים השונים גדולים משמעותית.

2. הסיבה העיקרית שבה בחרתי במבנה נתונים זה היא מושג הנקרא "מסכה" (mask).
המסכה היא אוסף של ביטים, שמאפשר לנו לבודד ולפעול על ביטים מסוימים. בעזרת פעולות לוגיות של "גם" ו"או". בפרויקט שלי בניתי מסיכות רבות אשר עזרו לממש את חוקי המשחק בקלות ובמהירות.

שימושים עיקריים במסכות:

- **כיבוי סיביות בלכידת כלים:**
כאשר מהלך מתבצע, פעולת ה"בצע מהלך" מחזירה לוח ביטים בו מודלקות הסיביות אשר מסמלות את משבצות היריב שבהם כלי נלכד. לכן, לאחר כל ביצוע מהלך אני משתמש במסכה "הכלים הלכודים" ומבצע בינה לבין לוח הביטים של היריב פעולה לוגית And Not על מנת לכבות את סיביות האלו ובכך ללכוד את הכלים היריבים.
חשוב להדגיש שניתן ללכוד יותר מכלי אחד במהלך ומימוש זה הוא יעיל וקל ואינו דורש התייחסות לכמות הכלים הלכודים במהלך.



דוגמא לשימוש בפעולה "גם" בין לוח הסיביות של היריב לבין לוח הסיביות של הכלים הלכודים לאחר שכל הביטים בו הוחלפו (פעולה Not), על מנת לכבות את הכלים הלכודים מלוח סיביות מייצג את חיילי היריב.

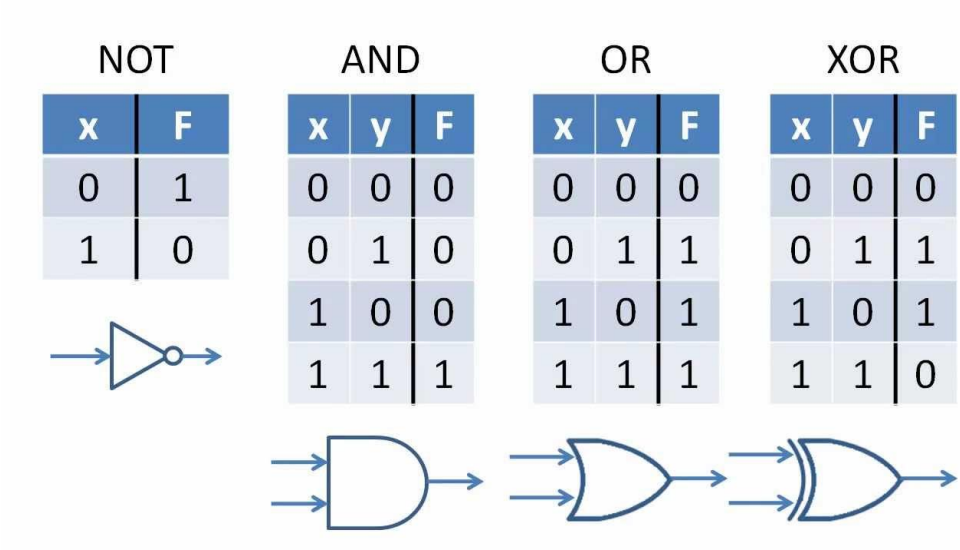
- **בדיקה האם סיבית דלוקה:**

כאשר אני רוצה לבדוק האם השחקן הלבן מנצח, אני נדרש לבדוק האם המלך נמצא באחד ממשבצות "הבריחה" במסגרת הלוח. לכן, אבנה מסיכה בגדול של הלוח ובה כל הביטים המייצגים מיקומים של משבצות אלו יודלקו ב1. כעת אפעיל את הפעולה "And" בין לוח הסיביות של המלך והמסכה. לאחר מכן אבחון את הלוח התוצאה של הפעולה הלוגית והראה האם אחד מהסיביות דלוקות, אם כן השחקן הלבן ניצח את המשחק. מכיוון שזה יסמל האם יש ביט אשר דלוק בשניים מלוחות אלו.

<u>בניית המסכות:</u>

בפיתוח הפרויקט בחרתי לבנות מבנה מטיפוס enum אשר מייצג שם לכל משבצת ע"י צירוף של אותו ומספר בדרך זאת:
("A1") A1 תא מספר 1 בעמודה הראשונה.
("E8") E8 תא מספר 8 בעמודה החמישית.

בטיפוס זה השתמשתי על מנת לבנות את המסכות ע"י מערך של שלמים שבו מיקומי הביטים הדלוקים במסיכה ולאחר מכן בנייה של לוח ביטים מהערכים בתאי המערך הזה. מוסבר בהרחבה בהמשך.



מבנה נתונים בחלק התצוגה

על מנת למנוע תלות בין החלק של המודל הלוגי backend והממשק הגרפי frontend,

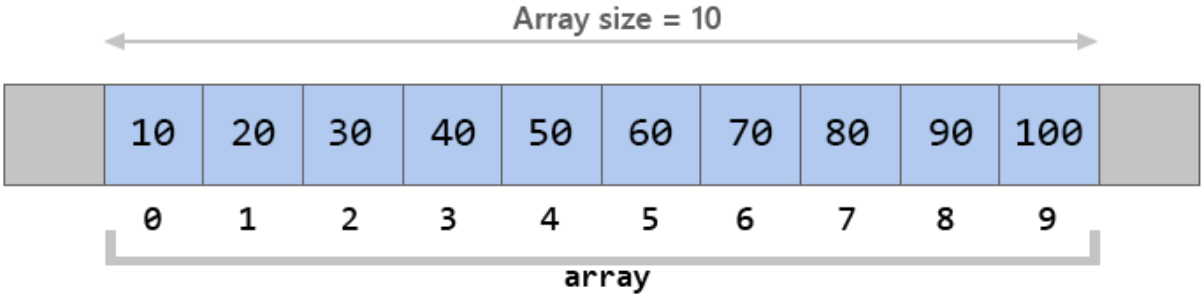
בחרתי לשמור מבנה נתונים נוסף בחלק התצוגה. מבנה נתונים זה אינו לוקח חלק באלגוריתם הבינה המלאכותית או במציאת המהלכים האפשריים והלכידות במשחק.

לכן על מנת לשמור את הנתונים בצד הלקוח בחרתי במבנה הנתונים מערך חד מימדי.

הסיבה לבחירת מבנה נתונים זה הוא שגודל המערך הוא סטטי מכיוון שגודל הלוח אינו משתנה.

כמו כן, מימוש הפונקציות המורכבות מתבצעות בחלק הלוגי ולא בחלק התצוגה לכן חסרונות היעילות של חיפוש במערך לעומת מבני נתונים שונים אינן מתקיימות בממשק הגרפי.

הנימוק לבחירת מערך חד מימדי ולא מטריצה היא רק מטעם נוחות, על מנת לפנות לביט מסויים בלוח סיביות במודל הלוגי אנו משתמשים באינדקס כמו במערך חד מימדי ולא בפנייה למס' שורה ומס' עמודה בנפרד. לכן היה לי נוח לשמור על אחידות בין מבני הנתונים השונים ולהימנע מחישובים מיותרים על מנת להמיר את ההיסטים בין מערך חד מימדי לדו מימדי (חישובים פשוטים ע"י חילוק ושארית בגודל צלע מערך).



כמו כן, בכל תא במערך נשמר סוג הכלי המשחק הקורדינטות x וy של מרכז התא בלוח המשחק הגרפי.

מיקום התא חשוב על מנת לדעת לזהות איזה כלי נלחץ כאשר הלקוח לוחץ על מקום מסוים בלוח.

בנוסף על כך, על מנת להציג ללקוח את המהלכים האפשריים שלו לאחר שלחץ על משבצת.

שמרתי במבנה נתונים נוסף את האינדקסים של המהלכים החוקיים שהשחקן יכול לבצע.

מבנה הנתונים הנבחר עבור צורך זה הוא Java ArrayList, אשר ממומש בחבילה java.util.ArrayList

במבנה נתונים זה לא מתבצע חיפוש או הכנסה/מחיקה מורכבת אלא רק הצגת המהלכים החוקיים בפונקצייה paint.

תרשימי UML

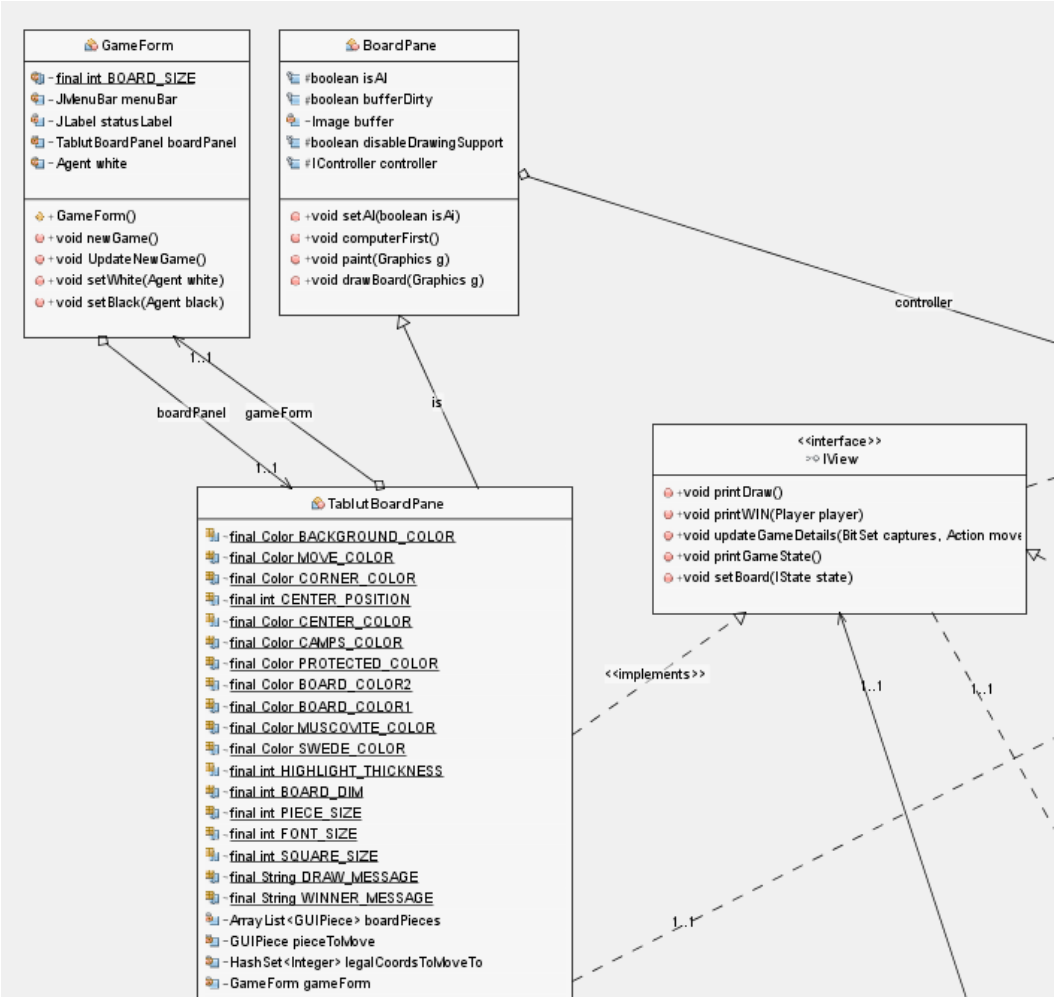
Class Diagram

בהנדסת תוכנה, דיאגרמת מחלקה, בשפת UML, היא סוג של תרשים סטטי המתאר את מבנה המערכת על ידי הצגת מחלקותיה, תכונותיהן והקשרים בין המחלקות.

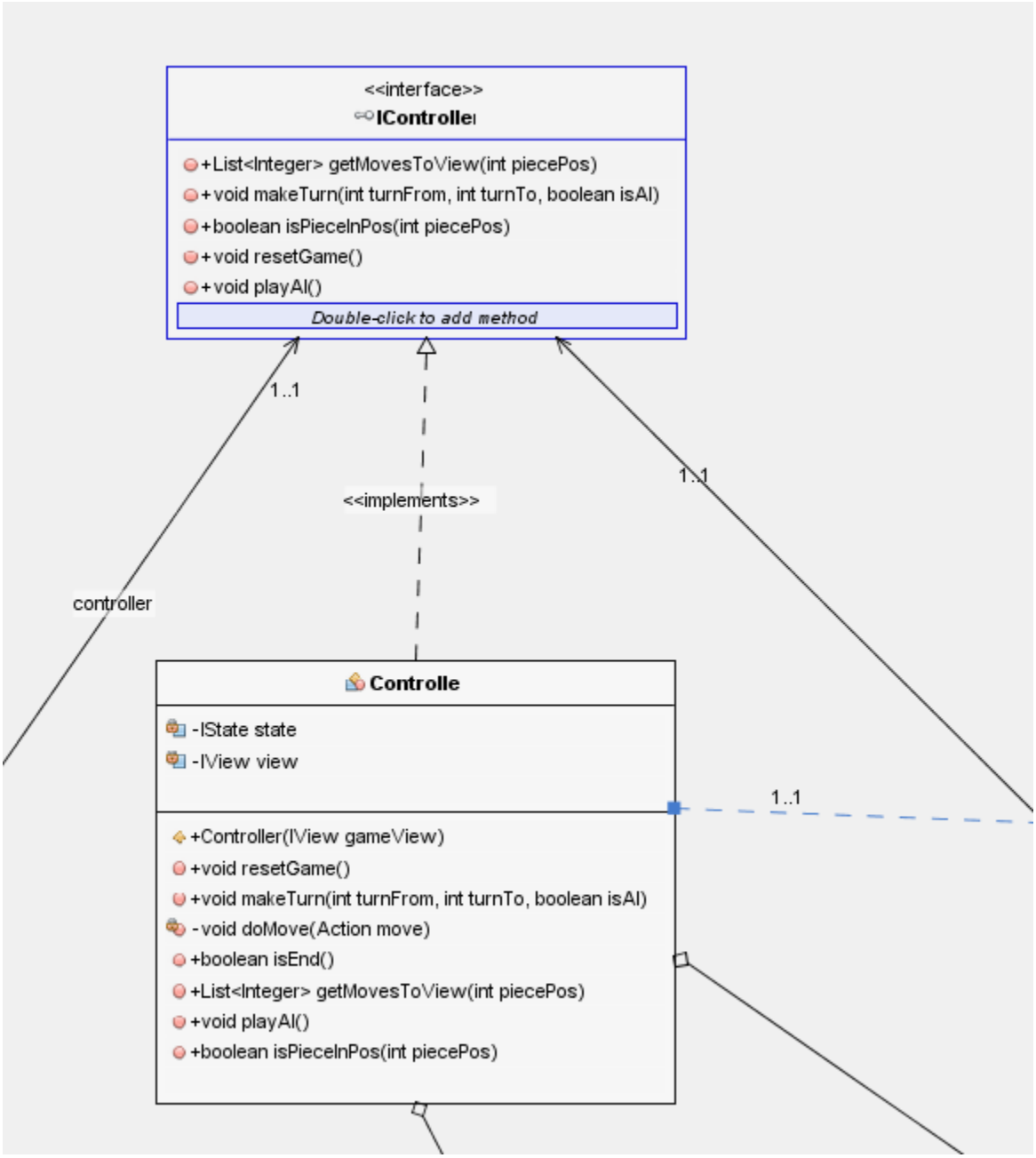
המחלקות בדיאגרמת המחלקה מייצגות גם את האובייקטים העיקריים וגם את היחסים בין האפליקציה ובין האובייקטים אשר יומרו לקוד מקור.

כתוצאה מכך שהפרויקט הוא גדול ורחב, אציג באופן נפרד את דיאגרמת מחלקה עבור כל חבילה מחלקות.

GameView Package: חבילת התצוגה

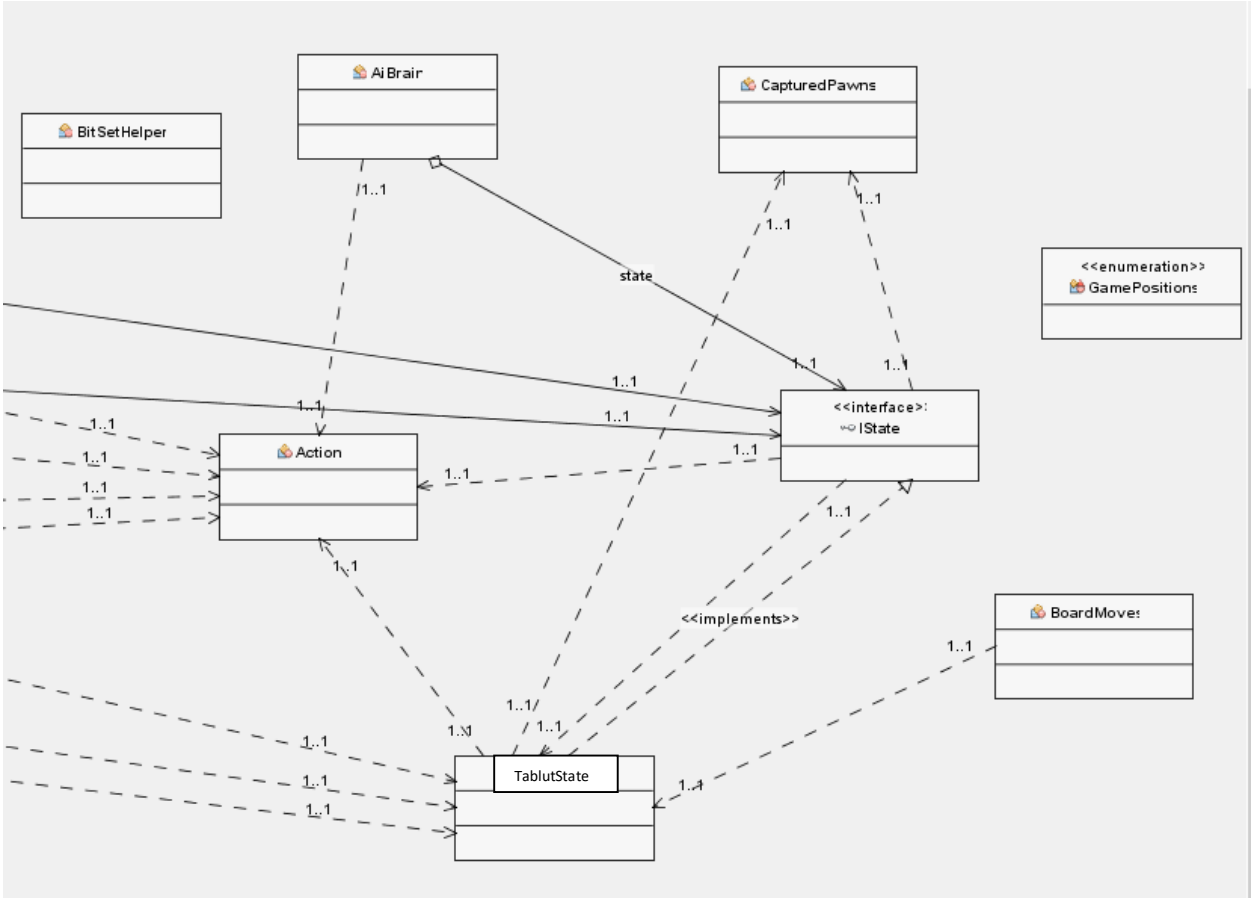


Controller Package: חבילת הבקר

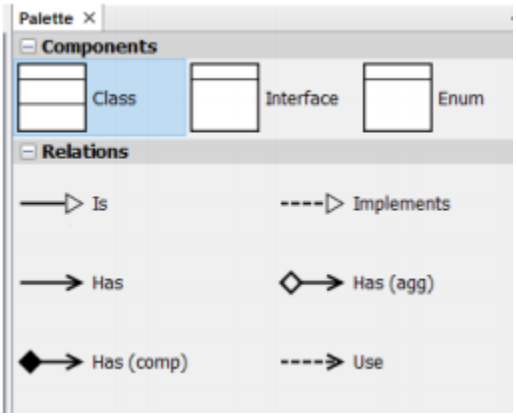


LogicModel Package: חבילת הלוגיקה

כתוצאה ממס' רב של פעולות ותכונות התקשיתי לקבץ את כולם בתמונה אחת לכן אציג את הפעולות והתכונות של מחלקות אלו בפירוט בחלק של תיאור המחלקות הראשיות בפרויקט.

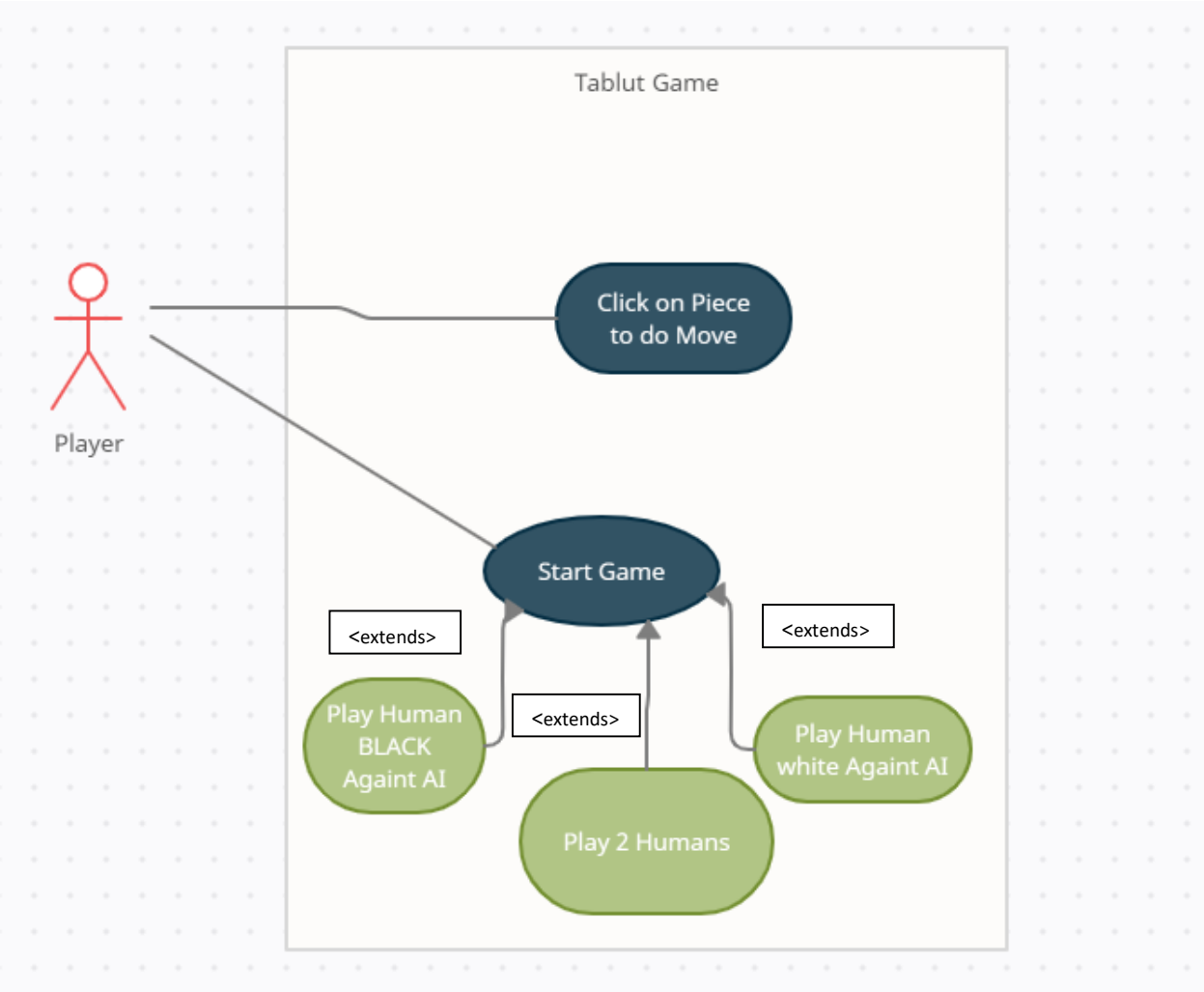


מקרא סימנים:



תרשים use-case

תרשים Case Use מתאר את הקשר שקיים בין (Actors שחקנים) שפועלים במערכת ל Cases Use-השונים. כל Use Caseמתאר סידרה של פעולות אשר מייצרות ערך עבור הלקוח שמשתמש במערכת.



Top-down levels diagram

תרשים down-top מתאר את חלוקת המערכת את תת המערכות שמרכיבות את המערכת והרכיבים של תת המערכות.

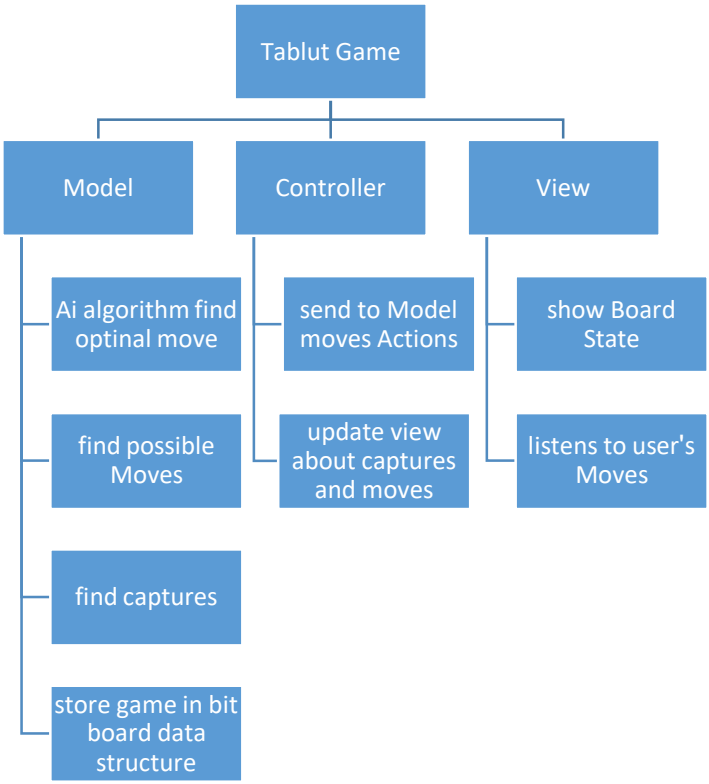
תבנית Model-View-Controller (בקיצור MVC):

היא תבנית עיצוב בהנדסת תוכנה המשמשת להפשטת יישום כלשהו. התבנית מתארת טכניקה לחלוקת היישום לשלושה חלקים, מודל, תצוגה ובקר, המחוברים ביניהם בצימוד רפוי מונחה אירועים. בדרך זו, התלות ההדדית בין ממשק המשתמש לשאר חלקי התוכנה פוחתת, ואת החלקים השונים ניתן לפתח באופן בלתי-תלוי. בנוסף, קל יותר לתחזק את התוכנה וכן לעשות שימוש חוזר בחלקי היישום שהופרדו.

מודל: כולל את שכבת הלוגיקה של התוכנה. כולל שימוש במבני נתונים ואלגוריתמים מורכבים.

בקר: תפקידו לעבד ולהגיב לאירועים המתרחשים בתצוגה, לרוב, כתגובה לפעולה של המשתמש. בעיבוד האירועים, הבקר עשוי לשנות את המידע במודל, באמצעות שפעול שירותים המוגדרים בו.

תצוגה: תפקידה להמיר את נתוני המודל לייצוג המאפשר למשתמש לבצע פעולת גומלין כלשהי. לרוב מדובר על המרה לממשק משתמש כלשהו.



תיאור סביבת העבודה ושפות תכנות

שפת התכנות בה בחרתי לכתוב את הפרויקט היא Java.

בתחילת העבודה היה לי קונפליקט האם לבחור בה או ב#c.

השיקולים העיקריים שלי באיזה שפה אני עדיין לא בקיא כמו שהייתי רוצה להיות.

מכיוון שבמהלך התיכון נהגתי לכתוב ב#c, החלטתי לקחת על עצמי אתגר לכתוב בשפה שיש לי בה יחסית חוסר ניסיון על מנת להשתפר.

כמו כן, כאשר בחרתי לעבוד עם מבנה הנתונים לוח הביטים bit Board, גיליתי את המבנה Bit Set אשר מוטמע בשפה Java ושמחתי לגלות את הנוחות ואת הממשק הרחב והגמיש אשר עזר לי רבות במהלך כתיבת הפרויקט.

גרסת השפה:

```
C:\Users\Ofir>java  -version
java version "1.8.0_281"
Java(TM) SE Runtime Environment (build 1.8.0_281-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.281-b09, mixed mode)
```

סביבת עבודה: netbeans 8.2

תיאור ממשקים

על מנת לבנות את הממשק הגרפי של הפרויקט, השתמשתי בממשק של java אשר נקרא SWING

Swing היא ממשק משתמש גרפי (GUI) של ערכת כלי לפיתוח בjava.

Swing פותחה על מנת לספק סט מתוחכם יותר של רכיבי GUI מאשר ה Abstract Window Toolkit.

Swing מספקת עיצוב ומראה אשר מחקים את אלו של מספר פלטפורמות, ובנוסף תומכת במכניזם המאפשר לשנות את העיצוב של GUI בזמן ריצה. יכולת זו מאפשרת לעיצוב ולמראה של האפליקציות להיראות לא קשורים לפלטפורמת היסוד.

תוצאות פיתוח הממשק הגרפי

הממשק הגרפי שלי מוצג במרכזו בהצגת לוח המשחק Tablut.

במהלך תור, כאשר שחקן לוחץ על כלי שלו מוצגים לו כל התאים אשר הוא יכול לנוע אליהם עם כלי המשחק.

כמו כן, כאשר המשחק נגמר, מוצגת הודעה מתאימה ומוצגת אפשרות להתחיל את המשחק מחדש.

בכל שלב של המשחק אפשר לבחור לחדש את המשחק.

או לשנות את Mode המשחק ע"י בחירת מצב המשחק הרצוי בסרגל מצד שמאל למעלה.

חוויה אישית בשימוש בממשק

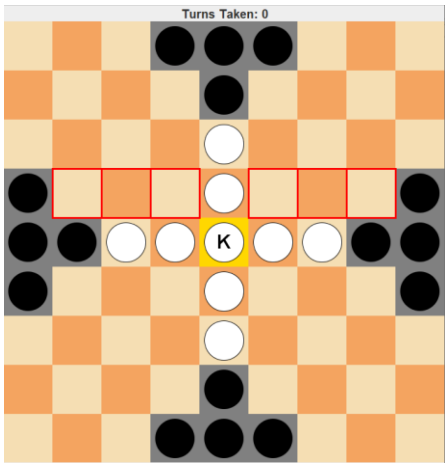
במהלך כתיבת הפרויקט נחשפתי בפעם הראשונה לשימוש בממשק זה. עד כה, הממשק הגרפים שפיתחתי היו לשימוש web ולכן נכתבו בJava Script.

בתחילת הכתיבה היה לי מעט מאתגר להבין איך להתנהל עם הממשק ולקח לי זמן להגיע לתוצאה הסופית שאציג בפרויקט.

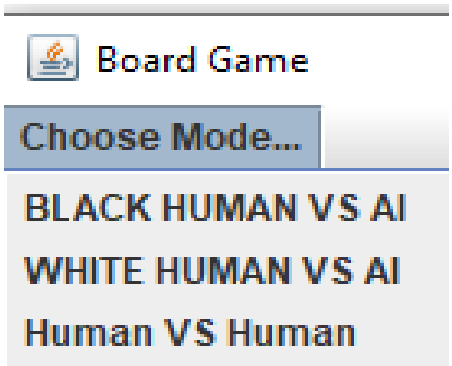
לדעתי, הדרך הטובה ביותר ללמוד שימוש בממשק הוא דרך מדריכי יוטיוב אשר מסבירים שלב שלב איך ללמוד ולהכיר את השפה ולאחר מכן הלימוד יותר מניב ומהנה.

תמונות של הממשק הגרפי

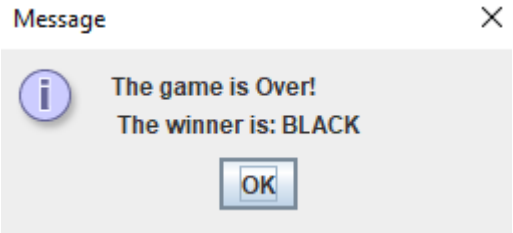
לוח המשחק והצגת המהלכים החוקיים



רשימה מצבי המשחק



הודעת ניצחון:



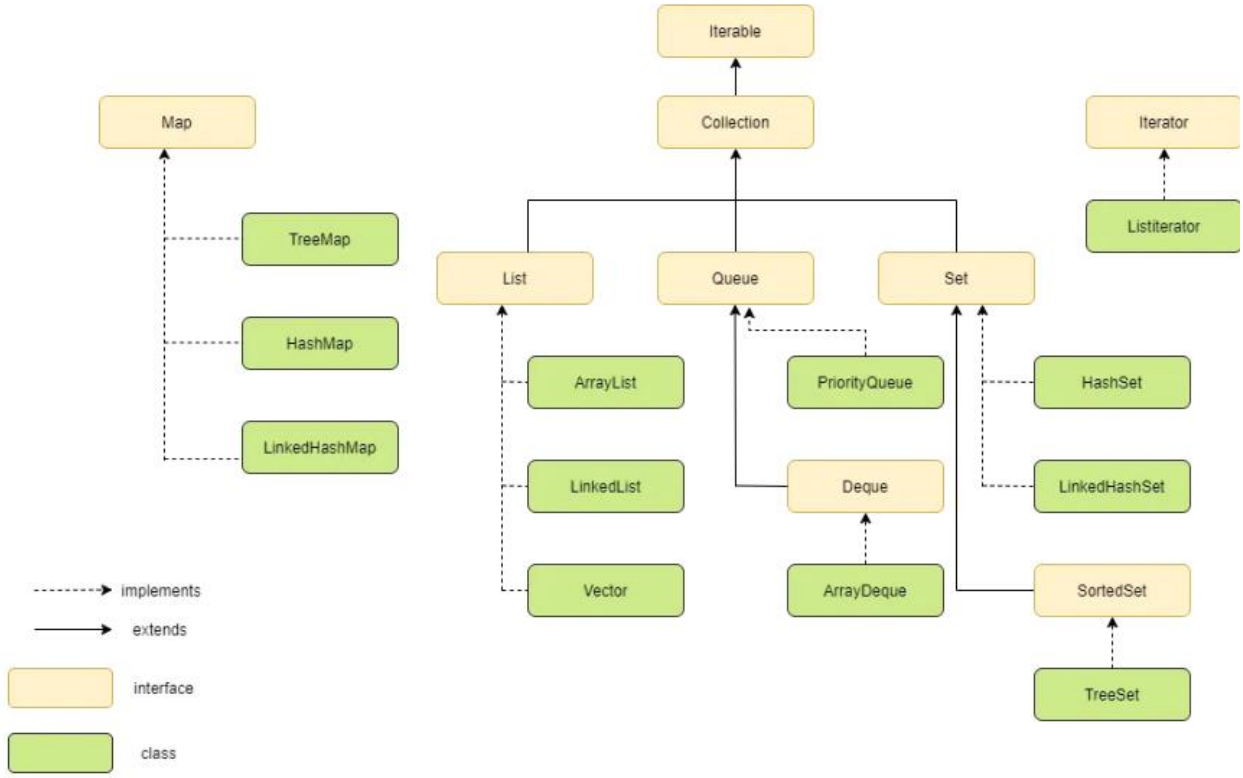
ממשק Java Collection

במהלך כתיבת הפרויקט, השתמשתי בממשק Interface Collection API.

בג'אווה, אוסף הוא ישות אחת המייצגת קבוצת אובייקטים ומסגרת היא ארכיטקטורה מוגדרת המכילה ממשקים ומחלקות. לפיכך, מסגרת Java Collection היא ארכיטקטורה המגדירה מערך ממשקים, מחלקות ואלגוריתמים. מסגרת מאפשרת לבצע פעולות כמו חיפוש, הכנסה, מיון, מחיקה, מניפולציה וכו'

בפרויקט שלי שימוש במבני הנתונים: ArrayList,List,HashMap,BitSet

היררכיית מסגרת אוספי Java



יתרונות אוספי Java

1. API עקבי - המפתחים יכולים להשתמש בממשק ה API -הקיים ואינם דורשים מאמץ נוסף לפיתוחו.
2. מפחית מאמץ תכנות - זה מפחית את המאמץ לתכנן את מבני הנתונים מכיוון שהמסגרת כבר מספקת אותו.
3. מגביר את איכות הקוד ופיתוחו - הוא מספק ביצועים גבוהים
4. מספק לשימוש חוזר

אלגוריתם ראשי

האלגוריתם הראשי בפרויקט שלי הוא אלגוריתם הנקרא Nega max אשר ממומש במחלקה AiBrain.

מטרת האלגוריתם הוא למצוא עבור השחקן הממוחשב את המהלך הטוב ביותר עבורו בכל תור במהלך המשחק.

לב האלגוריתם הוא לבנות עץ משחק בכל תור של השחקן הממוחשב ובעזרתו להפיק את המסקנה איזה מהלך הוא האופטימלי על מנת להגיע לניצחון.

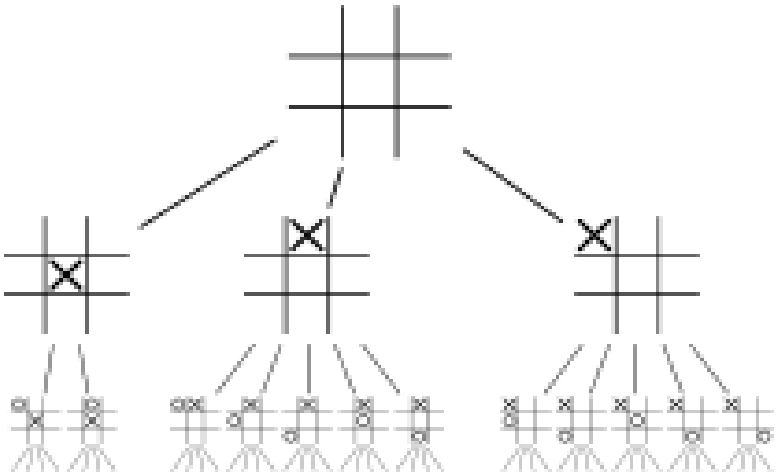
עץ משחק הוא גרף מכוון שצמתיו הם מצבים של משחק וקשתותיו הם מהלכים במשחק. עץ המשחק השלם של משחק הוא עץ ששורשו מהווה את המצב ההתחלתי של המשחק, ומכיל את כל המהלכים האפשריים.

העלים בעץ שנוצר הם מצבים סטטיים שנגיע אליהם לאחר רצף של מהלכים. ניתן ציון לכל מצב סטטי שכזה, שישקף כמה המצב טוב מבחינתנו.

מתן הציון מתבצע ע"י פונקציה שנקראת פונקציית הערכה (evaluation function).

דוגמא לעץ משחק פרוש

למשחק "איקס – עיגול "



אלגוריתם הnega Max:

האלגוריתם מייצר עץ שפורס לסירוגין את כל האפשרויות של שחקן א' לפעולה max ואת כל התגובות של שחקן בmin וכך הלאה. כל ענף בעץ מתאים לפעולה כלשהי של שחקן. אם נסמן את השורש כגובה 0 ,אז כל הענפים שיוצאים מקדקודים בגובה זוגי 2*n הם פעולות של הסוכן שלנו, וכל הענפים שיוצאים מרמות אי זוגיות 2*n+1 מייצגות פעולות של השחקן היריב. מכיוון שתור במשחק משמעו פעולה של שחקן א' ושחקן ב', אזי בפועל כל שתי שכבות בעץ מייצגות תור אחד, כלומר n1+ הוא למעשה מספר התור. ככל ש n-גדול יותר, כך ניתן לומר שאנו מסתכלים יותר תורות קדימה.

הקדקודים בעץ מייצגים מצב עניינים במשחק. מצב עניינים זה נוצר באופן הבא: נניח את המצב המתואר בשורש, כעת נלך לאורך הענף שמוביל מהשורש ועד לאותו קדקוד. כל שני צעדים בדרך מייצגים פעולה של שחקן א' ופעולה של שחקן ב', כלומר תור אחד במשחק, ולכן ניתן "להעביר" תור ולהגיע למצב חדש.

החיפוש נעשה באופן הבא: נסרוק את העץ מהעלים מעלה, כאשר לכל עלה וניתן ציון ע"י הGrader.

השחקן ברמה שמעליהם הוא היריב MIN ומטרתו לפגוע בסוכן שלנו, לכן היריב ינקוט בפעולה שתוביל לעולם עם הציון הנמוך ביותר מבין העלים. הוא מפעפע לקדקוד שמעליו את הציון. ברמה שמעליו נמצא הסוכן שלנו. הסוכן רוצה לפעול בצורה שתמקסם את הרווח שלו, ולכן יבחר מבין כל האופציות שהציב לו יריבו את הפעולה שתוביל אותו לעולם עם הציון הגבוה ביותר. הוא מפעפע את הציון הזה מעלה, וכך זה ממשיך עד השורש. בסופו של דבר הפעולה שהסוכן יבחר לבצע בתור הנוכחי היא הפעולה שיוצאת מהשורש ומובילה לעולם בעל הציון הגבוה ביותר, כפי שפעפע מלמטה.

<u>הסיבות לבחירות האלגוריתם:</u>

1. המשחק הוא <u>משחק סכום אפס</u> ,היינו, סכום הרווחים יהיה תמיד 0.
2. המשחק מהווה <u>מערכת דטרמיניסטית</u> - המשחק אינו תלוי בגורם אקראי או בלתי-קבוע, ובהינתן <u>אסטרטגיה</u> מסוימת על ידי שני השחקנים, תוצאת המשחק תמיד תהיה זהה.
3. המשחק הוא משחק <u>מידע מלא</u> - מטרת המשחק והאפשרויות החוקיות בו ברורות לכולם.
4. המשחק מתקיים בסביבה של <u>מידע מושלם</u> - כל פעולות העבר חשופות לעיני שני הצדדים, ואין מהלכים נסתרים או בלתי-חשופים.

עומק העץ:

בעיה קשה בבניית עץ מינימקס היא הזיכרון הרב שהוא צורך. מספר הקדקודים שיש לפתח עולה בטור הנדסי ככל שנעמיק את החיפוש. סוף המשחק נמצא הרחק מעבר לאופק החישוב ,ולא ניתן לנקד כל עמדה בצורה מדויקת.

לכן קובעים פרמטר אשר נקרא "עומק העץ" אשר זה הוא מספר המקסימלי של רמות בעץ אשר אלגוריתם "יורד" אליו בחיפוש אחר מהלך אופטימלי. כלומר המחשב "יודע לחשוב" כמות מסוים של צעדים קדימה. מכיוון שיש משחקים ארוכים ומורכבים אשר על מנת לבנות עץ משחק שלם שלהם יידרש זמן רב מאוד וקריאה רקורסיבית עצומה על מנת להגיע לשורשי העץ – סיום המשחק.

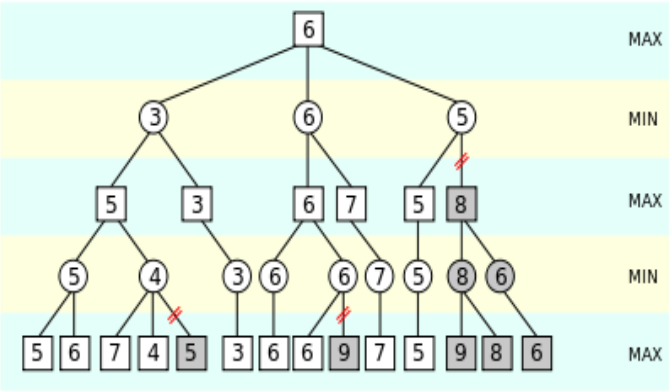
גיזום אלפא ביתא:

קיימת שיטת "גיזום" המבטלת בנייה של ענפים שברור לנו עוד בשלב מוקדם כי הם לא מועילים לחיפוש שלנו. ניתן כך להקטין מספר הצמתים בעץ לשורש המספר שהיה מתקבל ללא הגיזום (בממוצע). שיטה זו נקראת גיזום אלפא-ביתא.

עבור משחקים כמו טאבלוט וגו, עץ המשחק המתאר אותם הוא ענף ביותר, ונמצא הרבה מעבר לאופק החישוב המעשי של חומרה קיימת כלשהי.

כאן מגיע התמריץ המעשי לשימוש בגיזום אלפא-ביתא, ובשיטות ייעול נוספות. אנו מעוניינים "למצות" את שלל ההילוכים האפשריים על עץ משחק, מבלי להצטרך לבצע כל הילוך והילוך בפועל. הרעיון בשימוש בחיפוש אלפא-ביתא, הוא לבצע גיזום (חריף ככל הניתן) של ענפי העץ, מבלי לפגוע באיכות התוצאה המוחזרת.

האלגוריתם מחזיק שני משתני עזר, אלפא (α) וביתא ,(β) המייצגים את התוצאה המינימלית המובטחת לשחקן ה'מקס', ואת התוצאה המקסימלית המובטחת לשחקן ה'מין', בהתאמה.
בתחילה, נקבעים משתנים אלו להיות בעלי ערכי קיצון שרירותיים - אלפא למינוס אינסוף, וביתא לפלוס אינסוף. ערכים תחיליים אלו, מייצגים את המצב הגרוע ביותר מבחינתו של כל שחקן - לשחקן הממקסם מובטחת תוצאה גרועה ביותר (מינוס אינסוף), ולשחקן הממזער גם כן תוצאה גרועה ביותר לפי השקפתו (פלוס אינסוף). עם התקדמות החיפוש, הולך ומצטמצם המרחק בין אלפא לביתא. כאשר ביתא מקבלת ערך נמוך מאלפא, פירושו של דבר כי העמדה הנוכחית אינה תוצאה של משחק מושלם על ידי שני השחקנים ואין צורך להעמיק חקר בתת-העץ הנוכחי.



פסאדו קוד של האלגוריתם:

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := −∞
    for each child of node do
        value := max(value, −negamax(child, depth
− 1, −color))
    return value
```

```
(* Initial call for Player A's root node *)
negamax(rootNode, depth, 1)
```

```
(* Initial call for Player B's root node *)
negamax(rootNode, depth, −1)
```

NegaScout - move ordering

בשלב שלאחר פיתוח הקוד של הבינה המלאכותית עם גיזום אלפא ביתא הבחנתי

שכאשר אני מחפש בעץ המשחק בעומק 4. זמן בחירת המהלך הטוב ביותר עבור המחשב לוקח מעט מאוד זמן (פחות מחצי שנייה).

לכן החלטתי לנסות לראות איך העלאת העומק ל5 ישפיע על זמן זה.

לאחר הניסוי הבחנתי שכאשר העומק הוא 5 זמן בחירת התור יכול לעלות לפעמים ל-7-5

שניות. הסיבה לכך היא שמס' המהלכים החוקיים של שחקן יכול להגיע למעל 90 מהלכים שונים.

גדילה במספר מהלכי המשחק המחושבים עולה בצורה מעריכית.

לכן חקרתי באינטרנט דרכים שונות לעשות אופטימיזציה לגיזום אלפא ביתא כלומר לגרום לכך שיהיו יותר גיזומי עץ באופן כללי. דבר אשר יגרום לחקירת עץ משחק קטן יותר וכתוצאה מכך לירידה בזמן בחירת התור האופטימלי. הטכניקה אשר בחרתי היא **סידור מהלכים בגיזום אלפא ביתא.**

הרעיון של טכניקה זאת היא חישוב פונקציית הערכה משנית אשר תחושב בזמן חישוב O(1).

כמובן שאם היינו משתמשים בפונקציות הערכה הראשיות של הפרויקט חישוב כל ציוני המצבים יגדיל בצורה משמעותית את זמן בחירת התור וזאת הסיבה שהפונקציה המשנית היא ביעילות זמן ריצה קטנה.

לאחר חישוב זה, יש למיין את מבנה הנתונים אשר מחזיק את המהלכים האפשריים עבור השחקן במצב הלוח ולמיין אותו בסדר יורד ביחס לציון הערכה המשני שחושב.

העיקרון הוא פשוט מאוד, מתבסס על סידור מהלכים יעיל (מהלכים מועילים קודם). על-מנת לגלות מראש סידור כזה, הוא משקיע זמן לא מבוטל בסידור המהלכים. בדרך כלל הליך זה יבוצע בשיטה של "השלכת חכה". האלגוריתם מבצע חיפוש א"ב רגיל, לעומק רדוד וממיין את רשימת המהלכים הטובים לפי ציון בעומק זה.

ייתכן כי האינדיקציות הראשוניות אכן היו נכונות, ומהלכים שנראו מבטיחים בתחילה אכן התבררו ככאלה. במקרה כזה יבוצע גיזום חריף ביותר בעץ, והפסדנו בשכרנו - הזמן שהושקע בסידור המהלכים הוא זניח לעומת הזמן שנחסך על ידי הגיזום החריף, וסך הכל חסכנו זמן.

לאחר מימוש הטכניקה, הרצתי משחק של בינה מלאכותית נגד בינה מלאכותית ובדקתי את משך המשחק עם סידור מהלכים ובלי בכדי לבחון את תוצאות האופטימיזציה (כמובן המשחקים זהים)

הדהמתי לגלות פער של 17 שניות בין זמני סיום המשחקים כאשר במשחק 20 מהלכים בלבד!

מיון רשימת המהלכים:

פונקציות הערכה **המשנית** עבור שחקן שחור:

1. מהלכים אשר משבצת היעד שלהם היא ליד המלך הלבן – ציון 2.
קירבת חיילים שחורים למלך הלבן הוא גורם משפיע מאוד על מהלך המשחק וכנראה ישפיע על ציוני פונקציות המרכזיות בעומק העץ.

2. מהלכים אשר משבצת המוצא שלהם היא לא במחנה ההתחלתי – ציון 1.
חיילים אשר נמצאים במשבצות "השדה" מחוץ למחנות שלהם הם פחות מוגנים אך הם חופשיים לנוע לעבר המלך, להרוג חיילים אחרים ועוד.
לכן הסקתי שמהלכים אלו הם הסתברותית יהיו טובים לשחקן השחור.

3. שאר המהלכים – ציון 0.

שימוש ב comparator על מנת למיין את המהלכים של השחקנים השחורים:

```
if (this.boardGame.getCurrentPlayer() == Player.BLACK) {

    int kingPos = this.boardGame.getKing().nextSetBit(0);

    moves.sort(new Comparator<Action>() {
        // sort in descending order
        @Override
        public int compare(Action m1, Action m2) {
            return m2.actionBlackValue(kingPos) - m1.actionBlackValue(kingPos);
        }

    });
```

שימוש בטכניקה של JAVA על מנת למיין את רשימת המהלכים ע"י שימוש בcomparator אשר נלמדה בכיתה במהלך השנה.

פונקציות הערכה המשנית עבור שחקן לבן:

1. המהלכים של המלך יחושבו לפני כלל המהלכים האחרים של השחקן הלבן.
מצבים אלו הם המביאים להתקדמות השחקן הלבן לניצחון.

- **אין סיבה למיין** את רשימת המהלכים כאשר השחקן הלבן כתוצאה ממימוש הפונקצייה "קבל כל המהלכים החוקיים" בדרך כזאת שמהלכי המלך תמיד יהיו בתחילת הרשימה.

פירוט פונקציות ראשיות באלגוריתם הראשי:

1. NegaMax

פרמטרים וכותרת הפעולה:

```
/**
 * the function use negaMax algorithm with alpha beta pruning
 * to choose the optimal move for the current player in this.boardGame
 * @param depth depth of the search in the Game tree
 * @param alpha The best highest-value choice we have found so far
 * @param beta The lowest-value choice we have found so far
 * @return the best Action for the current player founded
 */
```

מטרת הפונקציה:

שימוש באלגוריתם המוסבר לעיל על מנת למצוא את המהלך הטוב ביותר עבור שחקן הממוחשב, ע"י חקירת עץ המשחק וניסיון למקסם את "ציון" הלוח במספר התורות הבאים.

יעילות החיפוש היא: מספר המהלכים החוקיים הממוצע בחזקת עומק החיפוש

שם מחלקה: AiBrain אשר אחראית על הבינה המלאכותית בפרויקט

פסואדו קוד:

1. אם עלה או עומק אפס ?
   1.1 החזר מהלך עם ציון שמחושב בפונקציית היוריסטיקה
2. מצא כל המהלכים החוקיים
3. מיין אותם לפי עיקרון move Ordering (הוסבר לעיל)
4. מהלך הטוב ביותר -> ציון נמוך ביותר
5. עבור כל מהלך:
   5.1 בצע מהלך
   5.2 אם ציון המהלך המוחזר מ: (קריאה רקורסיבית לפעולה עם עומק פחות 1) * 1- גדול מהמהלך הטוב ביותר:
      5.2.1 שנה מהלך הטוב ביותר למהלך הנוכחי
   5.3 מחק מהלך מהלוח
   5.4 האם אפשר לגזום חלק מהעץ?
      5.4.1 גזום את העץ
6. החזר מהלך הטוב ביותר

הקריאה לפונקציה תראה כך:

```
Action optimalAction = negaMax(DEPTH_SEARCH, Integer.MIN_VALUE, Integer.MAX_VALUE);
```

עומק הפונקציה הנבחר הוא 5.

לאחר אופטימיזציית גיזום העץ הזמן הממוצע לשחקן הממוחשב לבחור מהלך בעומק זה היא 0.7 שניות בלבד. לעומת הזמן הממוצע ללא גיזום שהוא יכול להגיע ל**15** שניות לתור.

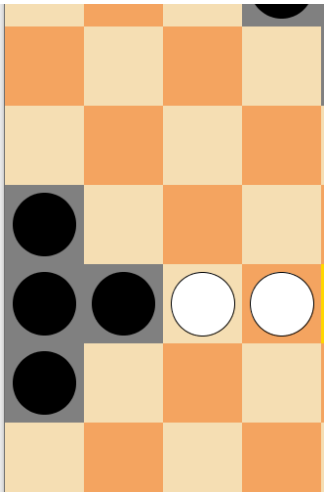
על מנת לנתח כמות זמן למהלך בחרתי במהלך בניית הפרויקט להדפיס עבור כל מהלך שנבחר את זמנו.

הפרמטר אשר משפיע על זמן החיפוש זה הוא:

- עומק החיפוש
- כמות המהלכים האפשריים לשחקן
- סיבוכיות פונקציית הערכה

דוגמא לפלט התוכנית במשחק נגד שחקן ממוחשב:

```
Time to Choose Move: 1.77 Seconds!
        Action{from=37, to=55, Grade: 24.859661400465196}
Time to Choose Move: 0.603 Seconds!
        Action{from=13, to=11, Grade: 24.82627193677363}
Time to Choose Move: 0.764 Seconds!
        Action{from=43, to=25, Grade: 25.065095016636068}
Time to Choose Move: 3.114 Seconds!
        Action{from=3, to=1, Grade: 25.52349802750443}
Time to Choose Move: 2.221 Seconds!
        Action{from=11, to=20, Grade: 23.798499130679232}
Time to Choose Move: 1.46 Seconds!
        Action{from=67, to=65, Grade: 26.298499130679232}
Time to Choose Move: 0.63 Seconds!
        Action{from=1, to=19, Grade: 27.15952231009369}
```



מימוש אסטרטגיית המשחק באלגוריתם הראשי:

אחת הסיבות העיקריות אשר גרמו לי לבחור את נושא הפרויקט היא העבודה שבניגוד למרבית ממשחקי הלוח הנפוצים, משחק ה Tablut הוא אסימטרי.

כלומר, לשני השחקנים יש דרך שונה על מנת להגיע לניצחון.

כתוצאה מכך נאצלתי לכתוב שני פונקציות היוריסטיקה שונות עבור כל שחקן.

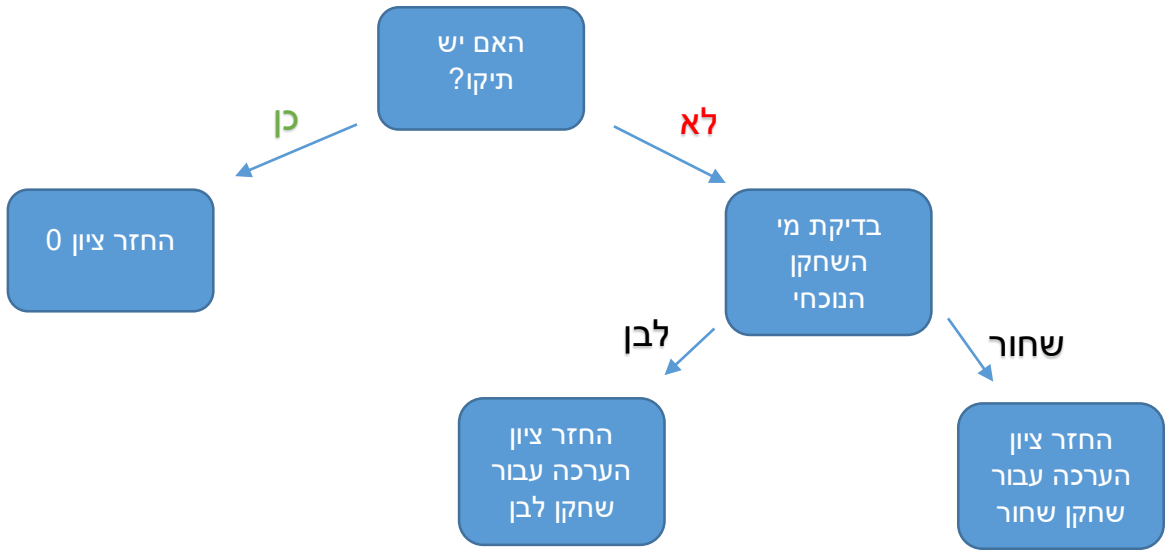
הפונקציות נכתבו במחלקה TablutState אשר מייצגת את לוח המשחק ומצבו בכל רגע נתון.

על מנת לכתוב את הפונקציה בצורה הטובה ביותר, חקרתי לעומק אסטרטגיות אנושיות של המשחק באינטרנט בעיקר ע"י קריאה של מאמרים באתר שוודי שמנתח מצבים שונים ותכסיסים של המשחק.

לאחר ניסוי ותהייה רבים צמצמתי הרבה אפשרויות והגעתי למס' הפרמטרים החשובים ביותר שיש לניתוח מצב לוח של המשחק ובחרתי לתת "משקל" שונה לכל פרמטר.

כאשר נרצה לבדוק מה "ציון" הלוח כל שעלינו יהיה לעשות יהיה לקרוא לפונקציה המתאימה על אובייקט

TablutState קבל ציון היוריסטיקה:



אשר אחראית לבדוק של איזה שחקן התור הנוכחי ולהפעיל את פונקציית היוריסטיקה המתאימה

לשחקן whiteHuristic or blackHuristic . כמו כן, אם המשחק במצב של תיקו, הפונקציה תחזיר ציון אפס.

הגורמים המשפיעים על ציון הלוח – פונקציית הערכה

כמות הכלים הלבנים המגנים במשחק

במצב ההתחלתי של המשחק ישנם 8 כלים לבנים אשר מטרתם להגן על המלך ועזור לו להגיע לאחת ממשבצות הבריחה.
לכלים אלו יש תפקיד משמעותי על מנת לנצח את המשחק בתור שחקן לבן:

- ללכוד כלים יריבים על מנת להפחית את הכלים המאיימים על המלך.
- להקריב את עצמם על מנת להגן על המהלך ולעיתים להילכד במקומו.

כתוצאה מחשיבותם ניתן לקבוע שכמות כלים אלו אשר נשארים במשחק משפיעים רבות על רמת ההתקדמות במשחק עבור שני שחקני המשחק – המגנים והתוקפים.
המגנים רוצים לשמור על מספר הכלים הלבנים החיים גבוה במהלך המשחק לעומת השחורים שרוצים למזער אותו.

קוד: בסך הכול לבדוק מה כמות הסיבות הדולקות בלוח הסיביות המייצג את הכלים הלבנים whitesPawns.cardinality() פעולה אשר בודקת כמות סיביות בBitSet.
יעילות הפונקצייה היא O(1)

כמות הכלים השחורים התוקפים במשחק

במצב ההתחלתי של המשחק ישנם 16 כלים שחורים אשר מטרתם ללכוד את המלך ולמנוע ממנו להגיע לאחת ממשבצות הבריחה.
לכלים אלו יש תפקיד משמעותי על מנת לנצח את המשחק בתור שחקן שחור:

- ללכוד כלים יריבים על מנת להפחית את הכלים המגנים על המלך.
- לסגור את דרכי היציאה של המלך "ולסגור עליו" ובכך ללכוד אותו.
-

כתוצאה מחשיבותם ניתן לקבוע שכמות כלים אלו אשר נשארים במשחק משפיעים רבות על רמת ההתקדמות במשחק עבור שני שחקני המשחק – המגנים והתוקפים.
התוקפים רוצים לשמור על מספר הכלים אלו החיים גבוה במהלך המשחק לעומת הלבנים שרוצים למזער אותו.

קוד: בסך הכול לבדוק מה כמות הסיבות הדולקות בלוח הסיביות המייצג את הכלים השחורים blackPawns.cardinality() פעולה אשר בודקת כמות סיביות בBitSet.
יעילות הפונקצייה היא O(1)

מיקום הכלים במקומות אסטרטגיים בלוח

כמו במשחקים רבים, בלוח המשחק יש מקומות אסטרטגיים אשר בהם יש כדאיות להציב את כלי המשחק על מנת להגיע לניצחון.
לאחר מחקר באינטרנט, בחרתי לתת לכל משבצת בלוח "משקל" כלומר כמה טוב לשחקן להימצא בתא זה. המשקלים נעים בין 0-5. כך שהמשבצת האופטימלית לשחקן היא 5 והכי פחות טובה/לא ניתנת להגעה היא 0. כל תא במערכי המשקלים מכיל את המשקל עבור שחקן שימצא בתא בלוח המשחק המתאים לו.
הפרמטר המכריע הוא חישוב הפרש סכום המשקלים של כלי המשחק עבור כל שחקן.
כלומר כל שחקן רוצה למקסם את סכום המשקלים שלו ובאותו הזמן לשמור על סכום השני נמוך.

שיקולי מתן המשקלים:

עבור שחקן מגן:

- תאים אשר חוסמים את הגעת המלך לפינות כמובן ינוקדו ב1. כמו כן הם הנגישים ביותר ע"י התוקפים ללכידתם.
- תאי משבצות הבריחה הם בעלי עדיפות פחותה לכלים רגילים לבנים.
- התאים אשר חוסמים את תנועת התוקפים ממחנותיהם הם בעלי משקל 4 והם בעדיפות גבוהה על מנת לנטרל את תנועתם של התוקפים.
- משבצות המוצא של הלבנים הם בעלי עדיפות 3, כי הם ממוקמים קרוב לטירה. במרבית המשחק המלך קרוב לאזור זה. כמו כן, שמירה על **מרכז הלוח בעליונות לבנה** היא הכרחית לדעתי לשליטה במשחק.

```
public static final int[] whiteWeights = {
        2, 2, 2, 0, 0, 0, 2, 2, 2,
        2, 1, 1, 4, 0, 4, 1, 1, 2,
        2, 1, 2, 2, 3, 2, 2, 1, 2,
        0, 4, 2, 1, 3, 1, 2, 4, 0,
        0, 0, 3, 3, 0, 3, 3, 0, 0,
        0, 4, 2, 1, 3, 1, 2, 4, 0,
        2, 1, 2, 2, 3, 2, 2, 1, 2,
        2, 1, 1, 4, 0, 4, 1, 1, 2,
        2, 2, 2, 0, 0, 0, 2, 2, 2
};
```

עבור שחקן תוקף:

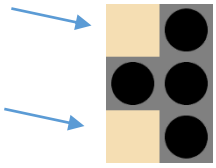
המשקלים:

```
public static final int[] blackWeights = {
        1, 3, 2, 3, 3, 3, 2, 3, 1,
        3, 2, 5, 1, 3, 1, 5, 2, 3,
        2, 5, 1, 5, 2, 5, 1, 5, 2,
        3, 1, 5, 1, 2, 1, 5, 1, 3,
        3, 3, 2, 2, 0, 2, 2, 3, 3,
        3, 1, 5, 1, 2, 1, 5, 1, 3,
        2, 5, 1, 5, 2, 5, 1, 5, 2,
        3, 2, 5, 1, 3, 1, 5, 2, 3,
        1, 3, 2, 3, 3, 3, 2, 3, 1
};
```

- עבור מיקום שחקן תוקף במשבצת המחנה, בחרתי לנקד 3 נקודות כתוצאה מאותם תוקפים מוגנים במשבצות אלו. לא מומלץ לגרום למספר תוקפים רב לעזוב אותם ללא עורף מכיוון שהם יהיו חשופים ללכידות הלבנים אשר שולטים במרכז הלוח.

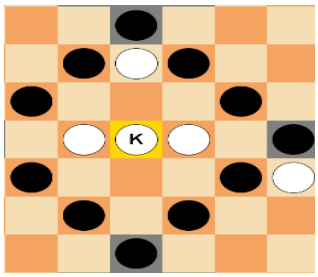
- ניקוד 2 יינתן למשבצות אשר חוסמות במעט את המלך אך הן "קו ההגנה" הפחות חזק מאשר משבצות הרומבוס שיפורטו בהמשך. לכן הם מבחינה אסטרטגית מתוגמלות פחות.

- ניקוד 1 הנמוך ביותר ניתן במשבצות "מסוכנות" מפני שהן חשופות ללכידה, כתוצאה מהחוק שניתן להילכד ע"י כלי יריב אחד בלבד ליד מחנות. ובעיקר בגלל שהן לא מקדמות את השחורים ללכידת המלך.



- עבור שחקן תוקף, יש משבצות ששמן "רומבוס" אשר הם יוצרות מעין חסם למלך ומקיפות אותו. למשבצות אלו יש עדיפות עליונה כלפי השחקן התוקף, על מנת לצמצם את דרכי המלך לכיוון אחת ממשבצות ה"בריחה". משבצות אלו מנוקדות בציון 5!

דוגמא לסידור לוח הכולל חסימת המלך ותפיסת טריטוריית משבצות הרומבוס



פונקציית מימוש האסטרטגיה: differenceSumWeights במחלקה TablutState

כותרת הפונקציה:

```
/**
 * The function calculate the Difference between The Sum Strategic Weights
 * of the white Pawns and the Black Pawns
 * @param whiteCount number of white Pawns Alive
 * @param blackCount number of black Pawns Alive
 * @return  the difference of sums weighted between 0 to 1
 */

private double differenceSumWeights(int whiteCount ,int blackCount) {
```

הפונקציה נעזרת בפעולת עזר:
מצא סכום משקלים sumWeights אשר מקבלת את סוג השחקן ומחשבת את סכום המשקלים עבור הכלים שלו. עוברת עבור כל סיבית דלוקה בלוח הסיביות של השחקן שהתקבל בפרמטר, ומוסיפה למשתנה הסכום את הערך שבמערך המשקלים המתאים במיקום הזהה.

שכותרה:

```
/**
 * Calculate the sum Of the weights of the places
 * that the Pawns of the player is on
 * @param player The player to Calculate his Pawns board Positions
 * @return The sum of all the weights
 */
private int sumWeights(Player player) {
```

להלן תיאור פסאודו קוד differenceSumWeights

1. סכום לבנים: מצא סכום משקלים (כלים לבנים)
2. סכום שחורים: מצא סכום משקלים (כלים שחורים)

על מנת לנרמל את התוצאות בין 0 ל-1

3. חלק את הסכום הלבנים בכמות הכלים הלבנים שעל הלוח כפול הציון הממוצע לכלי לבן
4. חלק את הסכום השחורים בכמות הכלים השחורים שעל הלוח כפול הציון הממוצע לכלי שחור
5. החזר סכום לבנים פחות סכום שחורים

<u>יעילות:</u> קריאה לפונקציית sumWeights פעמיים שיעילותה היא O(מספר המשבצות בלוח) לכן (מספר המשבצות בלוח) = 9*9 = N^2

כמות הכלים השחורים המחנות והטירה אשר "מאיימים" על המלך

פרמטר חשוב הוא כמות החיילים התוקפים אשר נמצאים במשבצות הסמוכות למלך.
כמו כן, נספור כמובן גם את המחנות סביבו מכיוון שלפי החוק ניתן ללכוד חייל בעזרת משבצת המחנה. פרמטר זה קיבל אצלי את השם: "הסכנה למלך", כמובן שהשחקן הלבן ירצה למזער את הסכנה למלך לעומת השחקן השחור.

כתוצאה מכך, שניצחון המשחק עבור השחור הוא לכידת המלך, כאשר מס' רב של חיילים שחורים מקיפים את המלך יגרור ללכידתו ואיום על חייו במשחק.

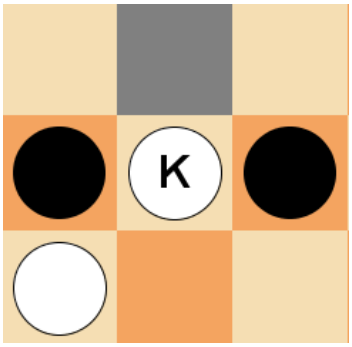
חשוב לציין, שישנם לכידות מלך מיוחדות אשר דורשות 2 3 או 4 חיילים אשר יקיפו את המלך על מנת ללכוד אותו ואנו נתייחס למס' חיילים אלו בחישוב פונקציית הערכה על מנת לנרמל את ניקוד הפרמטר בין 0 ל1.
כלומר, אם שני חיילים מאיימים על המלך כאשר הוא במצב אשר דרושים ארבעה בכדי ללכוד אותו יינתן ניקוד נמוך יותר בהשוואה למצב שבו דרושים בסך הכל 2 חיילים אשר יקיפו אותו.

כותרת הפונקציה:

```
/**
 * Find the protection level of the king
 * @param kingPos king bit position on the bit Board
 * @param blacks black bit board
 * @return count of black / obstacles cells around the king
 */
public static int dangerToKing(int kingPos, BitSet blacks) {
```

טענת יציאה: מניית כמות המחנות והחיילים השחורים ב4 המשבצות הסמוכות למלך.
יעילות: O(1)

להלן תיאור פסאודו קוד: בדיקת מצב הביטים בלוח הסיביות של הכלים השחורים והמחנות במיקומי התאים המקיפים את המלך.



דוגמא למצב שבו הפעולה תחזיר ערך 3 כתוצאה משני חיילים תוקפים ומחנה המקיף את המלך

כמות הדרכים למלך להגיע לאחת מפינות הלוח בתור אחד

פרמטר חשוב נוסף עבור הערכת מצב המשחק עבור כל אחד מהשחקנים הוא כמות אפשרויות שיש למלך על מנת להגיע לאחת מפינות המשחק.
פרמטר זה יהיה אחד מהמרכזיים בהערכת המשחק. מכיוון שכאשר יש למלך "דרך" לאחת הפינות זה אבן דרך המסמלת התקדמות משמעותית שלו לעבר הניצחון, לעיתים דרך זה יכולה להיחסם ע"י השחקנים המגנים אך במקרים אשר לא, המלך מוביל את דרכו לעבר משבצת הבריחה ומנצח.

<u>ניתוח מצב מיוחד:</u> כאשר לדרך יש יותר מדרך אחת לעבר משבצות "הבריחה", הוא מבטיח לעצמו מצב ניצחון מכיוון ששחקן השחור אינו יכול לחסום את 2 דרכי אלו בתור אחד בלבד.

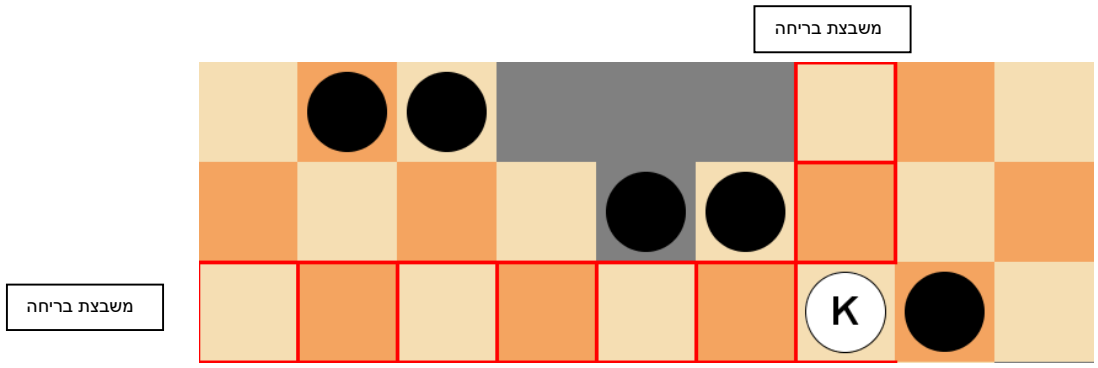
שם הפעולה הממשת את האסטרטגיה: freePathsToKing

להלן פסואדו קוד:
1. שמירה ברשימה את כל המהלכים החוקיים של המלך
2. בניית מסיכה בגודל לוח המשחק והדלקת הביטים אשר מסמלים את משבצות היעד של השחקן בכל מהלכיו האפשריים
3. ביצוע פעולה לוגית and בין לוח סיביות של משבצות הבריחה ולוח המסכה שנבנתה
4. החזרת מס' הביטים הדלוקים בלוח הסיביות שיצא כתוצאה מהפעולה "גם"

דרך הפעולה היא לבנות מסכה המסמלת את המיקומים אשר אליהם המלך יכול להגיע, ובצע את הפעולה הלוגית AND בין מסיכה זאת לבין מסיכה של משבצות הבריחה.
לאחר הפעולה הביטים הדלוקים יהיו תאי היעד של המלך אשר הם "גם" משבצות בריחה.

יעילות הפעולה:
היא O(N) כמספר המהלכים החוקיים האפשריים למלך

דוגמא למצב לוח אשר הפעולה תמנה 2 משבצות בריחה אשר המלך יכול לנוע אליהם בתור אחד.



כמות המגנים אשר צמודים למלך

גורם זה קרוי אצלי "הגנת המלך" כתוצאה ממטרתו: הכלים הלבנים במהלך המשחק צריכים להיות קרובים לכלי המלך על מנת למנוע מהכלים השחורים ללכוד אותו ולמנוע ממנו לנוע למשבצות הבריחה.
בחרתי בגורם זה מכיוון שההיגיון הוא פשוט: מספר רב של כלים לבנים מסביב למלך יעזרו לו לנצח את המשחק בבטחה ואף להקריב את החיילים המגנים הפשוטים על מנת לשמור על חיי מלכם.

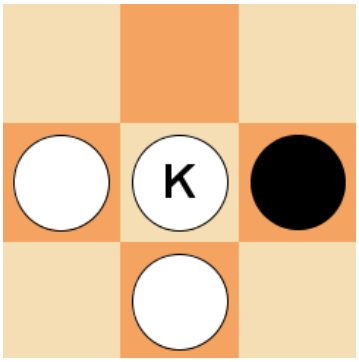
כותרת הפעולה המונה את רמת ההגנה של המלך:

```
/**
 * Find how many white Pawns circles Around the king
 * in order to keep him safe
 * @param whites whites bit Board
 * @param kingPos king bit position on the bit Board
 * @return count of whites Pawns around the king
 */
public static int protectKing(int kingPos, BitSet whites) {
```

טענת יציאה: מניית כמות המגנים הלבנים ב4 המשבצות הסמוכות למלך.
יעילות: O(1)

להלן תיאור פסאודו קוד: בדיקת מצב הביטים בלוח הסיביות של הכלים הלבנים במיקומי התאים המקיפים את המלך.

דוגמא למצב שבו הפעולה תחזיר ערך 2
כתוצאה משני חיילים מגנים המקיפים את המלך



מרחק המינימלי של המלך לאחת ממשבצות הבריחה:

פרמטר חשוב נוסף אשר משפיע על מתן הציון למצב לוח המשחק הוא המרחק המינימלי של המלך לאחת ממשבצות הבריחה הפנויות.
מרחק זה נקרא **"מרחק מנהטן"**. **מונח המגיע מתורת הגרפים.** מרחק זה הוא מרחק לכלי אשר יכול לנוע במאוזן ובמאונך בלבד בדומה לתנועת כלי המשחק שלנו.

מרחק זה בין שתי נקודות מחושב ע"י הנוסחה:

$$|x_1 - x_2| + |y_1 - y_2|$$

(האיבר המחובר מימין מייצג את התנועה במקביל לציר ה-y, וזה משמאל את התנועה במקביל לציר ה-x).

"התנועה" נעשית רק במקביל לאחד משני הצירים – על מנת לעבור מנקודה אחת לאחרת, יש לנוע מרחק מסוים ימינה (במקביל לציר ה-x) ומרחק מסוים מעלה (במקביל לציר ה-y), כך שהמסלול שנבחר כקצר ביותר בגאומטריה האוקלידית (אלכסון) אינו חוקי כאן.

<u>השחקן הלבן רוצה למזער את המרחק המינימלי ממשבצות הבריחה פנויה לעומת השחקן השחור:</u>

<u>כותרת הפעולה:</u>

```
/**
 * Find the minimum manhattan Distance for the king to
 * one of the free escape cells
 * @param xPos x Position of the king on the board
 * @param yPos y Position of the king on the board
 * @param fullBoard bit Set Of all the pieces on the board
 * @return the minimum manhattan Distance found from the king
 * to escapes
 */

public static int minDistanceToCorner(int xKing, int yKing, BitSet fullBoard) {
```

<u>פסאודו קוד:</u>

1. המרחק המינימלי -> המרחק המקסימלי האפשרי למשבצת בריחה = 7
2. עבור כל משבצת בריחה פנויה:
    - 2.1 מרחק מנהטן -> מרחק מנהטן בין מיקום המלך למיקום משבצת הבריחה
    - 2.2 המרחק המינימלי = מינימום(המרחק המינימלי , מרחק מנהטן)
    - 2.3 אם המרחק המינימלי:
        - 2.3.1 סיים חיפוש
3. החזר את המרחק המינימלי

<u>יעילות הפונקציה:</u> מעבר על איברי מערך מיקומי משבצות הבריחה O(N) בגודל 16

מצבי ניצחון:

בפונקציית הערכה הציון הגבוה ביותר שניתן עבור שחקן הוא ניצחון / הפסד.
כאשר ניצחון מעניק 1000 נקודות והפסד 1000-.

מניקוד זה מופחת מס' התור המבוצע בשביל להבטיח הגעה לניצחון בדרך המהירה ביותר.
כלומר אם הבוט מצא אפשרות לנצח את המשחק ב3 מהלכים לעומת מהלך אחד, אני רוצה שינצח במהלך אחד.
בדיקת מצבי ניצחון נבדקת רק לאחר 5 ניסיונות משחק מכיוון שזה מספר התורות המינימלי להגיע לניצחון.

בדיקת ניצחון עבור שחקן שחור:

1. אם לוח הסיביות של המלך ריק (מלך נלכד)
   החזר אמת
2. אם השחקן הנוכחי הוא שחקן לבן ואין לו אף תור לבצע
   החזר אמת
3. החזר שקר

בדיקת ניצחון עבור שחקן לבן:

1. אם יש ביט משותף הדלוק גם בלוח הסיביות של המלך וגם בלוח סיביות של משבצות הבריחה
   החזר אמת
2. אם השחקן הנוכחי הוא שחקן שחור ואין לו אף תור לבצע
   החזר אמת
3. החזר שקר

<u>יעילות בשתי הפונקציות:</u>
כאשר אני בודק האם יש לשחקן תור לבצע, אני לא מחפש את כל המהלכים האפשריים ובודק האם הוא שווה לאפס, אני עוצר כאשר אני פוגש מהלך חוקי אחד.

היעילות היא כמספר השחקנים של היריב (במקרה הכי גרוע 16 שחקנים לשחקן השחור) מכיוון שלכל כלי אני בודק ב- o(1) האם יש לו מהלך חוקי, לכן היעילות היא O(n).

תרשים זרימה- חישוב ציון פונקציות הערכה

הסבר בחירת האסטרטגיה

כחלק מכתיבת הבינה המלאכותית, החלק המשמעותי ביותר בפרויקט האתגר הגדול ביותר היה לבחור את האסטרטגיה הטובה ביותר עבור השחקן הממוחשב. חשוב לי לציין שהרבה פרמטרים נלקחו בחשבון אך לאחר ניסוי ותהייה, הרבה חקר וניתוח ביצועי השחקן הממוחשב הגעתי לתוצאה שאני גאה בה.
פרמטרים נוספים אשר שקלתי להכניס לפונקציית הערכה אך לא נבחרו לתוצר הסופי:

- ספירת כמות המהלכים עבור כל שחקן.
- חישוב כמות אפשרויות להגיע לפינה ב- 2 מהלכים.

<u>הסבר מתן המשקלים:</u>

לאחר בחירת הגורמים המשפיעים על ציון לוח המשחק.

חלק חשוב הוא לתת משקל עבור כל גורם – כלומר להחליט עד כמה הוא משפיע על המשחק ביחס לגורמים האחרים. לכל גורם ניתן ציון בין 0 ל-1 ולאחר מכן ערך הגורם מוכפל במשקלו על מנת שציוני הלוחות ינועו בין 0 ל100. כלומר כל פרמטר ניתן משקל לפי אחוזים בין 1-100.

דרך זאת היא מקובלת מכיוון שנוח לנו כבני אדם להשוות את חשיבות המשקלים באחוזים.

בחירת משקלים הוא תהליך קשה וכולל הרבה ניסיונות ושינויים. לאחר כל שינוי משקל שביצעתי בדקתי את השפעותיו על ביצועי השחקן הממוחשב ע"י משחק נגד המחשב מספר רב של פעמים.

כמו כן, דרך נוספת לבחירת המשקלים היא לתת לשחקנים ממוחשבים לשחק אחד נגד השני כאשר בכל אחד מהם ניתנו משקלים אחרים לגורמי האסטרטגיה על מנת לבחון מה הם המשקלים אשר מנצחים מספר רב כנגד יריביהם.

<u>המשקלים הסופיים שניתנו:</u>

לאחר הרבה שינויים ולמידה ניסיון במשחק של כמס' חודשים אלו הן המשקלים אשר בחרתי.

שם הפרמטר	משקל עבור שחקן שחור	משקל עבור שחקן לבן
כמות הכלים הלבנים המגנים	16* מספר הלבנים שנאכלו	20* מספר הלבנים בחיים
כמות הכלים השחורים התוקפים	16* מספר השחורים שבחיים	12* משחק השחורים שנאכלו
מיקום הכלים במקומות אסטרטגיים בלוח (בתחילת המשחק בלבד)	5	5
כמות הכלים השחורים והמחנות אשר "מאיימים" על המלך	16	-16
כמות הדרכים למלך להגיע לאחת מפינות הלוח בתור אחד	-32	32
המרחק המינימלי של המלך לאחת מהפינות הריקות	-10	10
כמות הכלים הלבנים אשר "מגנים" על המלך	-10	10

הבדל המשקלים בין השחקנים

ניתן להבחין שכיוצא מסימן המינוס המסמן האם המשקל הוא חיובי או שלילי עבור השחקן, מרבית המשקלים שווים בערכם המוחלט בין השחקנים.

ההבדל היחיד המשמעותי הוא שעבור השחקן הלבן – ניתן יותר משקל עבור מס' החיילים הלבנים שבחיים לעומת השחקנים השחורים שנאכלו. כתוצאה מכך שלשחקן השחור יתרון מספרי של כלים, ולכן יותר חשוב לשחקן הלבן לשמור על כליו בחיים מאשר לאכול את יריביו.

השפעת התקדמות המשחק על ציוני פונקציית הערכה

בפונקציית הערכה אשר בניתי מס' התורות ששוחקו משפיע בדרכים שונות:

א. השפעת תאים:
הפרמטר אשר הוסבר לעיל על משקלי מיקומי החיילים מחושב רק 15 התורות הראשונים. כתוצאה מכך שלאחר מס' תורות רבים, מיקומי החיילים פזורים לאורך כל הלוח ולכן קשה להעריך עד כמה משבצת אחת טובה עבור כל שחקן. לעומת תחילת המשחק בה אנו יודעים את המשבצות החשובות.

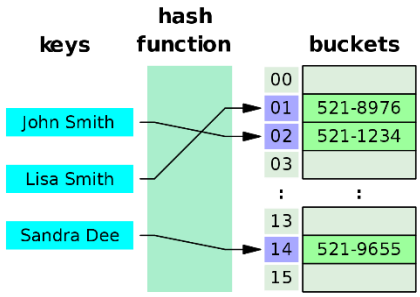
ב. השפעת התקדמות לבן:
לאחר תור מספר 35 הוחלט לגרום לשחקן הלבן להיות יותר התקפי ולהכפיל את משקל ההגעה למשבצת בריחה בתור אחד ב1.2.

ג. השפעת התקדמות שחור:
לאחר תור מספר 35 הוחלט לגרום לשחקן השחור להיות יותר התקפי ולהכפיל את משקל האיום על המלך ב1.2.

ד. מצבי ניצחון נבדקים לאחר תור מספר 5 מכיוון ש5 תורות זה מספר התורות המינימליים של אחד השחקנים להגיע לניצחון לכן אין סיבה לבדוק לפני מס' תור זה האם אחד השחקנים ניצח.

משקלים אלו שמורים במבנה הנתונים HashMap

במדעי המחשב, טבלת גִּבּוּב או טבלת ערבול (באנגלית: Hash table), היא מבנה נתונים מילוני, אשר נותן גישה לרשומה באמצעות המפתח המתאים לה.



המבנה הזה עובד באמצעות הפיכת המפתח על ידי פונקציית הגיבוב, למספר המייצג מיקום במערך שמפנה אל הרשומה המבוקשת.

בחירת מבנה נתונים זה כתוצאה מזמן חיפוש ממוצע של איבר ביעילות זמן O(1)

עבור כל שחקן שמור מבנה נתונים אשר שומר את המשקלים עבור השחקן

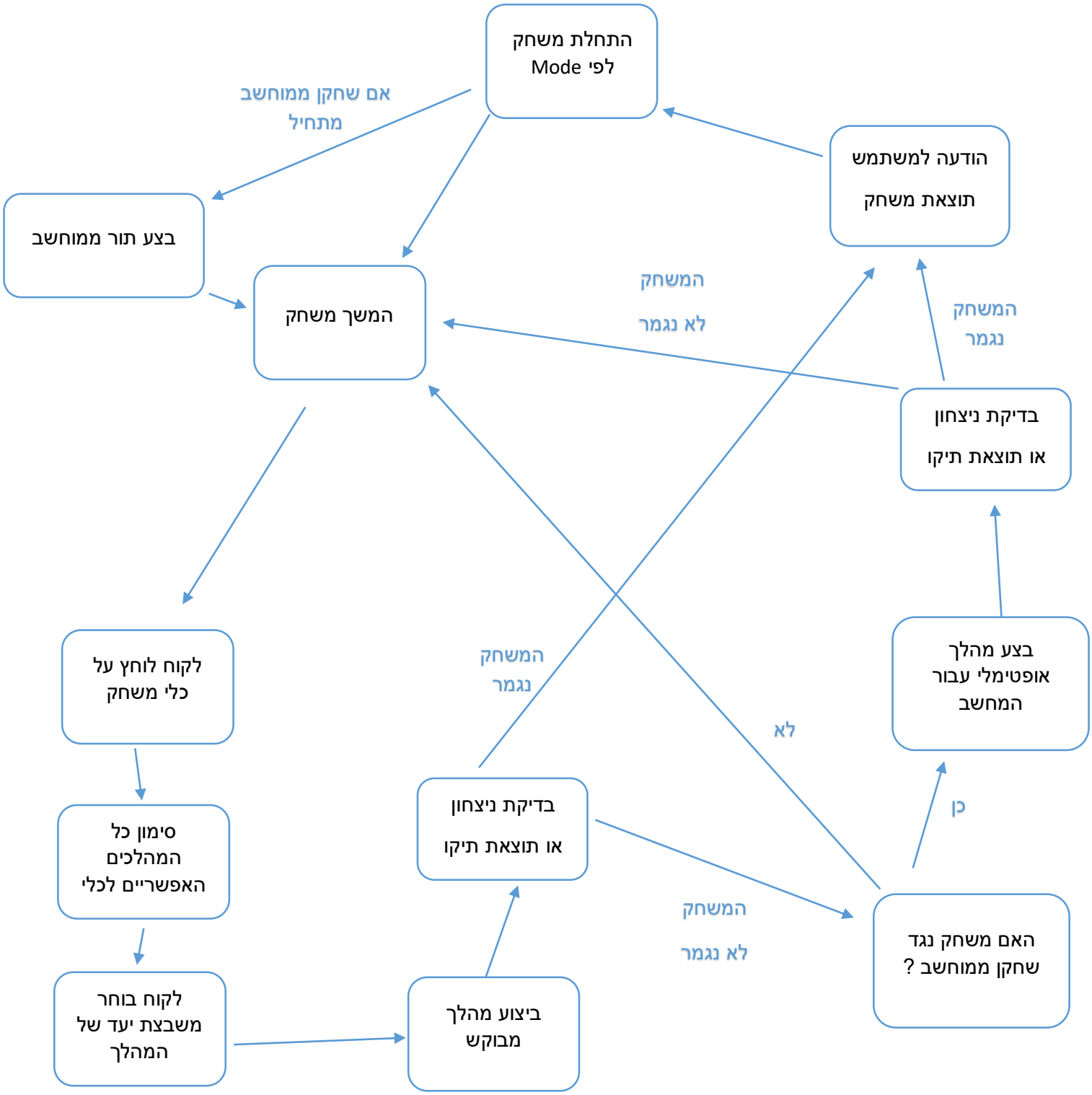
מבנה הנתונים של משקלי השחקן הלבן

```
private final HashMap<String, Integer> whiteHuristics = new HashMap<String, Integer>() {
    {
        put(WHITE_COUNT, 20);
        put(BLACK_COUNT, 12);
        put(KING_ADVANCE_CORNERS, 10);
        put(KING_PROTECTION, 12);
        put(DANGER_KING, -14);
        put(KING_PATHS, 32);
        put(CELL_WEIGHTS, 5);
    }
};
```

מבנה הנתונים של משקלי השחקן השחור

```
private final HashMap<String, Integer> blackHuristics = new HashMap<String, Integer>() {
    {
        put(WHITE_COUNT, 16);
        put(BLACK_COUNT, 16);
        put(KING_ADVANCE_CORNERS, -10);
        put(KING_PROTECTION, -12);
        put(DANGER_KING, 14);
        put(KING_PATHS, -32);
        put(CELL_WEIGHTS, 5);
    }
};
```

תרשים זרימה – מהלך המשחק



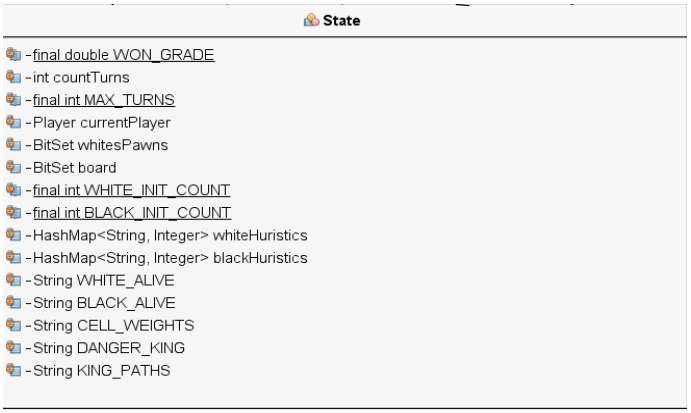
תיאור המחלקות הראשיות בפרויקט ופונקציות נבחרות

בחלק אציג את הפונקציות שבכל אחת מהמחלקות הראשיות של הפרויקט.

כמו כן אבחר מתוך המחלקות 2-3 פעולות חשובות ואפרט עליהן:

שם המחלקה TablutState

המחלקה מממשת את המנשק IState. מטרת מחלקה זאת היא לשמור מידע על לוח המשחק ומצבו בכל רגע נתון.



כותרת פונקציה	הסבר
public TablutState()	בנאי המחלקה
isWinningState() public Boolean	פעולה המחזירה אחד השחקנים ניצח
public boolean blackHasWon()	פעולה הבודקת האם השחקן השחור מנצח
whiteHasWon() public Boolean	פעולה הבודקת האם השחקן הלבן ניצח
private boolean hasMove()	פעולה הבודקת האם לשחקן הנוכחי יש מהלכים לבצע
public void performMove(Action action)	פעולה המקבלת מהלך ומבצעת אותו עבור השחקן הנוכחי
public List<Action> getAvailablePawnMoves()	פעולה המחזירה רשימה של מהלכים חוקיים לשחקן לבצע בתור הנוכחי
public boolean isGameOver()	פעולה המחזירה ערך בוליאני האם המשחק יסתיים או לא
public double getHuristic()	פעולה המחזירה את ציון הערכה עבור מצב הלוח הנוכחי ביחס לשחקן שתורו לבצע מהלך
public void cancelMove(CapturedPawns Captures, Action move)	פעולה המקבלת מהלך, ועצם CapturedPawns עם נתונים אודות המהלך ומבטלת את המהלך, הפעולה "מחזירה" לחיים את הכלים שנלכדו במהלך שהתקבל כפרמטר

פונקציה performAction

טענת כניסה:

הפונקציה מקבלת מהלך חוקי במשחק ומבצעת אותו

טענת יציאה:

הפונקציה מחזירה עצם מסוג CapturedPawns המכיל לוח סיביות של הכלים הרגילים שנאכלו כתוצאה מהתור והאם המלך נאכל או לא?

יעילות הפונקציה:

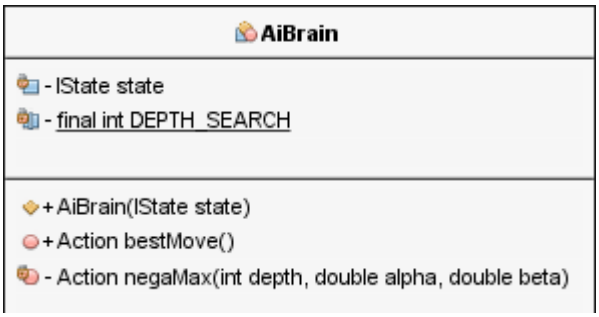
פעולות לוגיות כמו and ו not מעניקות לפעולה סיבוכיות של O(1)

להן פסואדו קוד:

1. לכידות -> קבל לוח סיביות של כל הכלים הנלכדו במהלך הפעולה (פעולה getCapturedPawns)
2. נקה את כלי המשחק ממשבצת המוצא ע"י כיבוי סיבית מתאימה בלוח הסיביות של השחקן
3. הצב כלי משחק במשבצת היעד ע"י הפעלת סיבית מתאימה בלוח הסיביות של השחקן
4. אם המהלך של שחקן השחור
    4.1 בדוק האם המלך נאכל, אם כן:
        4.1.1 כבה סיבית המלך בלוח סיביות של המלך
    4.2 בצע פעולה לוגית AND NOT בין לוח סיביות לבנים לבין לוח סיביות כלים לכודים
5. אחרת:
    5.1 בצע פעולה לוגית AND NOT בין לוח סיביות שחורים לבין לוח סיביות כלים לכודים
6. שנה שחקן נוכחי
7. הגדל ב1 את מספר התורות ששוחקו
8. עדכן את לוח הסיביות של המשחק כולו (פעולת איחוד בין כל לוחות הסיביות)
9. החזר עצם המורכב מלוח הסיביות של הכלים האכולים ומיקום המלך (אם נאכל)

שם המחלקה AiBrain

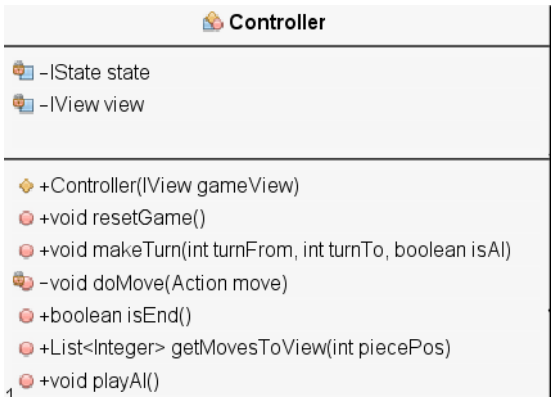
המחלקה אשר אחראית לבינה המלאכותית של המשחק ע"י שימוש בפונקציה negaMax.



כותרת פונקציה	הסבר
public AiBrain (IState boardGame)	בנאי המחלקה מקבל מצב לוח
public Action bestMove()	פונקציה המחזירה את המהלך הטוב ביותר עבור השחקן הנוכחי לשחק – הפונקציה קוראת לפונקציה NegaMax
private Action negaMax(int depth, double alpha, double beta)	פעולה המשתמשת באלגוריתם נגה מקס עם גיזום אלפא ביתא על מנת למצוא את המהלך האופטימלי לשחקן הנוכחי Depth: עומק העץ לחיפוש Beta: הערך המינימלי שמצאנו עד כה Alpha: הערך המקסימלי שמצאנו עד כה
moveOrdering(List<Action> moves)	פעולה המקבלת רשימה עם כל המהלכים האפשריים וממיינת את הרשימה בסדר יורד לפי פונקציית הערכה משנית (הוסבר בהרחבה בתיק)

שם המחלקה Controller

המחלקה אשר משמשת כ- "בקר" התוכנה. אחראית לידע את התצוגה על שינויים שהתבצעו בלוח המשחק. כמו כן, אחראית לשנות את לוח המשחק כתוצאה מהאזנה לאירועים בתצוגה כמו: לחיצה על כלי משחק. הבקר משמש כמעין קישור בין התצוגה למודל הלוגי.



כותרת פונקציה	הסבר
Controller(IView gameView)	בנאי המחלקה מקבל אובייקט של חלק התצוגה
public void resetGame()	פונקציה המאפסת את המשחק
public void makeTurn(int turnFrom, int turnTo, boolean isAI)	פעולה אשר מקבל מיקום מוצא ומיקום יעד ומזיזה את הכלי ממיקום המוצע בלוח למיקום היעד. כמו כן, אם הפרמטר isAI שווה לאמת, מבצעת גם תור לשחקן הממוחשב.
public void playAI()	פעולה אשר מבצעת תור עבור השחקן הממוחשב ומעדכנת את התצוגה בשינויי הלוח
public List<Integer> getMovesToView()	פעולה המחזירה רשימה של כל מיקומי המהלכים האפשריים עבור השחקן הנוכחי
public boolean isEnd()	פעולה אשר בודקת האם המשחק נגמר, אם כן מעדכנת את משתמשי המשחק

פונקציה makeTurn

טענת כניסה:

חלק התצוגה מעדכן את הבקר שהתבצע מהלך ומעביר לפונקציה את הפרמטרים מיקום משבצת היעד מיקום משבצת המוצא והאם המשחק מתנהל כנגד AI?

טענת יציאה:

הפונקציה מבצעת את המהלך במודל הלוגי ומעדכנת את חלק התצוגה בשינויי הלוח

יעילות הפונקציה: כיעילות הפונקציה "בצע-מהלך" שהוסברה לעיל: פעולות לוגיות כמו and ,not מעניקות לפעולה סיבוכיות של O(1)

להן פסואדו קוד:

1. בנה אובייקט מהלך
2. בצע מהלך במודל הלוגי
3. שלח לתצוגה את המהלך שהתבצע ואת לוח סיביות של הכלים הלכודים כתוצאה מן המהלך
4. אם המשחק נגמר:
    4.1 שלח לתצוגה שנגמר המשחק והתחל משחק חדש באותו MODE
5. אחרת:
    5.1 אם המשחק כנגד שחקן ממוחשב:
        5.1.1 בצע מהלך של שחקן ממוחשב
        5.1.2 אם המשחק נגמר:
            5.1.2.1 שלח לתצוגה שנגמר המשחק
            והתחל משחק חדש באותו MODE

שם המחלקה BitSetHelper

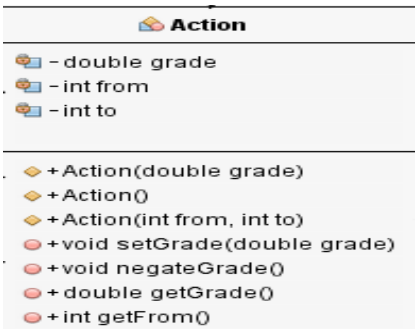
מחלקה סטטית עם פעולות עזר עבור מבנה הנתונים BitSet:

כותרת פונקציה	הסבר
public static BitSet newFromPositions(int[] positions)	המחלקה יוצרת ומחזירה מסיכה בגודל לוח המשחק שבה מיקומי הביטים הדלוקים מאוחסנים במערך positions שהתקבל כפרמטר
public static BitSet cloneAndResult(BitSet board1, BitSet board2)	פונקציה המקבלת 2 לוחות סיביות ומחזירה את תוצאת הAnd ביניהם ללא לשנות את לוחות הסיביות המקוריות

שם המחלקה Action

מחלקה אשר אחראית על מהלך במשחק Tablut. מהלך מאופיין ע"י ציון התור, משבצת יעד ומשבצת מוצא.

כותרת פונקציה	הסבר
public Action()	בנאי מחלקה אשר בונה מהלך בעל ציון מינימלי ומשבצת יעד ומוצא 1-
public Action(int from, int to)	בנאי מחלקה אשר בונה מהלך בעל ציון 0 ומשבצת יעד ומוצא לפי הפרמטרים שהתקבלו
public Action(double grade)	בנאי מחלקה אשר בונה מהלך עם ציון לפי פרמטר grade ומשבצת יעד ומוצא 1-
public void negateGrade()	פעולה המכפילה במינוס 1 את ציון המהלך כתוצאה מצרכי הנגה - מקס
actionBlackValue(int public int kingPosition)	פונקציית הערכה משנית למהלך של שחקן השחור מקבלת את מיקום כלי המלך מחזירה ציון מ-2-0 עד כמה התור טוב בסיבוכיות O(1)



Enum GamePositions

על מנת לזהות את מיקומי המשבצות נתתי לכל אחת מהמשבצות שם ע"י הטיפוס Enum.

A1("A1"), B1("B1"), C1("C1")

בקובץ זה בחרתי לבנות את המסיכות – לוחות הסיביות אשר משמשות אותי במהלך כתיבת התוכנית.

דוגמא לדרך בניית מסיכה של משבצות הבריחה:

1. בניית מערך של מספר שלמים שהערכים בו הם מיקומי המשבצות אשר יודלקו בלוח הסיביות.
פריסת שמות התאים הם בהתאם לפריסת בלוח המשחק המקורי.

```
/**
 * Array of ints representing all the escape cells on the board
 */
public static final int[] escapeCells = {
                B1.ordinal(),   C1.ordinal(),                                   G1.ordinal(),   H1.ordinal(),
        A2.ordinal(),                                                                           I2.ordinal(),
        A3.ordinal(),                                                                           I3.ordinal(),



        A7.ordinal(),                                                                           I7.ordinal(),
        A8.ordinal(),                                                                           I8.ordinal(),
                B9.ordinal(),   C9.ordinal(),                                   G9.ordinal(),   H9.ordinal()
};
```

2. בניית הלוח הסיביות ממערך זה:
שליחת המערך לפעולה אשר מקבל מערך ומחזירה לוח סיביות שבו הביטים הדלוקים הם אלו שאינדקסים שלהם מופיעים בלוח הסיביות

```
// Masks of special Cells on the board
public static final BitSet escape = BitSetHelper.newFromPositions(escapeCells);
public static final BitSet camps = BitSetHelper.newFromPositions(campCells);
public static final BitSet obstacles = BitSetHelper.newFromPositions(obstacleCells);
public static final BitSet kingSurrounded= BitSetHelper.newFromPositions(kingSurroundedCells);
```

שם המחלקה BoardMoves

מחלקה סטטית אשר אחראית על תזוזת ולכידת הכלים בלוח. הפעולות במחלקה זאת כוללות לוגיקה רבה וכתיבתה לקחה לי זמן רב.

כותרת פונקציה	הסבר
public static List<Action> getMovesForPawn(int pawnPosition, IState state)	הפעולה מחזירה רשימה של כל המהלכים של כלי משחק במיקום pawnPosition בלוח המשחק state
public static BitSet getCapturedPawns(Action move, IState state)	הפעולה מקבלת מהלך ואת לוח המשחק ומחזירה מסיכה עם כל מיקומי השחקנים שנאכלו כתוצאה ממהלך זה
private static BitSet getNormalCaptures(int position, BitSet attack, BitSet defense)	פעולה המקבלת מיקום היעד של מהלך, לוח סיביות של תוקפים ומגינים ומחזירה לוח סיביות עם כל מיקומי הכלים הלכודים בדרך רגילה במהלך זה
public static int dangerToKing(int kingPos, BitSet blacks)	פעולה המקבלת את מיקום המלך ולוח סיביות של הכלים השחורים ומחזירה כמה שחקנים לבנים / מחנות / הטירה מקיפים את המלך
public static int protectKing(int kingPos, BitSet whites)	פעולה המקבלת את מיקום המלך ולוח סיביות של הכלים הלבנים ומחזירה כמה שחקנים לבנים מקיפים את המלך
public in pawnsToEatKing(BitSet kingBoard)	הפעולה מקבלת את לוח הסיביות של המלך ומחזירה כמה כלים שחורים דרושים על מנת להרוג את המלך
public static int minDistanceToCorner(int xKing, int yKing, BitSet fullBoard)	פעולה המקבלת את מיקום המלך בציר הxy ואת לוח הסיביות של כל כלי המשחק ומחזירה את "מרחק מנהטן" המינימלי של המלך לאחת ממשבצות הבריחה הפנויות

פונקציה getMovesForPawn

טענת כניסה: הפונקציה מקבל הפנייה ללוח המשחק ומיקום ביט בלוח הסיביות

טענת יציאה: הפונקציה מחזירה רשימה של כל המהלכים האפשריים עבור כלי המשחק במיקום שהתקבל כפרמטר

יעילות הפונקציה: הפונקצייה עוברת על כל המשבצות הפנויות במאונך ובמאוזן לכלי הנבחר לכן יעילות הפונקציה היא במקרה הגרוע ביותר שווה למס' התורות המקסימליים שחייל יכול לנוע והיא 16

להן פסואדו קוד:

1. בנה מבנה נתונים שיחזיק את המהלכים החוקיים
2. בנה מסיכה של המקומות האסורים לחייל להגיע אליהם -> מקומות אסורים
   2.1 המסכה מורכבת מכל המקומות התפוסים בלוח + משבצת הטירה
3. אם הכלי הנע הוא של השחקן הלבן:
   3.1 הוסף מסיכה המקומות האסורים את מיקומי משבצות "המחנה" (הוספה ב OR)
4. אחרת:
   4.1 אם מיקום הכלי הוא לא בתוך מחנה:
      4.1.1 הוסף מסיכה המקומות האסורים את מיקומי משבצות "המחנה"

5. בדוק מהלכים אפשריים בכיוון למעלה
   עבור כל המיקומים שמעל הכלי הנבחר
   אם המשבצת חלק מהמקומות האסורים או לא בתחום הלוח : עצור
   אחרת: הוסף את המהלך למבנה הנתונים מהלכים חוקיים

6. בדיוק מהלכים אפשריים בכיוון מטה:
   עבור כל המיקומים שמתחת לכלי הנבחר
   אם המשבצת חלק מהמקומות האסורים או לא בתחום הלוח : עצור
   אחרת: הוסף את המהלך למבנה הנתונים מהלכים חוקיים

7. בדוק מהלכים אפשריים בכיוון ימינה:
   עבור כל המיקומים שמימין הכלי הנבחר
   אם המשבצת חלק מהמקומות האסורים או לא בתחום הלוח : עצור
   אחרת: הוסף את המהלך למבנה הנתונים מהלכים חוקיים

8. בדוק מהלכים אפשריים בכיוון שמאלה:
9. עבור כל המיקומים שמשמאל הכלי הנבחר
10. אם המשבצת חלק מהמקומות האסורים או לא בתחום הלוח : עצור
11. אחרת: הוסף את המהלך למבנה הנתונים מהלכים חוקיים

12. החזר את מבנה הנתונים המהלכים החוקיים

פונקציה getNormalCaptures

טענת כניסה: הפונקציה מקבלת מיקום משבצת יעד של מהלך, לוח סיביות של השחקן שמבצע את המהלך ולוח סיביות של היריב

טענת יציאה: הפונקציה מחזירה לוח סיביות אשר מייצג את המשבצות של הכלים האכולים כתוצאה מהמהלך

יעילות הפונקציה: בדיקת ארבעת הביטים מסביב למשבצת היעד מעניקה לפונקציה O(1)

להן פסואדו קוד:

1. בנה לוח סיביות בגודל הלוח אשר ביט דלוק יציין כלי נלכד במיקום הביט -> לכידות
2. אם משבצת היעד לא נמצאת בשתי השורות העליונות וגם מעליה יש בו כלי של היריב (המגן)
    2.1 אם בתא שמעל הכלי של היריב יש בו כלי של השחקן הנוכחי
    או משבצת מחנה / משבצת טירה
        2.1.1 הדלק ביט מתאים בלוח הלכידות

3. אם משבצת היעד לא נמצאת בשתי השורות התחתונות וגם מתחתיה יש כלי של היריב (המגן)
    3.1 אם בתא שמתחת הכלי של היריב יש בו כלי של השחקן הנוכחי
    או משבצת מחנה / משבצת טירה
        3.1.1 הדלק ביט מתאים בלוח הלכידות

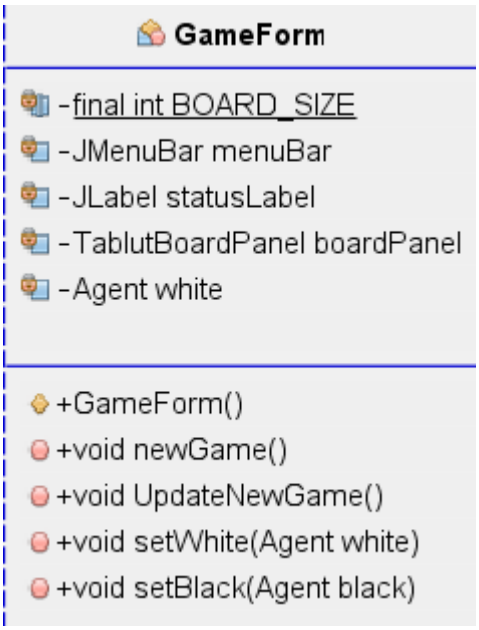
4. אם משבצת היעד לא בשתי העמודות הימניות ביותר וגם משמאלה יש כלי של היריב (המגן)
    4.1 אם התא שמשמאל הכלי של היריב יש כלי של השחקן הנוכחי שתוקף או
    משבצת מחנה / משבצת טירה
        4.1.1 הדלק ביט מתאים בלוח הלכידות

5. אם משבצת היעד לא בשתי העמודות השמאליות ביותר וגם מימינה יש כלי של היריב (המגן)
    5.1 אם התא שמימין הכלי של היריב יש כלי של השחקן הנוכחי שתוקף או משבצת
    מחנה / משבצת טירה
        5.2.1 הדלק ביט מתאים בלוח הלכידות

6 החזר לוח סיביות של לכידות

שם המחלקה GameForm

מחלקה זאת יורשת ממחלקה JFrame. שהיא מחלקה של Java Swing, אשר משמשת לבניית את חלון התצוגה של התוכנה. חלון התצוגה מורכב מאובייקטים של swing אשר עוזרים לבנות את הGUI כמו JLabe, JMenuBar.



כותרת פונקציה	הסבר
public GameForm()	בנאי המחלקה המאתחל את אובייקטים של swing ובונה את לוח המשחק
public void newGame()	הפעולה מאתחלת את התצוגה ובונה משחק חדש
public void UpdateNewGame()	הפעולה מציגה ללקוח את פרטי המשחק החדש המתחיל
public void updateTurns(int numberOfTurns)	הפונקציה מעדכנת את מס' התור המשוחק כעת

שם המחלקה TablutBoardPanel

מחלקה זאת יורשת ממחלקה BoardPanel. וממשת את מנשקי MouseListener ו- IView. מחלקה זאת אחראית על תצוגת לוח המשחק כולל: כלי המשחק, משבצות הלוח ועוד.

כותרת פונקציה	הסבר
Public TablutBoardPanel(GameForm mainView)	בנאי המחלקה אשר מאתחל את לוח המשחק למצב ההתחלתי, מקבל הפנייה לחלון הראשי של האפליקצייה
private void initBoard()	פעולה המאתחלת את מבנה הנתונים boardPieces אשר שומר נתונים על הקורדינטות של משבצות המשחק
public void resetGame()	הפעולה מאתחלת את התצוגה ובונה משחק חדש
public void drawBoard(Graphics g)	הפונקציה המציגה את לוח המשחק
public void updateGameDetails(BitSet captures, Action move)	הפונקציה מעדכנת את לוח המשחק כתוצאה מהמהלך move אשר הלכידות שלו מצויות באובייקט captures
public void printDraw()	הפעולה מודיע למשתמש על תוצאת תיקו ומאתחלת משחק חדש
public void printWin()	הפעולה מודיע למשתמש על תוצאת ניצחון ומאתחלת משחק חדש
public void mousePressed(MouseEvent e)	פעולה הנקראת כאשר הלקוח לוחץ על לוח המשחק, הפעולה מבצעת את המהלך אשר הלקוח ביקש אם הוא חוקי
private static boolean clickInSquare(int x, int y, int cx, int cy)	הפעולה בודקת האם הנקודה (x,y) נמצא בתוך המעגל שמרכזו (cx,cy)
private static boolean clickInCircle(int x, int y, int cx, int cy)	הפעולה בודקת האם הנקודה (x,y) נמצא בתוך הריבוע שמרכזו (cx,cy)

פונקציה updateGameDetails

טענת כניסה: הפונקציה מקבלת מהלך שקרה במשחק ולוח סיביות המסמן באיזה מיקומים נאכלו כלים כתוצאה מן המהלך

טענת יציאה: הפונקציה מעדכנת את חלק התצוגה בשינויי המשחק

יעילות הפונקציה: מעבר בלולאה על כל הביטים הדלוקים של לוח הסיביות של הכלים הלכודים, מס' הכלים הלכודים בתור הוא מקסימום 3, "מציאת הביט הדלוק הבא" ממומש ע"י Java ביעילות O(1)

להן פסואדו קוד:

1. עבור כל אחד מהביטים הדלוקים בלוח סיביות של הכלים הלכודים הפוך המשבצת המתאימה בלוח לריקה
2. הפוך משבצת המוצא של המהלך לריקה
3. הפוך משבצת היעד של המהלך לסוג החייל שביצע את התור
4. רענן את לוח התצוגה על מנת להראות שינויים שחלו

פונקציה mousePressed

טענת כניסה: הפונקציה מקבלת "אירוע" שקרה כתוצאה מלחיצת הלקוח על לוח המשחק

טענת יציאה: הפונקציה בודקת את תקינות האירוע ומבצעת את המהלך המתבקש אם הוא חוקי

יעילות הפונקציה: במקרה הגרוע, בו השחקן לחץ על כלי משחק שלו, מופעלת הפונקצייה במודל המחפשת את כל המהלכים האפשריים עבור כלי משחק, יעילות פונקציה זאת היא O(N) , כאשר N הוא מספר המהלכים המקסימיליים לכלי משחק – 16.

להן פסואדו קוד:

1. חשב ע"י מיקום לחיצת הכפתור את מיקום המשבצת הלחוצה
2. אם כלי לא נלחץ עדיין:
    2.1 אם הלקוח לחץ על כלי משחק:
        2.1.1 אם הכלי הנלחץ הוא לא של השחקן הנוכחי:
            2.1.1.1 אפס את מבנה הנתונים של המהלכים האפשריים
        2.1.2 אחרת:
            שמור את כל המהלכים האפשריים של כלי המשחק הנבחר
3. אחרת:
    3.1 אם משבצת היעד היא במבנה הנתונים השומר את המהלכים החוקיים לכלי הנבחר:
        3.1.1 בצע מהלך

    3.2 אפס את מבנה הנתונים השומר את המהלכים החוקיים לכלי משחק הנבחר

4. רענן את לוח התצוגה על מנת להראות שינויים שחלו

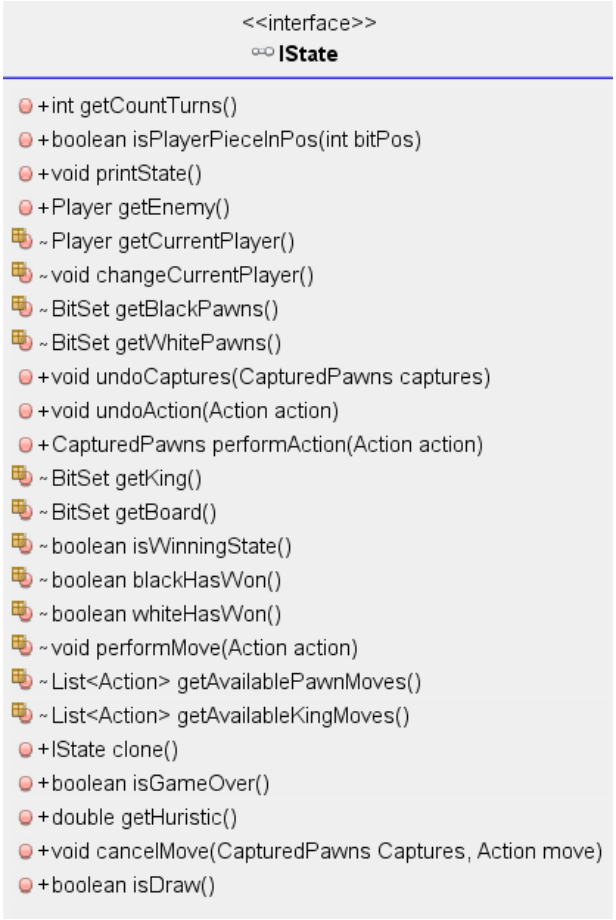
InterFaces ממשקים

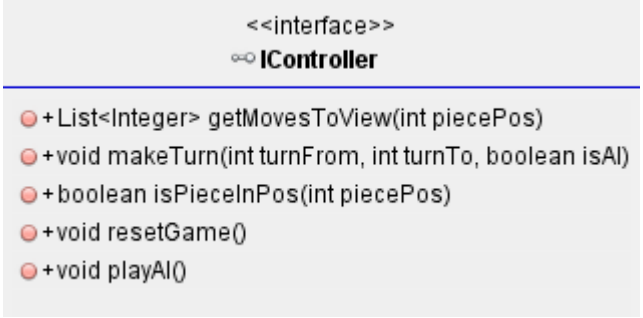
interface (ממשק) הינו מבנה לוגי מופשט (אבסטרקטי) המכיל רק הצהרות. אחד מתפקידי ה-interface העיקריים הוא ליצור ממשק זהה לאובייקטים שונים ובכך הוא אחד היישומים של עיקרון הפולימורפיזם בתכנות מונחה העצמים (OOP).

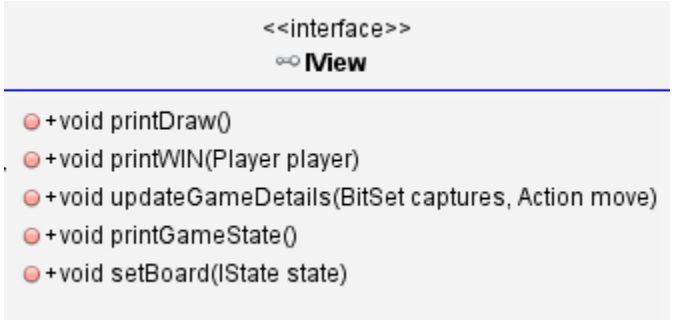
הממשקים IController IView, IState

כחלק מארכיטקטורת עיצוב התוכנה, לבקר יהיה הפנייה לממשק התצוגה והמודל הלוגי.

כמו כן, לחלק התצוגה יהיה הפנייה לממשק לבקר.

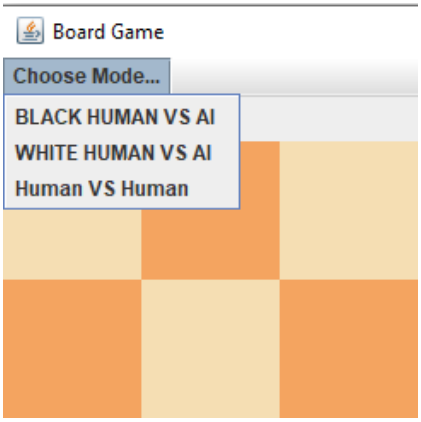






מדריך למשתמש

כאשר הלקוח מפעיל את האפליקציה, המצב ההתחלתי כברירת המחדל הוא משחק נגד שחקן אנושי.
בסרגל האפשרויות מצד שמאל למעלה יש לו אפשרות לבחור בין 3 מצבי המשחק:



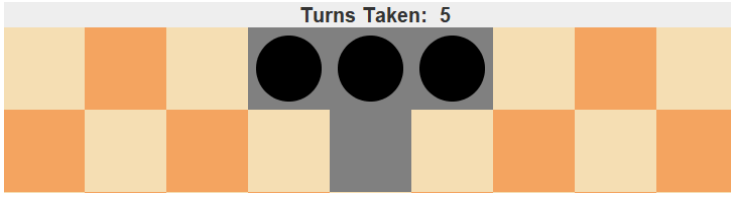
- משחק אנושי
- משחק כתוקף נגד המחשב
- משחק כמגן כנגד המחשב

במהלך המשחק, הלקוח מבצע מהלך ע"י לחיצה על כלי משחק שלו.

לאחר לחיצה, האפליקציה מראה ללקוח את כל המהלכים האפשריים של הכלי לנוע בשלב זה ועל השחקן לבחור את משבצת היעד המבוקשת.

במהלך המשחק, מופיע במעלה המסך את מס' התור המשוחק בכל שלב,

יש לשים לב במשחק ה Tablut לאחר 100 תורות נקבעת תוצאת המשחק לתיקו ולכן ב-10 תורות האחרונים מס' התור יופיע באדום על מנת להתריע לשחקנים שהמשחק לקראת סיומו.



על מנת להתחיל משחק מחדש יש לבחור את מצב המשחק המבוקש בסרגל מצבי המשחק.

הרצת משחק: על מנת להריץ את המשחק. יש להוריד למחשב Java SE Run Time Environment 8.

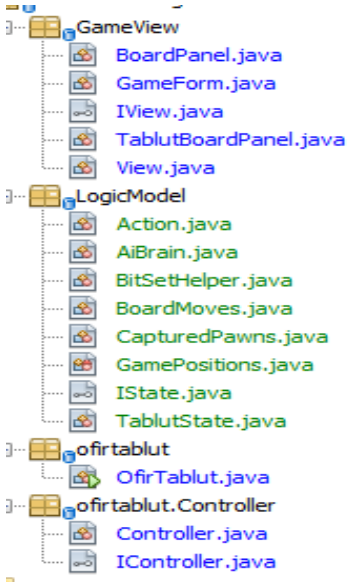
מהאתר של oracle בקישור:

https://www.oracle.com/java/technologies/javase-jre8-downloads.html

הפעולה הראשית Main נמצאת בקובץ OfirTablut.java

קוד הפרויקט

רשימת קבצים של הפרויקט:



בפרק זה אציג את קוד הפרויקט שנכתב על ידי כולל תיעוד לפי חוקי JAVA DOC

קובץ TablutState

```
public class TablutState implements IState {

    private int countTurns = 0;

    // Bits board
    private Player currentPlayer;
    private final BitSet whitesPawns, kingPawn, blackPawns;
    private final BitSet board;

    // Consts
    private static final int WHITE_INIT_COUNT = 8;
    private static final int BLACK_INIT_COUNT = 16;
    private static final double WON_GRADE = 1000;
    private final int COUNT_NEIGHBORS = 4;
    private final int MIN_WIN_MOVES = 5;

    // evaluation Functions Influencers
    private final String KING_PROTECTION = "White Pawns protects The king";
    private final String WHITE_COUNT = "White Pawns Alive";
    private final String BLACK_COUNT = "Black Pawns Alive";
    private final String KING_ADVANCE_CORNERS = "Min manhattan distance To
escapes Cells";
    private final String DANGER_KING = "Black pawns threats the king";
```

```
    private final String KING_PATHS = "Free paths to king to reach the
corner";
    private final String CELL_WEIGHTS = "Strategic cells Weights";


     * Deep copy constructor
     */
    public TablutState(Player currentPlayer, BitSet whitesPawns, BitSet
kingPawn, BitSet blackPawns, int turns) {
        this.currentPlayer = (Player) currentPlayer;
        this.whitesPawns = (BitSet) whitesPawns.clone();
        this.kingPawn = (BitSet) kingPawn.clone();
        this.blackPawns = (BitSet) blackPawns.clone();
        this.countTurns = turns;

        // Create Board from all the pieces
        this.board = new BitSet(IState.BOARD_DIMENSION);
        this.board.or(whitesPawns);
        this.board.or(blackPawns);
        this.board.or(kingPawn);
    }

    /**
     * build initial game constructor
     */
    public TablutState() {
        // The white player is always start the game
        this.currentPlayer = Player.WHITE;

        // Create the initial board of the game
        this.whitesPawns = (BitSet) GamePositions.initWhite.clone();
        this.blackPawns = (BitSet) GamePositions.initBlack.clone();
        this.kingPawn = (BitSet) GamePositions.initKing.clone();

        // Create Board from all the pieces
        this.board = new BitSet(IState.BOARD_DIMENSION);
        updateBoard();
    }

    //************************** IState Functions
****************************************/
    @Override
    public int getCountTurns() {
        return countTurns;
    }

    @Override
    public Player getCurrentPlayer() {
        return this.currentPlayer;
    }

    @Override
    public boolean isPlayerPieceInPos(int bitPos) {
        if (this.currentPlayer == Player.BLACK) {
            return this.blackPawns.get(bitPos);
        }
        return this.whitesPawns.get(bitPos) || this.kingPawn.get(bitPos);
```

```
    }

    private void updateBoard() {
        this.board.or(whitesPawns);
        this.board.or(blackPawns);
        this.board.or(kingPawn);
    }

    @Override
    public IState clone() {
        return new TablutState(this.currentPlayer, this.whitesPawns,
this.kingPawn, this.blackPawns, this.countTurns);
    }

    @Override
    public BitSet getBoard() {
        return this.board;
    }

    @Override
    public boolean isDraw() {
        return this.countTurns >= TablutState.MAX_TURNS;
    }

    @Override
    public boolean isWinningState() {
        return blackHasWon() || whiteHasWon();
    }

    /**
     * The function check if the black player is the winner
     *
     * @return if black won return true else return false
     */
    @Override
    public boolean blackHasWon() {
        if (kingPawn.isEmpty()) {
            return true;
        }
        return currentPlayer == Player.WHITE && !hasMove();
    }

    /**
     * The function check if the white player is the winner
     *
     * @return if white won return true else return false
     */
    @Override
    public boolean whiteHasWon() {
        if (kingPawn.intersects(GamePositions.escape)) {
            return true;
        }
        return currentPlayer == Player.BLACK && !hasMove();
    }

    /**
     * @return True if the current Player has At least one Move, else return
```

```
     * False
     */
    private boolean hasMove() {
        BitSet boardOfPlayer = this.currentPlayer == Player.WHITE ?
this.whitesPawns : this.blackPawns;

        // get all pawns Moves
        for (int i = boardOfPlayer.nextSetBit(0); i >= 0; i =
boardOfPlayer.nextSetBit(i + 1)) {
            if (BoardMoves.getMovesForPawn(i, this).isEmpty() == false) {
                return true;
            }
        }

        // if the current player is White so add King Moves
        if (this.currentPlayer == Player.WHITE) {
            if (this.getAvailableKingMoves().isEmpty() == false) {
                return true;
            }
        }

        return false;
    }


    /**
     * find all possible Moves of the current Player according to the Tablut
     * Rules
     *
     * @return list of all the available moves for this currentPlayer
     */
    @Override
    public List<Action> getAvailablePawnMoves() {

        BitSet boardOfPlayer = currentPlayer == Player.WHITE ?
this.whitesPawns : this.blackPawns;

        List<Action> moves = new ArrayList<>(boardOfPlayer.cardinality() *
10);

        // if the current player is White so add King Moves
        // for Move Ordering i put them in the start of the List
        if (currentPlayer == Player.WHITE) {
            moves.addAll(getAvailableKingMoves());
        }

        // get all pawns Moves
        for (int i = boardOfPlayer.nextSetBit(0); i >= 0; i =
boardOfPlayer.nextSetBit(i + 1)) {
            moves.addAll(BoardMoves.getMovesForPawn(i, this));
        }



        return moves;
    }
```

```
    /**
     * @return list of all the possible actions for the king to move in this
state
     */
    @Override
    public List<Action> getAvailableKingMoves() {

        int kingPosition = this.kingPawn.nextSetBit(0);
        // Only to be sure
        if (kingPosition == -1) {
            return new ArrayList<>();
        }
        return BoardMoves.getMovesForPawn(kingPosition, this);

    }

    /**
     * @return the evaluation of the board for the current player
     */
    @Override
    public double getHuristic() {
        if (isDraw()) {
            return 0;
        }

        if (currentPlayer == Player.BLACK) {
            return getHuristicForBlack();
        }
        return getHuristicForWhite();
    }

    @Override
    public boolean isGameOver() {
        return isWinningState() || isDraw();
    }

    /**
     * perform the action of the current Player on this board
     *
     * @param action move to perform
     * @return CapturedPawns object with details about the captured pawn
eaten
     * in this action
     */
    @Override
    public CapturedPawns performAction(Action action) {
        BitSet captures = BoardMoves.getCapturedPawns(action, this);

        int fromIndex = action.getFrom();
        int toIndex = action.getTo();

        int kingCaptruedPos = -1;
        if (currentPlayer == Player.BLACK) {

            // this is move of black player
            blackPawns.clear(fromIndex);
            blackPawns.set(toIndex);
```

```
            int kingPos = this.kingPawn.nextSetBit(0);

            // check if a king is captured
            if (captures.get(kingPos)) {
                this.kingPawn.clear();
                kingCaptruedPos = kingPos;
            }

            this.whitesPawns.andNot(captures);

        } else {

            // this is move of white player
            // check if the player move the king or regular pawn
            BitSet played = this.whitesPawns;

            if (this.kingPawn.get(fromIndex)) {
                played = this.kingPawn;
            }
            played.clear(fromIndex);
            played.set(toIndex);
            blackPawns.andNot(captures);
        }

        // change the player
        this.changeCurrentPlayer();

        // update the main board
        this.board.clear(fromIndex);
        this.board.set(toIndex);
        this.board.andNot(captures);

        // inc the number of moves played
        this.countTurns++;
        return new CapturedPawns(captures, kingCaptruedPos);
    }

    /**
     * The function cancel "action" by undo it
     * @param action action that made in nega Max algorithm and needs to be
deleted
     */
    @Override
    public void undoAction(Action action) {
        BitSet pawns = this.blackPawns;
        if (this.currentPlayer == Player.BLACK) {
            // check if is it the king or regular white Pawn
            pawns = this.whitesPawns;
            if (this.kingPawn.get(action.getTo())) {
                pawns = this.kingPawn;
            }
        }
        pawns.clear(action.getTo());
        pawns.set(action.getFrom());

        this.board.clear(action.getTo());
```

```
        this.board.set(action.getFrom());
    }

    /**
     * cancel all the captures made represented in captures mask
     * @param captures mask of captures made while searching in the tree
games
     */
    @Override
    public void undoCaptures(CapturedPawns captures) {
        BitSet capturedPawns = captures.getCaptured();
        if (this.currentPlayer == Player.BLACK) {
            this.blackPawns.or(capturedPawns);
        } else {
            int kingPos = captures.getKingPos();
            // check if the king captured
            if (kingPos != -1) {
                this.kingPawn.set(kingPos);
                capturedPawns.clear(kingPos);
            }
            this.whitesPawns.or(capturedPawns);
        }
        updateBoard();
    }

    /**
     * Cancel the move taken and return the board to the state before
     *
     * @param Captures CapturedPawns object with the details about the
captures
     * @param move the move to cancel
     */
    @Override
    public void cancelMove(CapturedPawns Captures, Action move) {

        undoAction(move);

        // set the captured to '1' again
        undoCaptures(Captures);

        // change the player back
        changeCurrentPlayer();

        // dec the move counter
        countTurns--;
    }

    //  the weights of the different parameters to evaluate the game state

    private final HashMap<String, Integer> whiteHuristics = new
HashMap<String, Integer>() {
        {
            put(WHITE_COUNT, 20);
            put(BLACK_COUNT, 12);
            put(KING_ADVANCE_CORNERS, 10);
            put(KING_PROTECTION, 10);
            put(DANGER_KING, -16);
```

```
            put(KING_PATHS, 32);
            put(CELL_WEIGHTS, 5);
        }
    };

    private final HashMap<String, Integer> blackHuristics = new
HashMap<String, Integer>() {
        {
            put(WHITE_COUNT, 16);
            put(BLACK_COUNT, 16);
            put(KING_ADVANCE_CORNERS, -10);
            put(KING_PROTECTION, -10);
            put(DANGER_KING, 16);
            put(KING_PATHS, -32);
            put(CELL_WEIGHTS, 5);
        }
    };


        /**

     * @return the evaluation grade of this Current state to the white Player
     */
    public double getHuristicForWhite() {
        // Check if is it possible to win according to the count of turns
made
        if (this.countTurns >= MIN_WIN_MOVES) {

            if (this.whiteHasWon()) {
                return WON_GRADE - this.countTurns;
            }

            if (this.blackHasWon()) {
                return (-1 * WON_GRADE) + this.countTurns;
            }
        }

        // get the bit number of the king
        int kingPos = this.kingPawn.nextSetBit(0);

        int blackCount = this.blackPawns.cardinality();
        int whiteCount = this.whitesPawns.cardinality();

        // Pawns Counter
        double whiteAlive = whiteCount / (double) WHITE_INIT_COUNT;
        double blackEaten = (BLACK_INIT_COUNT - blackCount) / (double)
BLACK_INIT_COUNT;

        // king danger
        int pawnsToEatKing = BoardMoves.pawnsToEatKing(kingPawn);

        double enemyNearKing = BoardMoves.dangerToKing(kingPos, blackPawns) /
(double) pawnsToEatKing;

        // kings protection
        double protectingKing = BoardMoves.protectKing(kingPos,
this.whitesPawns) / (double) COUNT_NEIGHBORS;
```

```
        // if this is the start of the game Consider the sum Weights of the
cells
        double strategicCells = this.countTurns < 15 ?
differenceSumWeights(whiteCount,blackCount) : 0;

        // Attempts to win the Game, divide by 1.3 because One way to win is
not always
        // promise a Win
        double winWays = freePathsToKing() / 1.3;

        int xKingPos = kingPos % EDGE_SIZE, yKingPos = kingPos / EDGE_SIZE;

        int minDistance = BoardMoves.minDistanceToCorner(xKingPos, yKingPos,
this.board);

        // MAX_DISTANCE_CORNER - minDistance
        // ---------------------------------------------  = 1 -
minDistance/MAX_DISTANCE_CORNER
        //     MAX_DISTANCE_CORNER

        double kingsAdvancement = 1 - (minDistance /
GamePositions.MAX_DISTANCE_CORNER);

        // evaluate according to the weights of each function
        double eval = blackEaten * whiteHuristics.get(BLACK_COUNT);
        eval += whiteAlive * whiteHuristics.get(WHITE_COUNT);
        eval += protectingKing * whiteHuristics.get(KING_PROTECTION);
        eval += enemyNearKing * whiteHuristics.get(DANGER_KING);
        eval += strategicCells * whiteHuristics.get(CELL_WEIGHTS);
        eval += kingsAdvancement * whiteHuristics.get(KING_ADVANCE_CORNERS);

        // From turn 35 and up, the white needs to be more Aggressive SO
Multipy the Weights of KING_PATHS by 1.2
        if (this.countTurns > 35) {
            eval += winWays  * whiteHuristics.get(KING_PATHS) * 1.2;
        } else {
            eval += winWays  * whiteHuristics.get(KING_PATHS);
        }

        return eval;
    }

     /**
     * @return the evaluation grade of this Current state to the black Player
     */
    public double getHuristicForBlack() {
         // Check if is it possible to win according to the count of turns
made
        if (this.countTurns >= MIN_WIN_MOVES) {
            if (this.blackHasWon()) {
                return WON_GRADE - this.countTurns;
            }

            if (this.whiteHasWon()) {

                return (-1 * WON_GRADE) + this.countTurns;
```

```
            }
        }

        // get the bit number of the king
        int kingPos = this.kingPawn.nextSetBit(0);

        int blackCount = this.blackPawns.cardinality();
        int whiteCount = this.whitesPawns.cardinality();

         // Pawns Counter
        double blackAlive = blackCount / BLACK_INIT_COUNT;
        double whiteEaten = (WHITE_INIT_COUNT - whiteCount) / (double)
WHITE_INIT_COUNT;

        // Danger to king
        double enemyNearKing = BoardMoves.dangerToKing(kingPos, blackPawns) /
(double) BoardMoves.pawnsToEatKing(this.kingPawn);

        // Attempts to win the Game, divide by 1.3 because One way to win is
not always
        // promise a Win
        double winWays = (double) freePathsToKing() / 1.3;

        int xKingPos = kingPos % EDGE_SIZE, yKingPos = kingPos / EDGE_SIZE;

        int minDistance = BoardMoves.minDistanceToCorner(xKingPos, yKingPos,
this.board);

        // MAX_DISTANCE_CORNER - minDistance
        // ---------------------------------------------  = 1 -
minDistance/MAX_DISTANCE_CORNER
        //     MAX_DISTANCE_CORNER

        double kingsAdvancement = 1 - (minDistance /
GamePositions.MAX_DISTANCE_CORNER);

          // if this is the start of the game Consider the sum Weights of the
cells
        double strategicCells = this.countTurns < 15 ?
differenceSumWeights(whiteCount,blackCount) : 0;


        double protectingKing = BoardMoves.protectKing(kingPos,
this.whitesPawns) / (double) COUNT_NEIGHBORS;

        double eval = blackAlive * blackHuristics.get(BLACK_COUNT);

        eval += whiteEaten * blackHuristics.get(WHITE_COUNT);

        eval += winWays * blackHuristics.get(KING_PATHS);

        eval -= strategicCells * blackHuristics.get(CELL_WEIGHTS);

        eval += protectingKing * blackHuristics.get(KING_PROTECTION);

        eval += kingsAdvancement * blackHuristics.get(KING_ADVANCE_CORNERS);
```

```
        // From turn 35 and up, the black needs to be more Aggressive SO
        // get more higher grade one they cause danger to the King
         if (this.countTurns > 35) {
             eval += enemyNearKing * blackHuristics.get(DANGER_KING) * 1.2;
         } else {
             eval += enemyNearKing * blackHuristics.get(DANGER_KING);
         }
         return eval;
     }

     /**
      * The function calculate the Difference between The Sum Strategic
Weights
      * of the white Pawns and the Black Pawns
      * @return the difference of sums weighted between 0 to 1
      */
     private double differenceSumWeights() {
         int blackCount = this.blackPawns.cardinality();
         int whiteCount = this.whitesPawns.cardinality();
         double blackWeights = this.sumWeights(Player.BLACK) / (blackCount *
GamePositions.AVG_WEIGHTS_BLACK);
         double whiteWeights = this.sumWeights(Player.WHITE) / (whiteCount *
GamePositions.AVG_WEIGHTS_WHITE);
         return whiteWeights - blackWeights;
     }

     /**
      * Calculate the sum Of the weights of the places that the Pawns of the
      * player is on
      *
      * @param player The player to Calculate his Pawns board Positions
      * @return The sum of all the weights
      */
     private int sumWeights(Player player) {
         int sum = 0;
         int[] cellsGrade = (player == Player.WHITE) ?
GamePositions.whiteWeights : GamePositions.blackWeights;

         BitSet bitBoard = (player == Player.WHITE) ? this.whitesPawns :
this.blackPawns;

         for (int i = bitBoard.nextSetBit(0); i != -1; i =
bitBoard.nextSetBit(i + 1)) {
             sum += cellsGrade[i];
         }
         return sum;
     }

     /**
      * @return count of free path to king to get to one of the corners Cells
and
      * win the Game
      */
     private int freePathsToKing() {
         List<Action> movesKing =
BoardMoves.getMovesForPawn(this.kingPawn.nextSetBit(0), this);
```

```
        BitSet kingDestinations = new BitSet(BOARD_DIMENSION);

        for (Action move : movesKing) {
            // set bit in Mask
            kingDestinations.set(move.getTo());
        }

        kingDestinations.and(GamePositions.escape);
        // count the number of 1 bits
        return kingDestinations.cardinality();
    }


    private int countOnStratagicForPlayer() {

        if (this.currentPlayer == Player.BLACK) {
            return BitSetHelper.cloneAndResult(GamePositions.strategicBlack,
this.blackPawns).cardinality();
        }
        return BitSetHelper.cloneAndResult(GamePositions.strategicWhite,
this.whitesPawns).cardinality();

    }

    @Override
    public void printState() {
        System.out.println("The board");
        int place = 0;
        char player;
        // loop over all the board cells
        for (int i = 0; i < IState.BOARD_DIMENSION; i++) {
            player = ' ';
            if (this.blackPawns.get(i)) {
                System.out.print(ANSI_RED);
                player = 'B';
            } else {
                System.out.print(ANSI_BLUE);
                if (this.kingPawn.get(i)) {
                    player = 'K';
                } else {
                    if (this.whitesPawns.get(i)) {
                        player = 'W';
                    }
                }
            }
            if (place < 10) {
                System.out.print(" ");
            }
            System.out.print(place + " [" + player + "] ");
            System.out.print(ANSI_RESET);
            if (i % 9 == 8) {
                System.out.println("");
            }
            place++;
        }
        System.out.println("\n\n");
    }
```

קובץ boardMoves

```
public class BoardMoves {

    /**
     * the function build a list with all the possible Moves for the pawn in
     * pawnPosition in the Game state
     *
     * @param pawnPosition positions of the pawn on Tablut board
     * @param state state of the game of tablut
     * @return a list of the possible actions for this specific pawn
     */
    public static List<Action> getMovesForPawn(int pawnPosition, IState
state) {

        // 16 is the maximum amount of moves we can possibly have:
        // it ensures no further allocations are needed
        List<Action> moves = new ArrayList<>(16);

        BitSet forbiddenCells = (BitSet) state.getBoard().clone();

        // The castle cell is forbidden to step on To Every pieces
        forbiddenCells.set(GamePositions.E5.ordinal());

        // Camps may or may not be forbidden to step on
        if (state.getCurrentPlayer() == Player.WHITE) {
            // Whites can never go on the camps
            forbiddenCells.or(GamePositions.camps);
        } else {
            // Blacks can't go back on the camps
            // but if he right now in camp, he can move inside the camp
            if (!GamePositions.camps.get(pawnPosition)) {
                forbiddenCells.or(GamePositions.camps);
            }
        }

        // Check for moves in all directions
        // Up
        for (int cell = pawnPosition - IState.EDGE_SIZE; cell >= 0; cell -=
IState.EDGE_SIZE) {
            // When we find a forbidden cell, we can stop to check in this
direction
            if (forbiddenCells.get(cell)) {
                break;
            }
            moves.add(new Action(pawnPosition, cell));
        }

        // Down
        for (int cell = pawnPosition + IState.EDGE_SIZE; cell <
IState.BOARD_DIMENSION; cell += IState.EDGE_SIZE) {
            // When we find a forbidden cell, we can stop
            if (forbiddenCells.get(cell)) {
                break;
            }
```

```
            moves.add(new Action(pawnPosition, cell));

        }

        // Left: check only if the pawn isn't on column A
        if (pawnPosition % IState.EDGE_SIZE != 0) {

            // Make sure we don't end up out of the board
            // or one row above in column I
            for (int cell = pawnPosition - 1; cell >= 0 && cell %
IState.EDGE_SIZE != IState.EDGE_SIZE - 1; cell--) {

                // When we find a forbidden cell, we can stop
                if (forbiddenCells.get(cell)) {
                    break;
                }

                moves.add(new Action(pawnPosition, cell));

            }

        }

        // Right: check only if the pawn isn't on column I
        if (pawnPosition % IState.EDGE_SIZE != IState.EDGE_SIZE - 1) {

            // Make sure we don't end up out of the board
            // or one row below in column A
            for (int cell = pawnPosition + 1; cell < IState.BOARD_DIMENSION
&& cell % IState.EDGE_SIZE != 0; cell++) {

                // When we find a forbidden cell, we can stop
                if (forbiddenCells.get(cell)) {
                    break;
                }
                moves.add(new Action(pawnPosition, cell));
            }
        }

        return moves;
    }

    /**
     * find all the captures of a move taken in the game
     *
     * @param move last action taken in the state
     * @param state state of the game of tablut
     * @return bit board mask of all the positions of captured pawns taken in
     * move
     */
    public static BitSet getCapturedPawns(Action move, IState state) {

        if (state.getCurrentPlayer() == Player.BLACK) {
            return getCapturesForBlack(move, state);
        }
        return getCapturesForWhite(move, state);
    }
```

```
    private static BitSet getCapturesForWhite(Action move, IState state) {

        BitSet blacks = state.getBlackPawns();
        BitSet whites = (BitSet) state.getWhitePawns().clone();

        // add the king to the whites Bit Set
        whites.or(state.getKing());

        // clear the moving piece from the attacker board
        whites.clear(move.getFrom());

        return getNormalCaptures(move.getTo(), whites, blacks);

    }

    private static BitSet getCapturesForBlack(Action move, IState state) {

        BitSet blacks = (BitSet) state.getBlackPawns().clone();
        // clear the miving piece
        blacks.clear(move.getFrom());

        BitSet king = (BitSet) state.getKing().clone();
        BitSet whites = (BitSet) state.getWhitePawns().clone();

        // move the piece
        blacks.set(move.getTo());

        // Normal captures
        BitSet capturesPawns = getNormalCaptures(move.getTo(), blacks,
whites);

        // Check if the king needs a special capture in four size
        int kingPosition = king.nextSetBit(0);

        if (king.intersects(GamePositions.specailKingCapture)) {
            if (kingPosition == GamePositions.E5.ordinal()) {
                // the king is in the castle, check the four cells Surrounded
him
                if (BitSetHelper.cloneAndResult(blacks,
GamePositions.kingSurrounded).cardinality() == 4) {
                    capturesPawns.set(kingPosition);
                }
            } else {
                BitSet result = new BitSet(IState.BOARD_DIMENSION);
                BitSet enemyMask = null;
                // need three captures....
                switch (kingPosition) {

                    // BitSetPosition.E4.ordinal() = 31
                    case 31:
                        enemyMask = GamePositions.kingE4Surrounded;
                        break;

                    // BitSetPosition.D5.ordinal() = 39
                    case 39:
                        enemyMask = GamePositions.kingD5Surrounded;
```

```
                        break;
                    // BitSetPosition.E6.ordinal() = 49
                    case 49:
                        enemyMask = GamePositions.kingE6Surrounded;
                        break;
                    // GamePositions.F5.ordinal() = 41
                    case 41:
                        enemyMask = GamePositions.kingF5Surrounded;
                        break;
                }
                if (BitSetHelper.cloneAndResult(blacks,
enemyMask).cardinality() == 3) {
                    capturesPawns.set(kingPosition);
                }
            }
        } else {
            BitSet capturesKing = getNormalCaptures(move.getTo(), blacks,
king);
            capturesPawns.or(capturesKing);
        }

        return capturesPawns;
    }

    /**
     * The function find the captures for an action in destination position
and
     * build mask of that captured pawns
     *
     * @param position destination position of an action
     * @param attack attackers bitSet
     * @param defense defenders bitSet
     * @return mask of that captured pawns
     */
    private static BitSet getNormalCaptures(int position, BitSet attack,
BitSet defense) {

        // Bit Board represent all the potensial captured pieces to the
defence
        BitSet captured = new BitSet(IState.BOARD_DIMENSION);

        //Check UP
        // Check if the postion is not in the first 2 rows
        if (position / IState.EDGE_SIZE > 1) {
            int oneUpCell = position - IState.EDGE_SIZE;
            if (defense.get(oneUpCell)) {
                int twoUpCell = oneUpCell - IState.EDGE_SIZE;
                if (attack.get(twoUpCell) ||
GamePositions.obstacles.get(twoUpCell)) {
                    captured.set(oneUpCell);
                }
            }
        }

        //Check DOWN
        // Check if the postion is not in the last 2 rows
```

```
        if (position / IState.EDGE_SIZE < IState.EDGE_SIZE - 1) {
            int oneDownCell = position + IState.EDGE_SIZE;
            if (defense.get(oneDownCell)) {
                int twoDownCell = oneDownCell + IState.EDGE_SIZE;
                if (attack.get(twoDownCell) ||
GamePositions.obstacles.get(twoDownCell)) {
                    captured.set(oneDownCell);
                }
            }
        }

        //Check LEFT
        // Check if the postion is not in the left 2 columns
        if (position % IState.EDGE_SIZE > 1) {

            int oneLeftCell = position - 1;
            if (defense.get(oneLeftCell)) {

                int twoLeftCell = oneLeftCell - 1;
                if (attack.get(twoLeftCell) ||
GamePositions.obstacles.get(twoLeftCell)) {
                    captured.set(oneLeftCell);
                }

            }
        }

        //Check RIGHT
        // Check if the postion is not in the right 2 columns
        if (position % IState.EDGE_SIZE < IState.EDGE_SIZE - 2) {

            int oneRightCell = position + 1;
            if (defense.get(oneRightCell)) {

                int twoRightCell = oneRightCell + 1;
                if (attack.get(twoRightCell) ||
GamePositions.obstacles.get(twoRightCell)) {
                    captured.set(oneRightCell);
                }

            }
        }

        return captured;
    }

    /**
     * Find the protection level of the king
     *
     * @param kingPos king bit position on the bit Board
     * @param blacks black bit board
     * @return count of black / obstacles cells around the king
     */
    public static int dangerToKing(int kingPos, BitSet blacks) {

        int threat = 0;
```

```
        if (kingPos % IState.EDGE_SIZE != 0 && (blacks.get(kingPos - 1) ||
GamePositions.obstacles.get(kingPos - 1))) {
            threat++;
        }

        if (kingPos % IState.EDGE_SIZE != IState.EDGE_SIZE - 1 &&
(blacks.get(kingPos + 1) || GamePositions.obstacles.get(kingPos + 1))) {
            threat++;
        }

        if (kingPos > IState.EDGE_SIZE && (blacks.get(kingPos -
IState.EDGE_SIZE) || GamePositions.obstacles.get(kingPos -
IState.EDGE_SIZE))) {
            threat++;
        }

        if (kingPos < 72 && (blacks.get(kingPos + IState.EDGE_SIZE) ||
GamePositions.obstacles.get(kingPos + IState.EDGE_SIZE))) {
            threat++;
        }

        return threat;

    }

    /**
     * Find how many white Pawns circles Around the king in order to keep him
     * safe
     *
     * @param whites whites bit Board
     * @param kingPos king bit position on the bit Board
     * @return count of whites Pawns around the king
     */
    public static int protectKing(int kingPos, BitSet whites) {

        int protects = 0;

        if (kingPos % IState.EDGE_SIZE != 0 && whites.get(kingPos - 1)) {
            protects++;
        }

        if (kingPos % IState.EDGE_SIZE != IState.EDGE_SIZE - 1 &&
whites.get(kingPos + 1)) {
            protects++;
        }

        if (kingPos > IState.EDGE_SIZE && whites.get(kingPos -
IState.EDGE_SIZE)) {
            protects++;
        }

        if (kingPos < 72 && whites.get(kingPos + IState.EDGE_SIZE)) {
            protects++;
        }

        return protects;
```

```
    }

    public static int pawnsToEatKing(BitSet kingBoard) {
        // castle
        if (kingBoard.get(GamePositions.CENTER_POSITION)) {
            return 4;
        }
        // near castle
        if (GamePositions.specailKingCapture.intersects(kingBoard)) {
            return 3;
        }
        // regular eating
        return 2;
    }

    /**
     * Find the minimum manhattan Distance for the king to one of the free
     * escape cells
     *
     * @param xPos x Position of the king on the board
     * @param yPos y Position of the king on the board
     * @param fullBoard bit Set Of all the pieces on the board
     * @return the minimum manhattan Distance found from the king to escapes
     */
    public static int minDistanceToCorner(int xKing, int yKing, BitSet
fullBoard) {

        int minDistance = GamePositions.MAX_DISTANCE_CORNER;
        int manhattanDistance;
        for (int escapeCellPosition : GamePositions.escapeCells) {

            // check if the escape cell is empty
            if (fullBoard.get(escapeCellPosition) == false) {

                int xCellEscape = escapeCellPosition % IState.EDGE_SIZE;
                int yCellEscape = escapeCellPosition / IState.EDGE_SIZE;

                manhattanDistance = Math.abs(yKing - yCellEscape) +
Math.abs(xKing - xCellEscape);

                minDistance = Math.min(minDistance, manhattanDistance);

                // the minimum distance can be One so if we got to So end the
loop
                if (minDistance == 1) {
                    return 1;
                }

            }
        }
        return minDistance;
    }

}
```

קובץ Controller

```
public class Controller implements IController {

    private IState state;
    private IView view;

    /**
     * Controller constructor
     *
     * @param gameView get object of IView and build the controller and the
     * model Layers
     */
    public Controller(IView gameView) {
        this.view = gameView;
        this.state = new TablutState();
        this.view.setBoard(state);
    }

    /**
     * The function reset The game of Tablut by reset the TablutState to the
     * starting board
     */
    @Override
    public void resetGame() {
        this.state = new TablutState();
    }

    /**
     * the function perform human action if the game is against AI perform
also
     * the Computer's Turn
     *
     * @param turnFrom piece index to move
     * @param turnTo destination to move the piece in turnFrom place
     * @param isAI boolean flag that say if the game is against computer
     */
    @Override
    public void makeTurn(int turnFrom, int turnTo, boolean isAI) {

        // build the action that the user asked
        Action desiredAction = new Action(turnFrom, turnTo);

        // perform human move
        doMove(desiredAction);

        boolean isGameOver = isEnd();
        // check if the Game Over
        if (!isGameOver) {
            // if the Game is not over and we play
            // against Computer Call to AI function to do turn
            if (isAI) {
                playAI();
            }
        }
```

```
    }

    /**
     * the function perform Action and updates the view about the move
results
     *
     * @param move move to perform on the board
     */
    private void doMove(Action move) {
        CapturedPawns captures = state.performAction(move);

        // update the view about the changes on the board
        this.view.updateGameDetails(captures.getCaptured(), move);
    }

    /**
     * The function check if the game is Over and update the view accordingly
     *
     * @return True if the Game is Over else return False
     */
    public boolean isEnd() {

        if (state.isGameOver()) {
            if (state.isDraw()) {
                view.printDraw();
            } else {
                view.printWIN(state.getEnemy());
            }
            return true;
        }
        return false;
    }

    /**
     * the function find the possible moves for Piece in order to mark them
in
     * the Board
     *
     * @param piecePos piece position on the board
     * @return list of possible moves to the piece At piecePos
     */
    @Override
    public List<Integer> getMovesToView(int piecePos) {
        List<Action> moves = BoardMoves.getMovesForPawn(piecePos, state);
        return moves.stream().map(urEntity ->
urEntity.getTo()).collect(Collectors.toList());
    }

    /**
     * the function call to the artificial intelligence Algorithm and perform
     * the optimal move for the computer
     */
    @Override
    public void playAI() {
        AiBrain brain = new AiBrain(this.state);

        Action turn = brain.bestMove();
```

```
        // perform Computer Move
        doMove(turn);
        // check if the Computer Won
        isEnd();
    }

    @Override
    public boolean isPieceInPos(int piecePos) {
        return this.state.isPlayerPieceInPos(piecePos);
    }

    @Override
    public int getCountTurns() {
        return state.getCountTurns();
    }
}
```

קובץ AiBrain

```
public class AiBrain {

    private IState boardGame;
    private static final int DEPTH_SEARCH = 5;

    public AiBrain(IState boardGame) {
        this.boardGame = boardGame;
    }

    /**
     * The function call to NegaMax function and find the best Move for the
     * current player
     *
     * @return Action with the positions of the best move founded in the
search
     */
    public Action bestMove() {
        Action optimalAction;
        long start = System.currentTimeMillis();

        optimalAction = negaMax(DEPTH_SEARCH, Integer.MIN_VALUE,
Integer.MAX_VALUE);

        long finish = System.currentTimeMillis();
        System.out.println("Time to Choose Move: " + (finish - start) /
(double) 1000 + " Seconds! ");
        System.out.println("\t" + optimalAction);
        return optimalAction;
    }

    /**
     * the function use negaMax algorithm with alpha beta pruning to choose
the
     * optimal move for the current player in this.boardGame
     *
     * @param depth depth of the search in the Game tree
```

```java
 * @param alpha The best highest-value choice we have found so far
 * @param beta The lowest-value choice we have found so far
 * @return the best Action for the current player founded
 */

private Action negaMax(int depth, double alpha, double beta) {

    if (boardGame.isGameOver() || depth == 0) {
        // if we get to leaf or max search depth
        return new Action(boardGame.getHuristic());
    }

    // get all possible moves
    List<Action> possibleMoves = boardGame.getAvailablePawnMoves();

    moveOrdering(possibleMoves);

    Action bestMove = new Action();

    for (Action move : possibleMoves) {

        // get to captured pawns from the move
        // in order to "undo" the move after searching
        CapturedPawns captures = boardGame.performAction(move);

        // call the function recursively with less depth
        Action childBestMove = negaMax(depth - 1, -beta, -alpha);

        // use the NegaMax principle that max(a, b) = -min(-a, -b)
        childBestMove.negateGrade();

        double childGrade = childBestMove.getGrade();

        if (childGrade > bestMove.getGrade()) {
            bestMove = move;
            bestMove.setGrade(childGrade);
        }

        // Undo move
        this.boardGame.cancelMove(captures, move);

        // check if we can "cut" parts of the game tree
        alpha = Math.max(alpha, bestMove.getGrade());

        if (alpha >= beta) {
            break;
        }

    }
    return bestMove;
}
```

```
    /**
     * The function responsible for sort the list of possible moves in way
that
     * will cause more "cutting" of the game tree will be explained in the
     * article
     *
     * @param moves the list of possible moves in node of the game tree
     */
    private void moveOrdering(List<Action> moves) {
        // if the player is black
        // sort by priority:
        // 1. if the destination of the action is near the King
        // 2. if the origin Is not in camp Cells
        // 3. all other actions

        // for white Player:
        // the king moves will be calculated first

        if (this.boardGame.getCurrentPlayer() == Player.BLACK) {

            int kingPos = this.boardGame.getKing().nextSetBit(0);

            moves.sort(new Comparator<Action>() {
                // sort in descending order
                @Override
                public int compare(Action m1, Action m2) {
                    return m2.actionBlackValue(kingPos) -
m1.actionBlackValue(kingPos);
                }

            });

        }
    }
}
```

קובץ IState

```
public interface IState {
    final int MAX_TURNS = 100;
    final int EDGE_SIZE = 9;
    final int BOARD_DIMENSION = EDGE_SIZE * EDGE_SIZE;
    public int getCountTurns();
    public boolean isPlayerPieceInPos(int bitPos);
    public void printState();
    public Player getEnemy();
    Player getCurrentPlayer();
    void changeCurrentPlayer();
    BitSet getBlackPawns();
    BitSet getWhitePawns();
    public void undoCaptures(CapturedPawns captures);
    public void undoAction(Action action);
    public CapturedPawns performAction(Action action);
    BitSet getKing();
    BitSet getBoard();
    boolean isWinningState();
    boolean blackHasWon();
    boolean whiteHasWon();
    List<Action> getAvailablePawnMoves();
    List<Action> getAvailableKingMoves();
    public IState clone();
    public boolean isGameOver();
    public double getHuristic();
    public void cancelMove(CapturedPawns Captures, Action move);
    public boolean isDraw();}
```

קובץ IView

```
/**
 * Interface for the View part of the application the view include Any
 * representation of information such as a chart, diagram or table of the
 * application
 *
 * @author Ofir
 */
public interface IView {

    // update that we have a draw
    public void printDraw();

    // update who is the player that Won
    public void printWIN(Player player);

    public void updateGameDetails(BitSet captures, Action move);

    // will be deleted
    public void printGameState();

    public void setBoard(IState state);

}
```

קובץ IController

```
public interface IController {

    public List<Integer> getMovesToView(int piecePos);

    public void makeTurn(int turnFrom, int turnTo, boolean isAI);

    public boolean isPieceInPos(int piecePos);

    public void resetGame();

    public int getCountTurns();

    public void playAI();
}
```

סיכום אישי – רפלקציה

כתיבת פרויקט זה תרמה לי רבות במובנים רבים. ראשית, הצלחתי להתמודד עם אתגר שהיה נראה בתחילת הדרך קשה ומסובך ואני שמח על כך. כמו כן, תהליך הכתיבה לימד אותי איך להתמודד עם באגים ותקלות בדרך ולא להתייאש עד שאני מקבל תוצר סופי שאני גאה בו.

הכלים המרכזיים שאני לוקח להמשך היא הרצינות וההתמדה.

המשחק שבחרתי אינו היה מוכר לי תחילה. על מנת להשתפר במשחק וללמוד אסטרטגיות שונות נאלצתי לשחק באפליקציה בטלפון שלי משחקים רבים עד שהצלחתי לנצח שחקנים אחרים במשחק, דבר שעזר לי לבנות את האלגוריתם היעיל והמורכב שבניתי.

במהלך הדרך, נאלצתי לחוות קונפליקטים רבים והמרכזיים שבהם:

1. בחירת מבנה הנתונים אשר ייצג את לוח המשחק.
2. איך לתת משקלים לגורמים השונים בפונקציית הערכה.

צלחתי את האתגריים הללו באמצעות חקר מאמרים בספרות, ניסוי והסקת מסקנות אישיים והדרכה ע"י המרצים שלי שעזרו לי להסתכל על הדברים בצורה בוגרת ושקולה.

תחום הבינה המלאכותית קסם לי והמסקנה העיקרית שלי מן הפרויקט שאני אשמח לעסוק בתחום זה בהמשך דרכי אם זה בצבא או במרחב האזרחי. הסיבה לכך היא שאינך יכול לדעת תמיד למה לצפות ואני הופתעתי בכל פעם מחדש ממהלכי המחשב למרות שבסך הכל אני הייתי אחראי עליהם.

אם הייתי מתחיל את הפרויקט היום הייתי בונה את הממשק הגרפי בסביבת android מכיוון שאני וחברי הקרובים התחלנו לשחק בתוכנה שבניתי בפרויקט והייתי רוצה להוציא גרסה בנויה של משחק זה לשוק האפליקציות על מנת שאוכל לשחק איתם online. כמו כן, הייתי שמח אם היה לי זמן לחקור עוד אלגוריתמים של בינה ולהשוות בין אלגוריתמים שונים בתחום המשחק.

ישנם עוד שאלות חקר רבות אשר מסקרנות אותי בפיתוח אלגוריתם למשחק Tablut.

לדוגמא: השפעת מספר הכלים השחורים במשבצות המחנה בזמן סיום המשחק, ומתי זה הוא הזמן האופטימלי עבור השחקן הלבן לנוע עם כלי המלך ממשבצת הטירה. כל תהיות אלו ישארו איתי גם לאחר כתיבת הפרויקט ואני מקווה למצוא להם תשובה בעתיד עם הניסיון שארכוש ופיתוח החשיבה האסטרטגית שלי.

לסיום אני רוצה לציין את ההדרכה הצמודה והקשובה של מוריי לאורך כל הדרך. מיכאל ואלון היו שם בשבילי לכל שאלה ותמיכה. מיכאל עזר לי בבחירת מבני הנתונים ואלון שלימד אותי Java בצורה הכי מקיפה ומקצועית שיש.

ביבליוגרפיה

Ashton, J.C

Linnaeus's game of tablut and its relationship to the ancient viking game hnefatafl

The Heroic Age: A Journal of Early Medieval Northwestern Europe 13, 1526–1867 (2010), https://www.heroicage.org/ issues/13/ashton.php

Chess and Bitboards 2009

Physical Location Bitboards
http://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/index.html

Chess programming Wiki

Nega Max, https://www.chessprogramming.org/Negamax

Damian Gareth Walker

An Introduction to Hnefatafl 2008

Hnefatafl Modern

video series about the ancient Norse game of "Tablut"

10/09/2011

https://www.youtube.com/watch?v=eYjqq9kPHPU

Philip Hingston

Evolving Players for an Ancient Game: Hnefatafl
Computational Intelligence and Games, 2007
https://www.researchgate.net/publication/4249890_Evolving_Players_for_an_Ancient_Game_Hnefatafl