עבודת גמר

לקבלת תואר

טכנאי תוכנה

הנושא: משחק הדמקה

המגיש : אמיר וולברג

ת.ז. המגיש: 212939631

שמות המנחים : אלון חיימוביץ' ומיכאל צ'רנלובסקי

אפריל 2021 תשפ"א

תוכן עניינים

Contents

אמיר וולברג
משחק הדמקה

תקציר

מטרת הפרויקט בפן האישי

מטרת הפרויקט עבורי היא להעשיר את הידע שלי בנושא הבינה המלאכותית ולחקור דרכים ליעל את סיבוכיות זמן הריצה והמקום של משחק מחשב כמה שיותר.

מבוא

קודם כל הייתי צריך לבחור משחק לפרויקט, חיפשתי משחק יחסית פשוט שאוכל להתמקד פחות על המשחק ויותר על ייעולו ועל הוספת אופציות להתאמה אישית (בייחוד אפשרויות להתאים אישית את הבוט) והגעתי למשחק הדמקה.

בנוסף להיותו של משחק הדמקה משחק יחסית פשוט הוא גם משחק ישן עם הרבה סוגים שונים של חוקים מה שהקשה קצת על להחליט באיזו וריאציה של דמקה להשתמש, בסוף החלטתי להשתמש בוריאציה של דמקה שלא מחייבת לאכול חתיכות של היריב (על מנת להרחיב את חופש הבחירה של הבוט והשחקן ולאפשר יותר סוגי מהלכים בכל תור (לא להקל מדי על הבוט)) ושבה חתיכה שמגיעה לקצה הלוח בצד היריב הופכת למלך אשר יכול לזוז גם אחורה וגם קדימה (ולא רק קדימה כמו חתיכה רגילה).

יתרון נוסף בבחירה במשחק הדמקה היא שישנן הרבה אסטרטגיות מוכחות עבורו והרבה חומר באינטרנט על בניית היוריסטיקות חכמות לבוט שמשחק דמקה.

לסיכום הפרויקט אשר בחרתי לעשות הוא משחק הדמקה עם בינה מלאכותית, בפרויקט הוספתי אפשרויות למשחק שחקן נגד שחקן, שחקן נגד בוט ובוט נגד בוט.

תיאור הפרויקט

דמקה נגד המחשב

השחקן משחק נגד AI במשחק דמקה. (יש אפשרות גם לשחקן נגד שחקן וAI נגד AI).

משחק הדמקה

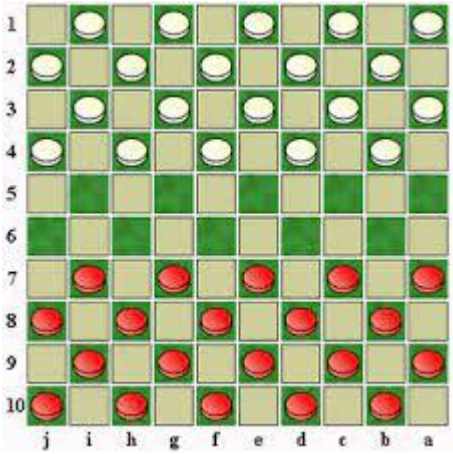
הלוח מונח כך שהמשבצת הימנית ביותר בשורה הקרובה לכל שחקן תהיה לבנה (כמו במשחק השחמט), האבנים מונחות על המשבצות השחורות של הלוח וההתקדמות היא רק על משבצות אלה באלכסון(אסור לזוז אחורה אלא אם כן יש לך מלכה).

בתחילת המשחק אבני השחקן האחד מונחות בשלוש (ארבע, בגרסה הגדולה) השורות הראשונות של הלוח ואילו אבני השחקן השני מונחות באותו אופן בצד שלו. לפי המוסכם, הלבן מבצע את המהלך הראשון והשחור משיב עליו. שתי הפעולות גם יחד נחשבות כמסע אחד.

אפשר לאכול אבן יריב כאשר יש משבצת שחורה ריקה מאחוריה והאבן שלך משבצת באלכסון ממנה (לא ניתן לאכול אחורה עם חייל רגיל).

אפשר לאכול כמה אבני יריב אם יש בין כל אחת מהן מרווח של משבצת שחורה אחת ריקה באלכסון.

אם אבן מגיעה לסוף הלוח היא הופכת למלכה ויכולה לזוז אחורה.



המנצח הוא האחד שאכל את כל אבני היריב.

אם במשך 40 מהלכים רצופים לא השתנה מספר הכלים על הלוח מוכרז תיקו.

רקע תיאורטי בתחום הפרויקט

משחק אסטרטגיה הוא משחק שעל מנת לנצח בו יש צורך בקביעת תוכנית פעולה , דהיינו אסטרטגיה.

משחק מחשב הוא תוכנה המהווה משחק, ומקיימת אינטראקציה עם השחקן, המשחק נשלט באמצעות קלט כמו לחיצות המקלדת או העכבר. במשחקי מחשב ניתן לשחק מול שחקנים אחרים או מול bot דמוי שחקן אשר נשלט על ידי המחשב עצמו.

ממשק משתמש(UI) הוא החלק המוצג למשתמש בתוכנה

והדרך של המשתמש ליצור קשר עם התוכנה או האלגוריתם. הגרסה הנפוצה היום לממשק משתמש נקראת ממשק משתמש גרפי(GUI).

הממשק בדרך כלל מכיל בתוכו יישומונים (widgets) המאפשרים למשתמש לתקשר עם התוכנה באמצעות העכבר, המקלדת , השלט או כל מכשיר קלט מותאם אחר .

בינה מלאכותית היא ענף של מדעי המחשב העוסק ביכולת לתכנת מחשבים לפעול באופן המציג יכולות המאפיינות את הבינה האנושית.

הגדרת הבעיה האלגוריתמית

<u>בניית בינה מלאכותית</u>

בניית בינה מלאכותית שתשחק דמקה נגד המשתמש.

<u>בניית משחק דמקה יעיל</u>

בניית משחק דמקה עם מבנה נתונים יעיל בזיכרון ובזמן ריצה.

<u>חשיבה מהירה של הבינה המלאכותית</u>

מציאת אלגוריתם בינה מלאכותית מהיר בזמן ריצה בכדי שהמשתמש לא יצטרך לחכות יותר מכמה שניות לכל היותר כשהבינה המלאכותית עושה מהלך

<u>הערכת מצב משחק עם אסטרטגיה מסוימת</u>

מציאת אסטרטגיה להערכת מצב לוח של דמקה בצורה שתיתן תמונה כמה שיותר מדויקת על כמה טוב מצב זה עבור שחקן מסוים

סקירת אלגוריתמים בתחום הבעיה

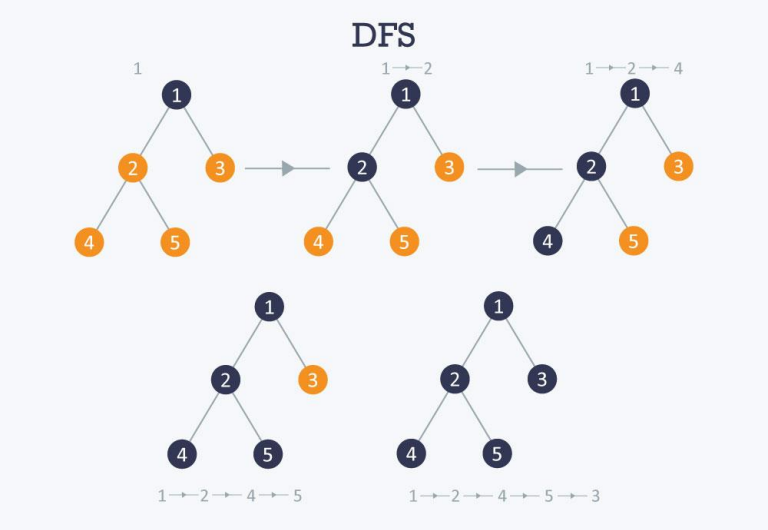
טבלת טרנספוזיציה

טבלת טרנספוזיציה (Transposition table) היא מטמון של עמדות שנראו בעבר והערכות נלוות, בעץ משחקים שנוצר על ידי תוכנית משחקי מחשב. אם עמדה חוזרת על עצמה באמצעות רצף אחר של מהלכים, ערך המיקום נשלף מהטבלה, ונמנע מחיפוש חוזר בעץ המשחק שמתחת למיקום זה. **טבלת הטרנספוזיציה** לא רק משפרת את זמן הריצה של עץ המשחק אלא היא גם משפרת את הביצועים שלו, במקרים מסוימים כשהעץ מעריך מצב לוח מסוים שהוא הגיע אליו בעומק A והוא רואה שיש הערכה למצב הלוח הזה ומהלך הכי טוב עבורו בעומק B הוא בעצם חוזה קדימה עומק של B+A עבור הענף הזה של עץ המשחק, העומק של B+A יכול במקרה הטוב ביותר להיות כפול מעומק העץ שנקבע מראש ובמקרה הרע שווה לעומק העץ שנקבע מראש ובכך מתאפשר לעץ המשחק להסתכל לפעמים יותר מהלכים קדימה מאשר שהוגדר לו מראש.

אלגוריתם חיפוש לעומק

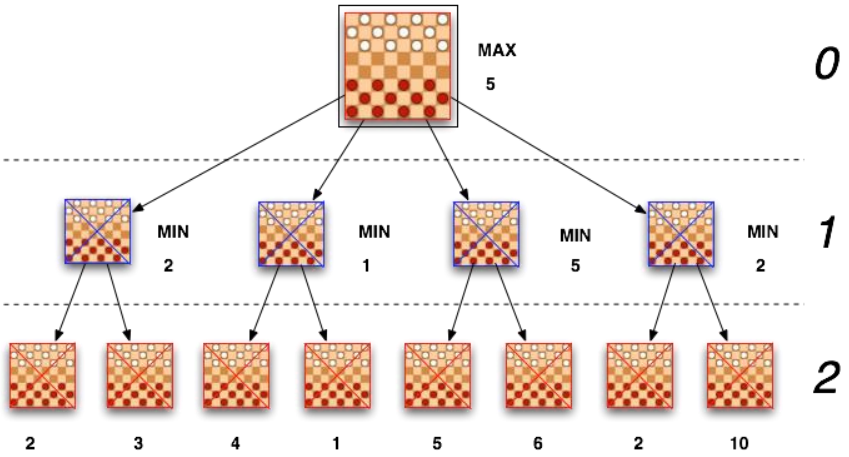
אלגוריתם חיפוש לעומק (DFS) הוא אלגוריתם המשמש למעבר על גרף או לחיפוש בו.

אינטואיטיבית, האלגוריתם מתחיל את החיפוש מצומת שרירותי בגרף ומתקדם לאורך הגרף עד אשר הוא נתקע, לאחר מכן הוא חוזר על עקבותיו עד שהוא יכול לבחור להתקדם לצומת אליו טרם הגיע. דרך פעולת האלגוריתם דומה במידת מה לסריקה שיטתית של מבוך.



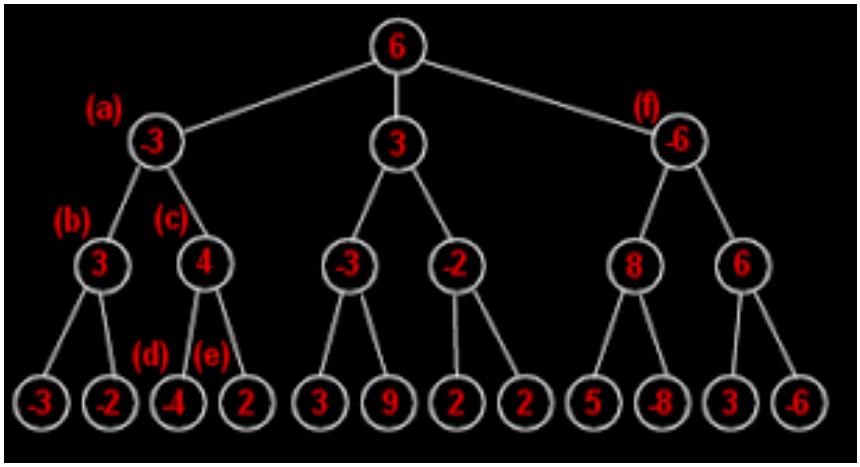
אלגוריתם מינימקס

אלגוריתם מינימקס (Minimax algorithm) הוא אלגוריתם המאפשר לבוט למצוא את המהלך הטוב ביותר עבורו. האלגוריתם דואג למצוא בכל שלב במשחק את המהלך שמקבל את הציון המקסימלי (בתור של הבוט) ואת הציון המינימלי (בתור היריב), כך ניתן למצוא את המהלך עם הציון הטוב ביותר ביחס למהלכים הכי טובים שהיריב יכול לבצע. בכל תור שיתבצע (כעומק העץ שהגדרנו מראש) נדאג לבחור כל פעם ציון אחר ביחס לשחקן הנוכחי.



אלגוריתם נגמקס

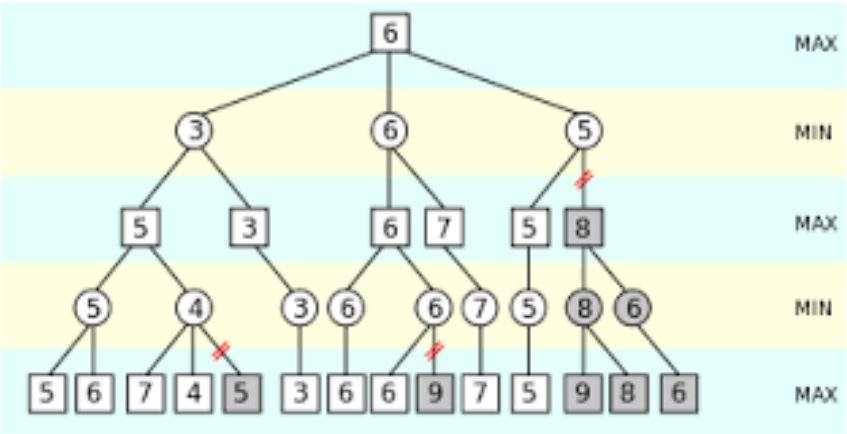
נגזרת של אלגוריתם מינימקס, **אלגוריתם נגמקס** (NegaMax algorithm) נעזר בעובדה של max(a,b) = -min(-a,-b) בכדיי לפשט את האימפלמנטציה של אלגוריתם מינימקס, מזה אפשר להבין שהערך של מצב לוח לשחקן א במשחק שכזה הוא השלילה של ערך הלוח של שחקן ב במשחק כזה. ולכן השחקן שעושה מהלך מחפש מהלך שהוא בעל הערך המקסימלי של השלילה של הערך שנובע ממהלך שלו, המצב לוח לאחר המהלך שלו צריך לפי הגדרה להיות מוערך על ידי היריב של השחקן, ההסבר הזה עובד גם אם שחקן א וגם אם שחקן ב הם אלה שעושים את המהלך, בזכות עובדות אלו ניתן לקצר את הקוד של אלגוריתם המינימקס בהרבה והקוד שמתקבל כתוצאה ניקרא נגמקס.



גיזום אלפא בטא

גיזום אלפא בטא (Alpha beta pruning) הוא דרך ליעל אלגוריתם מינימקס/נגמקס על ידי גיזום חלק מענפיו וקיצור זמן הריצה באמצעות חסם עליון וחסם תחתון

הגיזום פועל בצורה הבאה: במהלך החיפוש לעומק ניתן לזנוח פתרונות חלקיים ברגע שברור שהם גרועים מפתרונות שכבר ראינו.



אלגוריתם לחיפוש לעומק איטרטיבי מעמיק

חיפוש לעומק איטרטיבי מעמיק (Iterative deepening depth-first search)הוא אלגוריתם חיפוש בגרף שנועד לאפשר חיפוש לרוחב בדומה לאלגוריתם חיפוש לרוחב ,אך ללא דרישות הזיכרון הגבוהות שלו, על ידי שימוש חוזר במתודולוגיית אלגוריתם חיפוש לעומק לעומק גדל והולך בכל איטרציה.

האלגוריתם עובד באיטרציות בכל איטרציה i מריצים את אלגוריתם חיפוש לעומק שיסרוק את כל העץ עד לעומק ,i אם הפתרון נמצא הוא יחזיר אותו, אם לא נשארה עוד רמה לפתח הוא יחזיר שאין פתרון, אחרת הוא עובר לאיטרציה.i+1

האלגוריתם מבטיח שאם הפתרון בעומק המינימלי נמצא בעומק k הוא יימצא באיטרציה k מאחר שעד לאיטרציה זו האלגוריתם לא סורק בעומקים k ומעלה ובעומק k הצומת עם הפתרון תיסרק ותוחזר. סיבוכיות הזיכרון באיטרציה i ליניארית ב i-כי מריצים חיפוש לעומק עד עומק .i סיבוכיות הזמן פולינומית בכמות הצמתים עד עומק.k

מושגים

עץ חיפוש

עץ חיפוש (Search tree) הוא מקרה פרטי של גרף. מבנה נתונים המאפשר לנו לשמור בכל צומת של העץ ערך מסוים ולחבר אליו ערכים אחרים שיהיו הילדים שלו, עץ חיפוש יכול להיות עם כמות בלתי מוגבלת של ילדים לכל צומת אך יש לו שורש אחד ויחיד שממנו לרוב נתחיל את החיפוש.

עץ משחק

עץ משחק (Game tree) 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.

היוריסטיקה

היוריסטיקה היא כלי חשיבה המבוסס על הגיון פשוט או אינטואיציה ,המציע דרך קלה ומהירה לקבלת החלטות ופתרון בעיות ,ללא התעמקות ובמחיר דיוק נמוך.

לוח ביטים

לוח ביטים (Bit board) הוא מספר בינארי שהביטים שלו מייצגים את לוח המשחק ועשים עליו פעולות לוגיות בשביל להזיז את הביטים (הפעולות הלוגיות מאוד יעילות מבחינת זמן ריצה), באמצעות לוח ביטים ניתן לייצג לוח משחק בצורה יעילה מבחינת זיכרון וזמן ריצה.

טבלת טרנספוזיציה

טבלת טרנספוזיציה (Transposition table) היא מטמון של עמדות שנראו בעבר והערכות נלוות, בעץ משחקים שנוצר על ידי תוכנית משחקי מחשב. אם עמדה חוזרת על עצמה באמצעות רצף אחר של מהלכים, ערך המיקום נשלף מהטבלה, ונמנע מחיפוש חוזר בעץ המשחק שמתחת למיקום זה.

אסטרטגיה

למשחק בחרתי להשתמש בלוח ביטים בשביל שיהיה יעיל בזיכרון ובזמן הריצה.

לבינה בחרתי לעשות עץ משחק עם אלגוריתם נגמקס וגיזום אלפא בטא בנוסף הוספתי טבלת טרנספוזיציה שמתמלאת עם אלגוריתם לחיפוש לעומק איטרטיבי מעמיק על מנת לשפר עוד את זמן הריצה של האלגוריתם.

האסטרטגיה שאני משתמש בה בפונקציות היוריסטיות שמעריכות את מצב הלוח עבור השחקן הנוכחי בעץ המשחק כוללות את הפרמטרים הבאים:

1. מספר החיילים של השחקן
2. מספר המלכים של השחקן
3. מצב משחק (ניצחון/הפסד/תיקו)
4. מספר מהלכים אפשריים שיכול השחקן לבצע
5. מספר חתיכות שמגינות על השורה הראשונה של השחקן (השורה שאם מגיע אליה היריב עם חייל הוא מוכתר למלך)
6. מספר החיילים הבטוחים (שנמצאים סמוך לקצה הלוח ולכן לא ניתנים לאכילה)
7. מספר המלכים הבטוחים (שנמצאים סמוך לקצה הלוח ולכן לא ניתנים לאכילה)

הערך שמחזירה הפונקציה היוריסטית (Score) מחושב עם כל הפרמטרים בנוסחה הבאה

$$Score = \sum_{P=1}^{N}\big(P(Player) - P(Enemy)\big) \times PWeight$$

כאשר

P = אחד הפרמטרים

N = מספר הפרמטרים שהפונקציה היוריסטית משתמשת בהם

Player = השחקן הנוכחי שעבורו מנוקד הלוח

Enemy = השחקן היריב

PWeight = משקל הפרמטר הנוכחי

Score = התוצאה שמחזירה הפונקציה היוריסטית שמנקדת את הלוח לפי כמה המצב טוב לשחקן הנתון

ישנן 3 פונקציות היוריסטיות שאני משתמש בהן כל אחת משלבת חלק או את כל הפרמטרים המוזכרים למעלה בצורה מסוימת, להלן הסבר על כל אחת מהפונקציות:

הפונקציה היוריסטית הבסיסית (BasicHeuristic) היא פונקציה היוריסטית המשלבת את הפרמטרים 1, 2, 3 ו4 שהוזכרו למעלה ומציבה אותם בנוסחה שהוזכרה למעלה כאשר נבחרת פונקציה זו יכול המשתמש לתת איזה משקל שהוא רוצה לכל אחד מהפרמטרים שיש בה.

הפונקציה היוריסטית הקיצונית (ExtremeHeuristic) היא פונקציה היוריסטית שמשלבת את כל הפרמטרים 1, 2, 3, 4, 5, 6 ו7 שהוזכרו למעלה ומציבה אותם בנוסחה שהוזכרה למעלה והמשקלים בה הוחלטו מראש ולא ניתנים לשינוי על ידי המשתמש.

הפונקציה היוריסטית המותאמת (AdaptiveHeuristic) היא פונקציה היוריסטית המשלבת את הפרמטרים 1, 2, 3 ו4 שהוזכרו למעלה ומציבה את פרמטרים 3 ו4 כרגיל בנוסחה שהוזכרה למעלה והמשקלים בה הוחלטו מראש ולא ניתנים לשינוי על ידי המשתמש.
פונקציה זו נותנת משקל שונה לחיילים והמלכים במהלך המשחק, ככל שיש פחות חיילים או מלכים ככה משקל כל אחד מהם גדל. לחישוב משקל כל חתיכה על הלוח מתווסף למשקל החתיכה הרגיל עוד פרמטר שמחושב לפי מספר החתיכות שיש לשחקן עם הנוסחה הבאה

$$\text{WeightOfPiece} = \frac{12}{(\text{NumberOfSoliders} + \text{NumberOfKings} + 1)}$$

כאשר

NumberOfSoliders = מספר החיילים של השחקן

NumberOfKings = מספר המלכים של השחקן

WeightOfPiece = פרמטר שקובע את המשקל של כל חתיכה של השחקן על הלוח

ובכדי למצוא את הניקוד הניתן על החיילים והמלכים מספר החתיכות מוכפל במשקל כל חתיכה (שנמצא באמצעות המשוואה הקודמת) כך

$$ScoreOfSoliders = NumberOfSoliders \times WeightOfPiece$$

$$ScoreOfKings = NumberOfKings \times WeightOfPiece$$

3 נוסחאות אלו מחושבות עבור השחקן ועבור היריב שלו מוצבות בנוסחה המוזכרת למעלה בתור פרמטרים ומוכפלות במשקל הבסיסי הנתון למספר חיילים ומספר מלכים יחד עם פרמטרים 3 ו4.

מבנה נתונים

לוח ביטים

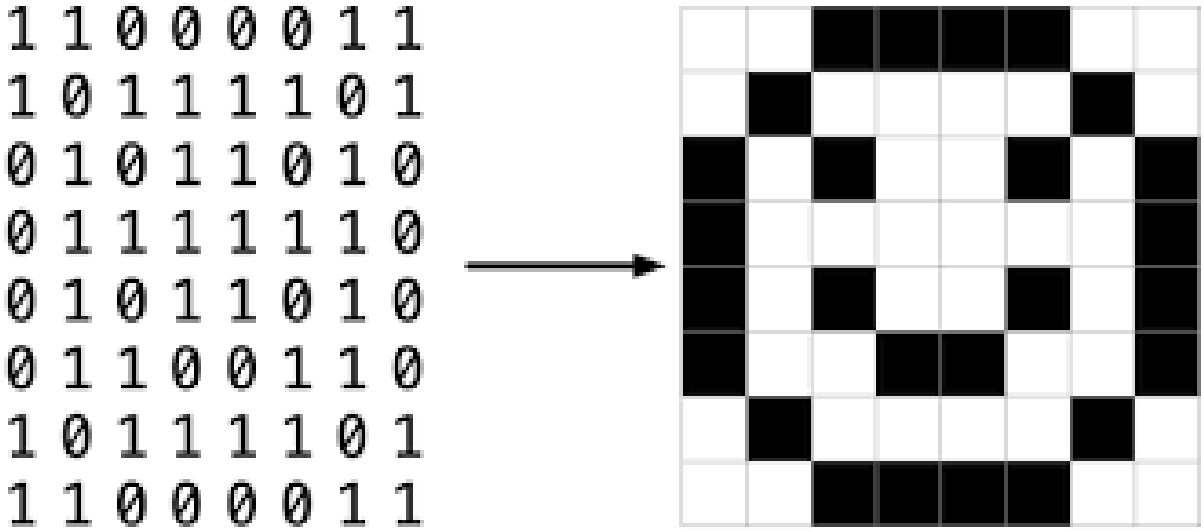
אני משתמש במבנה נתונים מסוג **לוח ביטים** (Bit board) בצורה הבאה אני משתמש ב3 UInt32 לייצוג לוח המשחק, 32 ביטים לייצוג 32 המשבצות במשחק שיכולות להיות עליהן חתיכות.

כל UInt32 מייצג לוח עם סוג חתיכות שונה, בראשון כל סיבית דולקת מייצגת חתיכה שחורה, בשני כל סיבית דולקת מייצגת חתיכה לבנה ובשלישי כל סיבית דולקת מייצגת מלך על הלוח (שחור או לבן). בכדי למצוא את כל המלכים הלבנים או השחורים עושים פעולה לוגית AND ללוח של הצבע הרצוי יחד עם לוח המלכים.

בכדי לבצע מהלך אני משתמש בחתיכה שנבחרה על ידי המשתמש והומרה למסכת ביטים מסוג UInt32 שבה הסיבית הדולקת מייצגת את מיקום החתיכה על הלוח ושולח אותה לפונקציה שמחזירה את כל המיקומים שאליהם יכולה החתיכה לזוז באמצעות בדיקה עם לוחות הביטים ופעולת הזזת סיביות ימינה ושמאלה בתור רשימה של Struct שיצרתי בשם PieceMove שבו מסכת ביטים המייצגת את המיקום שאליו החתיכה יכולה לזוז, ומשתנה בוליאני האם בצעד זה מתבצעת אכילה, אם כן מוחזרת ב PieceMove גם מסכת ביטים המייצגת את המיקום של החתיכה שנאכלת בצעד הזה.

לאחר מכן נשלחים מסכת החתיכה שאותה רוצים להזיז ואחד מהPieceMove שמייצגים את המיקום שאליה היא תזוז לפונקציה הדואגת לביצוע המהלך ועדכון הלוח ובאמצעות פעולות לוגיות של OR וXOR של לוח הביטים שהחתיכה בו יחד עם 2 המסכות מעודכן הלוח (במידה והחתיכה היא מלך מעודכן גם לוח המלכים) אם צעד זה היה צעד אכילה ובנוסף ניתן לאכול עם חתיכה זו עוד פעם מחזירה הפונקציה TRUE בכדי לסמן שניתן לבצע שרשרת אכילות אחרי צעד זה.

בכדי להכתיר חתיכה למלך נבדק אם היא זזה לקצה הלוח (על ידי השוואת גודלה לגודל הערך של מסכת תחילת השורה הקיצונית וסוף השורה הקיצונית) ואם כן משנים את לוח הביטים של המלכים ועם פעולת OR מוסיפים את הביט של החתיכה במיקום המתאים.



רשימה

אני נעזר במבנה נתונים מסוג **רשימה** (List) על מנת להחזיר את המהלכים האפשריים של חתיכה מסוימת על הלוח ועל מנת לתאר תור של שחקן בעץ המשחק (רשימה של רצף מהלכים).

עץ משחק

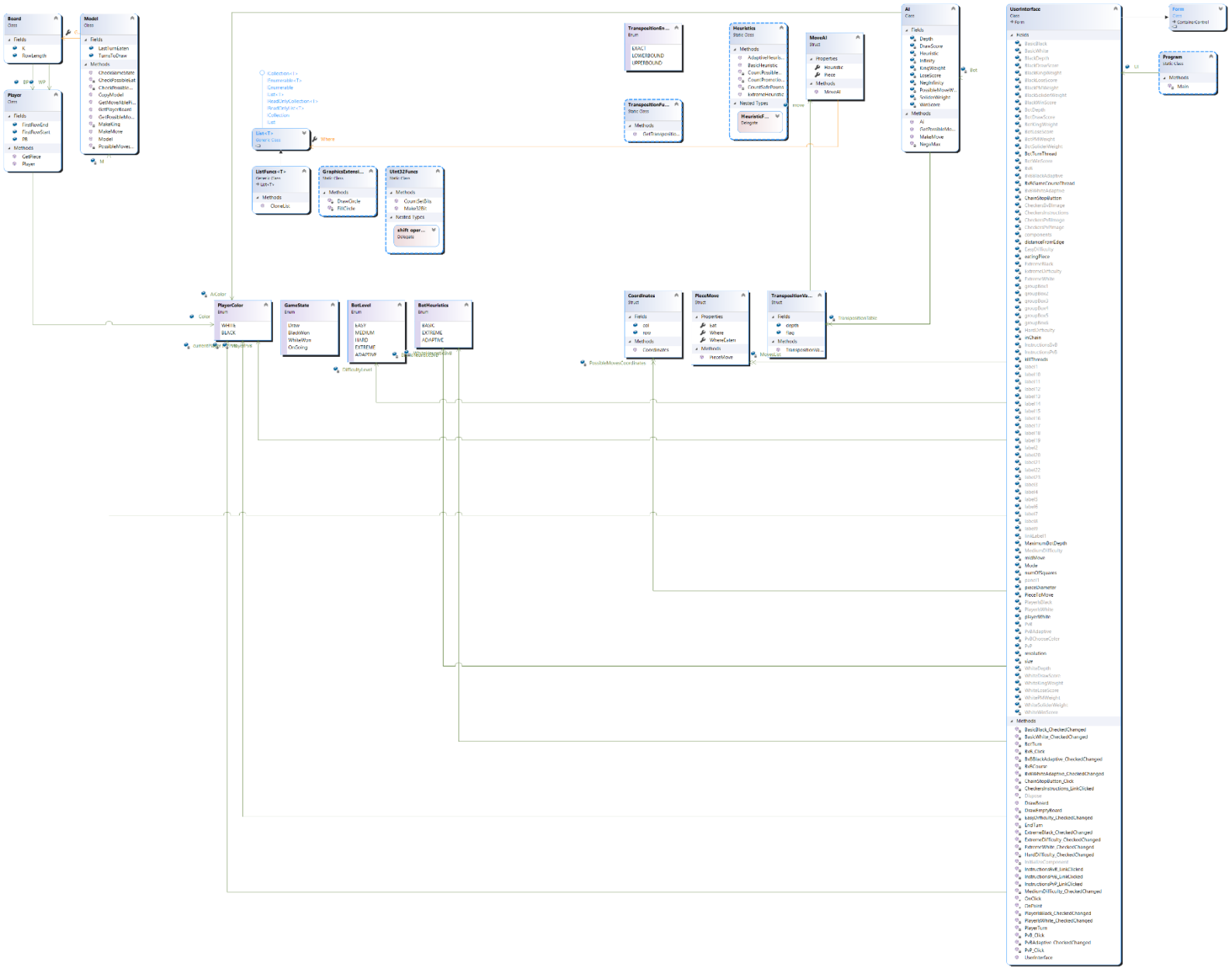
אני משתמש במבנה הנתונים **עץ משחק** (Game tree) עבור הבוט על מנת לגרום לו לשחק בצורה חכמה. אני משתמש באלגוריתם נגה מקס עם גיזום אלפא בטא בכדיי לממש את עץ המשחק.
אלגוריתם זה חוזה X צעדים קדימה במשחק, הוא עובר על כל המהלכים האפשריים שהבוט יכול לבצע כולל שרשראות אכילה שונות שאותן הוא מוצא באמצעות פונקציה רקורסיבית שיוצרת רשימה של רשימות שמייצגת את כל המסלולים האפשריים ללקיחה בשרשרת האכילה (כל המסלולים בעץ שיוצרת השרשרת אכילה), למשל לאכול באלכסון ימין ואז אלכסון שמאל או אלכסון ימין ואז אלכסון ימין או רק אלכסון ימין ושם לסיים את שרשרת האכילה.

לאחר מכן האלגוריתם חוזה את כל הצעדים האפשריים שהיריב יכול לבצע וככה הוא ממשיך עד שהוא מגיע לעומק שניתן לו, בעומק זה הוא מעריך את מצב הלוח באמצעות פונקציה היוריסטית ומחזיר ניקוד כלשהו על מצב הלוח, האלגוריתם יוצא מנקודת הנחה שהבוט יעשה כל פעם את הצעד הכי טוב עבורו ושהיריב של הבוט יעשה כל פעם את הצעד הכי טוב עבורו (שהוא הכי גרוע עבור הבוט) וככה הוא מחזיר את הצעד שעל הבוט לקחת בתור הזה על מנת להגיע לתוצאה הטובה ביותר.

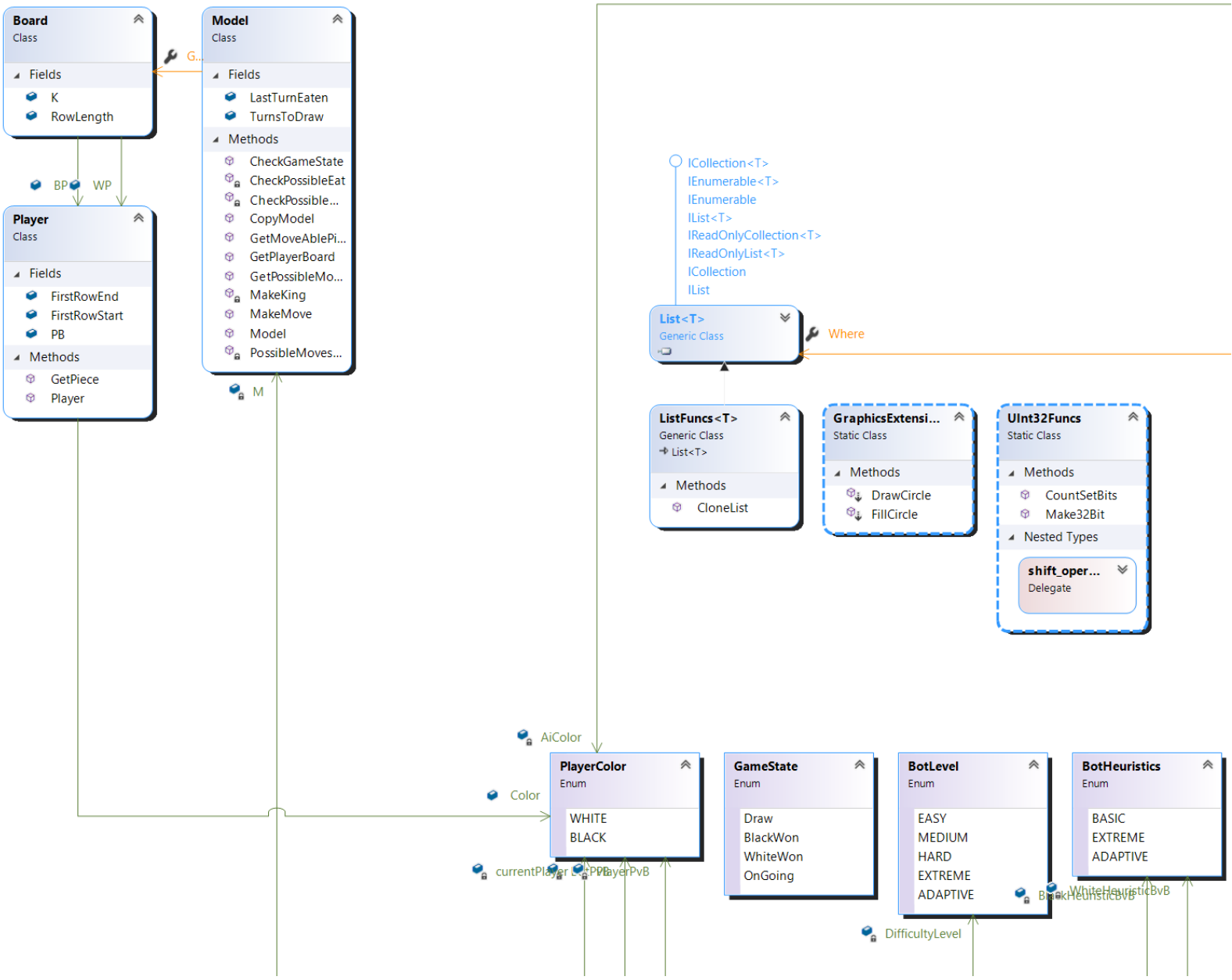
מילון

השתמשתי במבנה הנתונים **מילון** (Dictionary) בשביל טבלת הטרנספוזיציה שנועדה לשפר את יעילות הבוט על ידי שמירת מצבי לוח שכבר דורגו והעומק שאליו הם דורגו, המפתח של המילון נוצר מ3 uint32 שמייצגים את לוח המשחק וint שסופר את מספר המהלכים מהאכילה האחרונה,
המספרים האלה מומרים כל אחד ל4 אותיות ומחוברים למחרוזת בגודל 16 בייט שהיא מפתח ייחודי שמייצג מצב לוח מסוים, הערך של מצב הלוח הזה הוא מורכב מאיזה עומק חישבו אותו ומהניקוד שהוא קיבל מההיוריסטיקה.

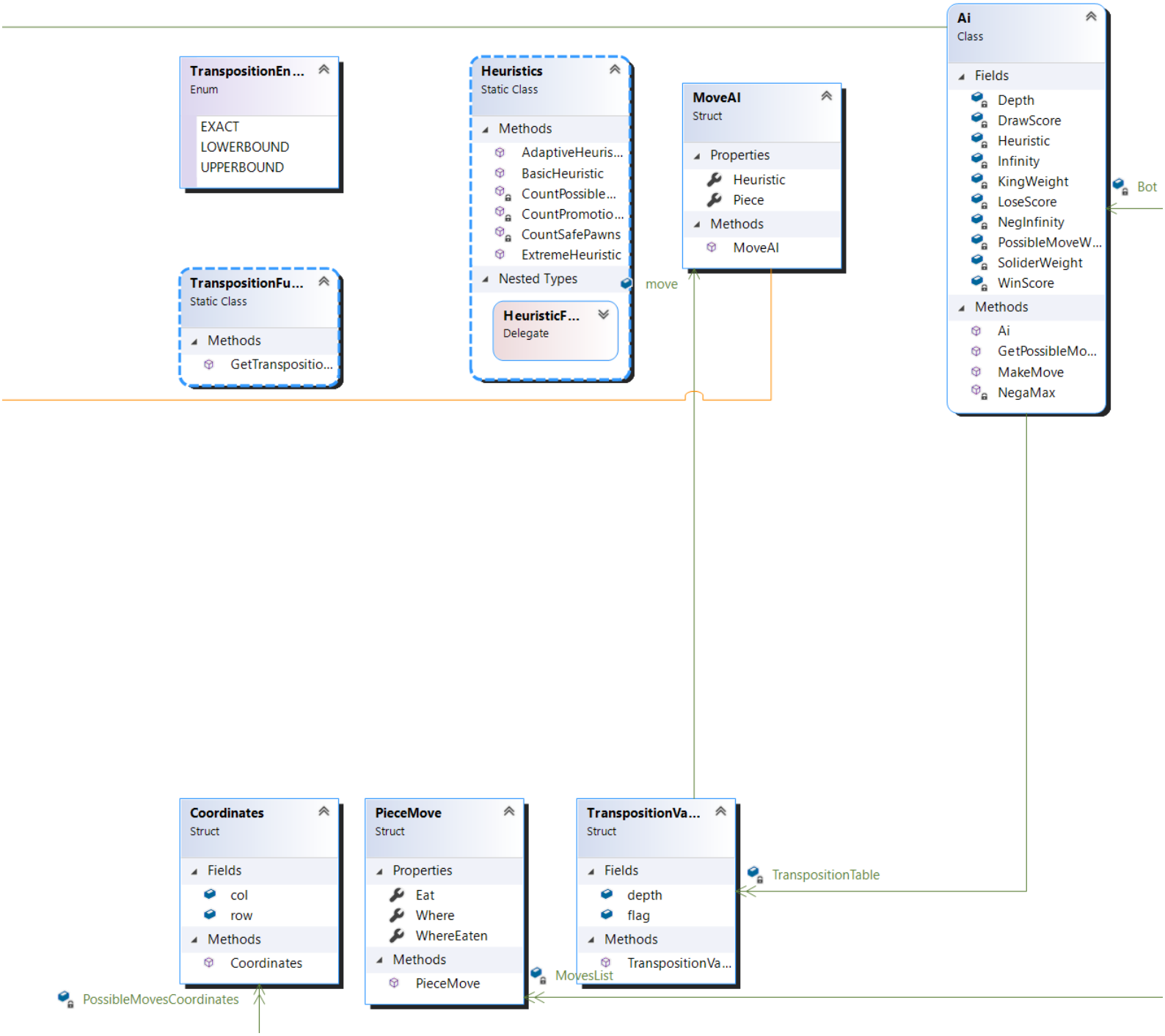
תרשים מחלקות



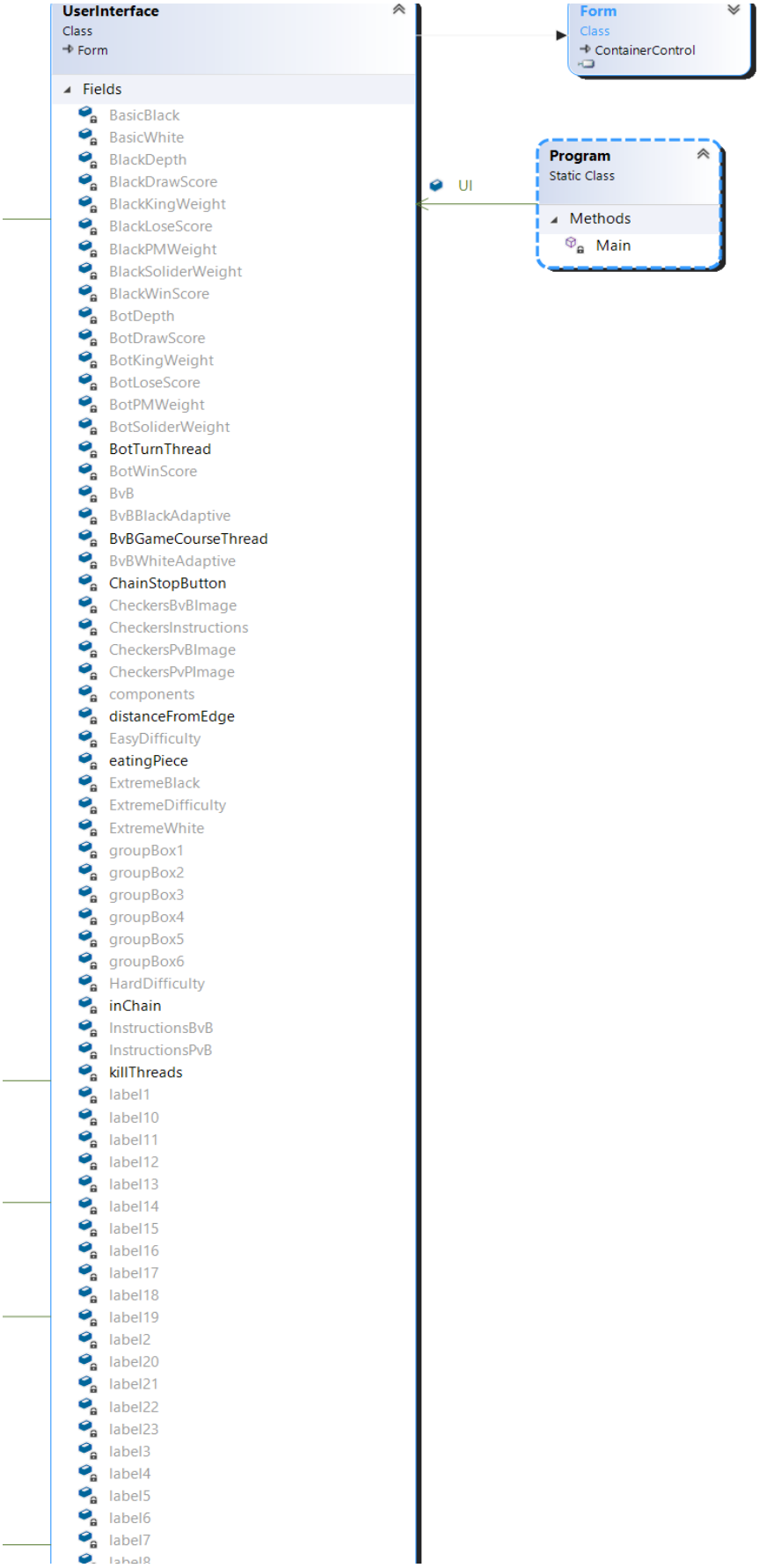
זום על שליש שמאלי ביותר



זום על שליש אמצעי



זום על שליש ימני

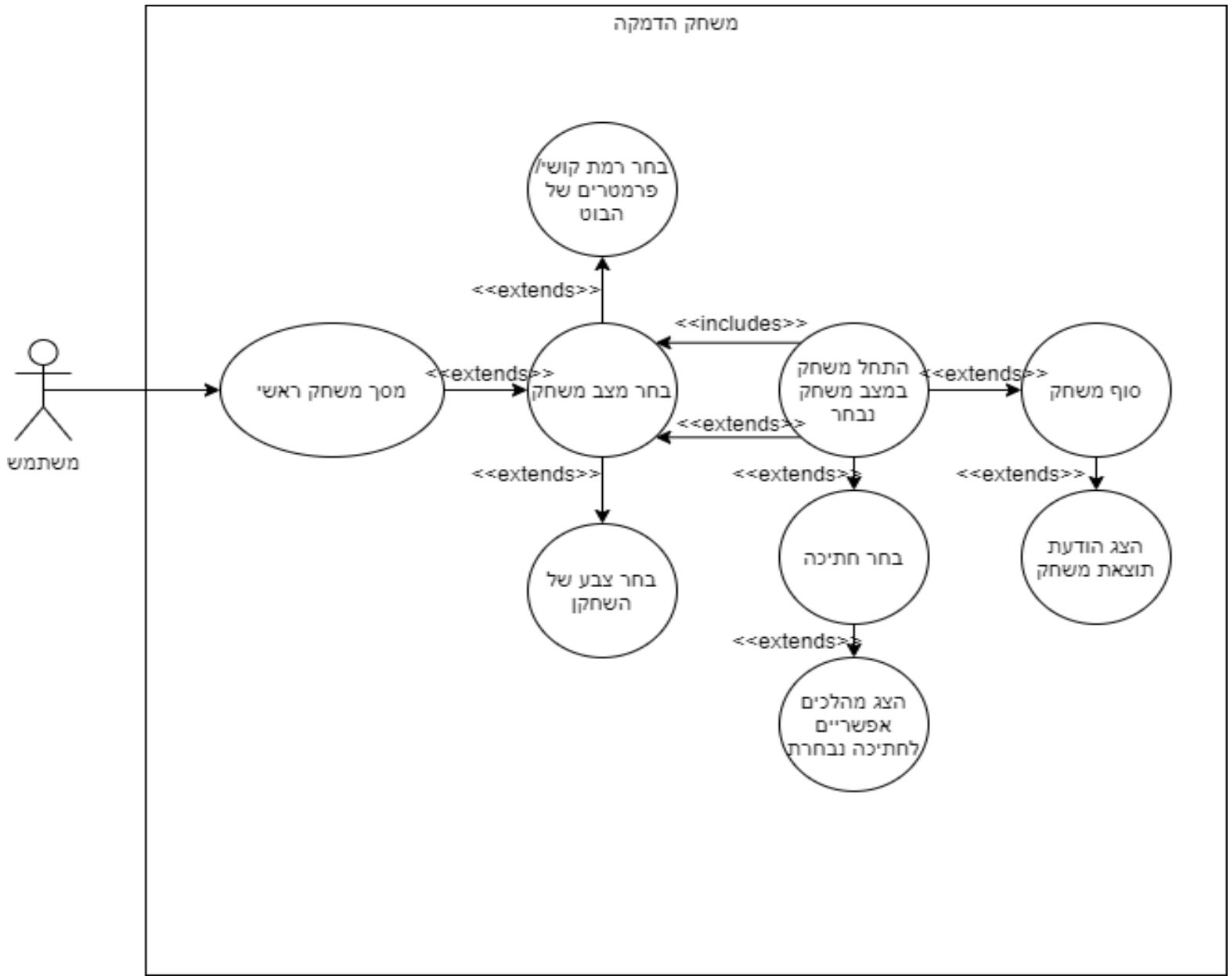


- label9
- linkLabel1
- MaximumBotDepth
- MediumDifficulty
- midMove
- Mode
- numOfSquares
- panel1
- pieceDiameter
- PieceToMove
- PlayerIsBlack
- PlayerIsWhite
- playerWhite
- PvB
- PvBAdaptive
- PvBChooseColor
- PvP
- resolution
- size
- WhiteDepth
- WhiteDrawScore
- WhiteKingWeight
- WhiteLoseScore
- WhitePMWeight
- WhiteSoliderWeight
- WhiteWinScore

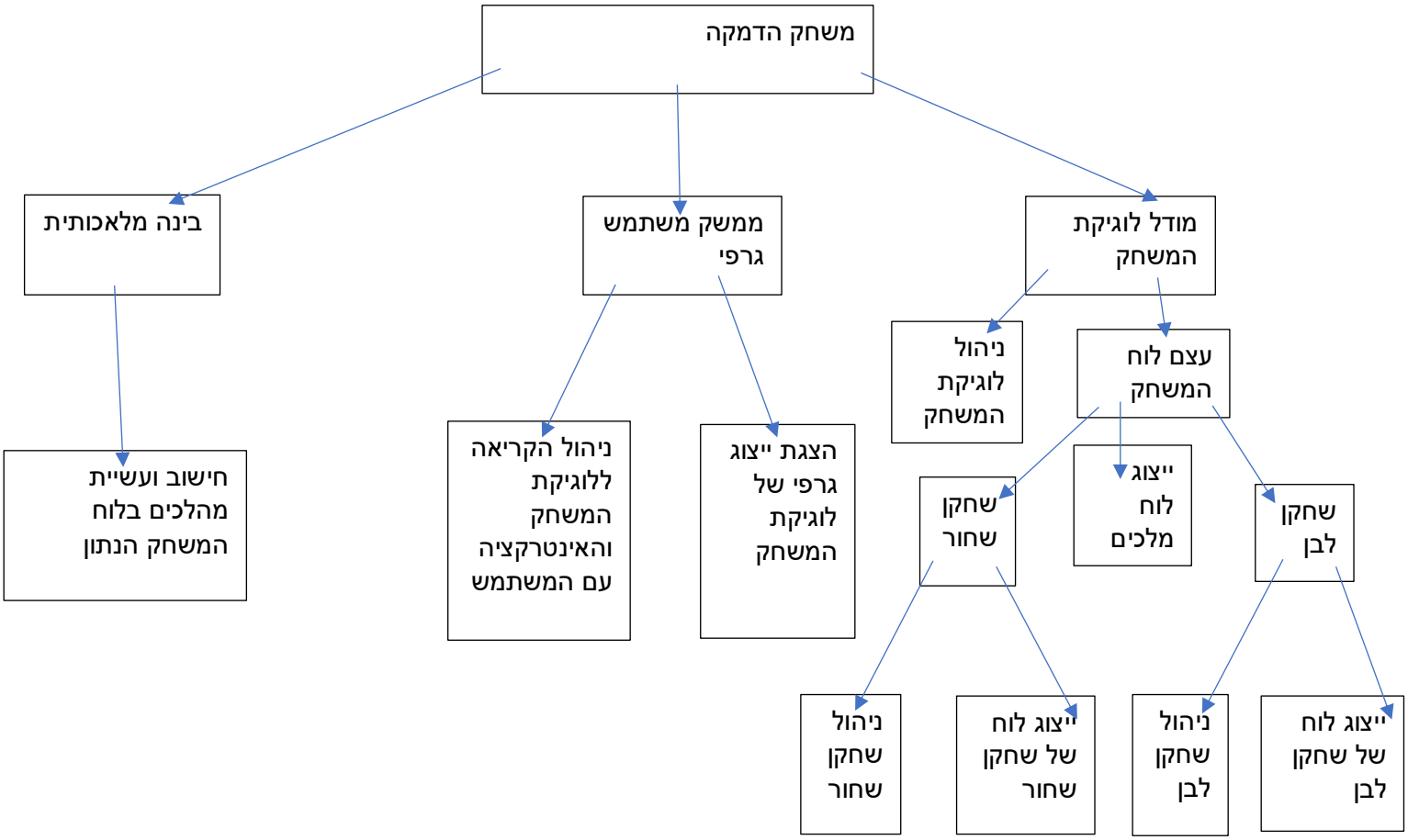
Methods

- BasicBlack_CheckedChanged
- BasicWhite_CheckedChanged
- BotTurn
- BvB_Click
- BvBBlackAdaptive_CheckedChanged
- BvBCourse
- BvBWhiteAdaptive_CheckedChanged
- ChainStopButton_Click
- CheckersInstructions_LinkClicked
- Dispose
- DrawBoard
- DrawEmptyBoard
- EasyDifficulty_CheckedChanged
- EndTurn
- ExtremeBlack_CheckedChanged
- ExtremeDifficulty_CheckedChanged
- ExtremeWhite_CheckedChanged
- HardDifficulty_CheckedChanged
- InitializeComponent
- InstructionsBvB_LinkClicked
- InstructionsPvB_LinkClicked
- InstructionsPvP_LinkClicked
- MediumDifficulty_CheckedChanged
- OnClick
- OnPaint
- PlayerIsBlack_CheckedChanged
- PlayerIsWhite_CheckedChanged
- PlayerTurn
- PvB_Click
- PvBAdaptive_CheckedChanged
- PvP_Click
- UserInterface

תרשים Use Case



ארכיטקטורה של הפתרון המוצע



תיאור סביבת העבודה ושפת התכנות

שפת תכנות

C#

סביבת עבודה

Microsoft Visual Studio Community 2019 Version 16.9.2
.Net Core 3.1

ספריה/תוכנה גרפית

Windows forms app

תיאור ממשקים

הספריות כלליות שבהן השתמשתי הן

```
using System;
```

הספריות הגרפיות שבהן נעזרתי בשביל ליצור את הממשק הגרפי הן

```
using System.Drawing;
using System.Windows.Forms;
```

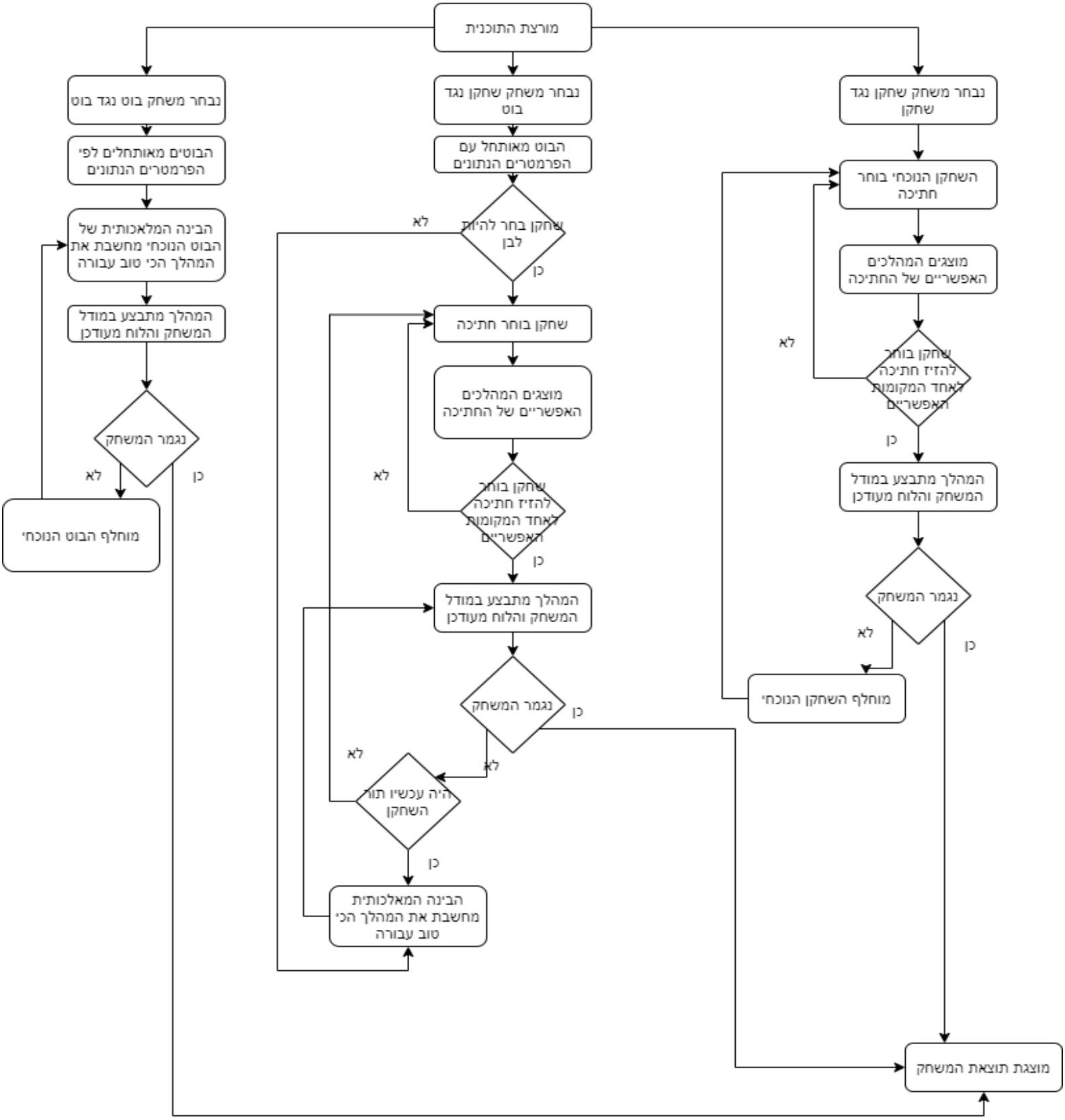
הספריות ניהול תהליכונים שבהן נעזרתי להרצת כמה תהליכים במקביל (למשל לגרום לכך שהבינה המלאכותית תוכל לרוץ במקביל לגרפיקה) הן

```
using System.Threading;
using System.Threading.Tasks;
```

הספריות שבהן נעזרתי בשביל להשתמש באוספים מסוימים (כגון מילון ורשימה) ופעולות עליהם הן

```
using System.Collections.Generic;
```

אלגוריתם ראשי



הפונקציות הראשיות בפרויקט / תיאור המחלקות הראשיות בפרויקט

Classes

public static class GraphicsExtensions – מחלקה המכילה פונקציות עזר לגרפיקה

כותרת הפונקציה	תיאור הפונקציה
public static void DrawCircle(this Graphics g, Pen pen, int centerX, int centerY, int radius)	מצייר מעגל בצבע נתון בקואורדינטות נתונות
public static void FillCircle(this Graphics g, Brush brush, int centerX, int centerY, int radius)	ממלא מעגל בצבע נתון בקואורדינטות נתונות

public class ListFuncs<T>: List<T> - מחלקה המכילה פונקציות עזר למבנה נתונים מסוג רשימה

כותרת הפונקציה	תיאור הפונקציה
public static List<T> CloneList(List<T> ToClone)	משכפל רשימה מסוג כלשהו

public static class TranspositionFuncs – מחלקה המכילה פונקציות עזר לטבלת הטרנספוזיציה

כותרת הפונקציה	תיאור הפונקציה
public static string GetTranspositionKey(Model m)	ממיר לוח למפתח לטבלת הטרנספוזיציה

public static class UInt32Funcs – מחלקה המכילה פונקציות עזר לטיפוס אינטג'ר לא מסומן בעל 32 ביט

כותרת הפונקציה	תיאור הפונקציה
public static string Make32Bit(UInt32 num)	ממיר UInt32 למחרוזת באורך 32
public static int CountSetBits(UInt32 number)	סופר כמה ביטים דולקים בUInt32

מחלקה המכילה את הבינה המלאכותית שמשחקת דמקה – class Ai

כותרת הפונקציה	תיאור הפונקציה
public void MakeMove(Model m)	עושה את המהלך הטוב ביותר לבוט בתור הבוט במודל הנתון
private MoveAI NegaMax(int depth, PlayerColor currentPlayer, int h, Model m, int alpha,int beta)	מוצא את המהלך הטוב ביותר עבור הבוט על ידי שימוש באלגוריתם נגמקס עם גיזום אלפא בטא וטבלת טרנספוזיציה
public void GetPossibleMovesTurn(PlayerColor Player, Model m, PieceMove WherePiece, UInt32 MoveAblePiece, List<PieceMove> MovesList, List<List<PieceMove>> ListOfTurnMoves)	מוצא את כל המהלכים שחתיכה מסוימת יכולה לעשות בכיוון מסוים (למשל אם החתיכה יכולה לבצע שרשרת אכילה אז מחזיר את האפשרות שבה היא לא עושה שרשרת אכילה ואת האפשרות שהיא כן עושה בה וכולי...)

מחלקה המכילה את כל הפונקציות היוריסטיות – static class Heuristics

כותרת הפונקציה	תיאור הפונקציה
public static int BasicHeuristic(Model m, PlayerColor currentPlayer, int KingRatio, int SoliderRatio, int PossibleMovesRatio, int WinScore, int LoseScore, int DrawScore)	נותנת ציון למצב המשחק לטובת השחקן הנוכחי (הנתון)
public static int ExtremeHeuristic(Model m, PlayerColor currentPlayer, int KingRatio, int SoliderRatio, int PossibleMovesRatio, int WinScore, int LoseScore, int DrawScore)	נותנת ציון למצב המשחק לטובת השחקן הנוכחי (הנתון)
public static int AdaptiveHeuristic(Model m, PlayerColor currentPlayer, int KingRatio, int SoliderRatio, int PossibleMovesRatio, int WinScore, int LoseScore, int DrawScore)	נותנת ציון למצב המשחק לטובת השחקן הנוכחי (הנתון)
private static int CountPossibleMoves(Model m, PlayerColor color)	סופר כמה מהלכים שחקן מסוים יכול לעשות במצב הלוח במודל הנתון
private static int CountPromotionLine(Model m, PlayerColor player)	סופר כמה חיילים מגנים על השורה הראשונה של שחקן נתון בלוח במודל הנתון
private static int CountSafePawns(UInt32 Board)	סופר כמה חיילים מוגנים (נמצאים בצדדים בצורה שבה אי אפשר לאכול אותם) יש לשחקן נתון בלוח המודל הנתון

מחלקה שמכילה את התיאור של לוח המשחק (מיקום חיילי 2 השחקנים – public class Board
המאפיינים של הלוח שלהם ולוח המלכים),

מחלקת הלוגיקה הראשית שמכילה את כל לוגיקת משחק הדמקה – public class Model

כותרת הפונקציה	תיאור הפונקציה
public void CopyModel(Model newM)	מעתיק את ערכי המודל לתוך המודל הנתון
public GameState CheckGameState()	מחזיר את מצב המשחק הנוכחי (ניצחון של מישהו או תיקו או מתמשך)
public List<UInt32> GetMoveAblePieces(Player currentPlayer)	מחזיר רשימה של כל החתיכות שהשחקן הנתון יכול להזיז
private void MakeKing(UInt32 kinged)	הופך את החתיכה הנתונה למלך
public Player GetPlayerBoard(PlayerColor player)	מחזיר את הלוח של השחקן הנתון
public bool MakeMove(PlayerColor player, UInt32 piece, PieceMove whereMask)	מזיז את החתיכה הנתונה של השחקן הנתון למיקום הנתון ומבצע אכילה אם היא זזה מעל חתיכה אחרת באלכסון, אם התבצעה אכילה המהלך מחזירה אמת אחרת מחזירה שקר
public List<PieceMove> GetPossibleMoves(PlayerColor player, UInt32 piece)	מחזיר את המהלכים האפשריים של החתיכה הנתונה של השחקן הנתון בתור רשימה
private List<PieceMove> PossibleMovesSpecificPlayer(UInt32 piece, UInt32 playerBoard, UInt32 enemyBoard, PlayerColor player, int nextRowMove, int nextRowDiagonalMoveEven,int nextRowDiagonalMoveOdd, UInt32Funcs.shift_operation op, UInt32Funcs.shift_operation kingBackOp)	מחזיר את המהלכים האפשריים של החתיכה הנתונה של שחקן ספציפי שפרטי הלוח שלו נתונים בתור רשימה
private PieceMove CheckPossibleMove(UInt32 piece, int pieceIndexCheck, UInt32Funcs.shift_operation op, UInt32 playerBoard, UInt32 enemyBoard, PlayerColor player, int move, bool backWardsMove)	בודק אם מהלך ספציפי אפשרי לחתיכה נתונה ואם כן מחזיר תיאור (מבנה) של המהלך
private PieceMove CheckPossibleEat(UInt32 playerBoard, UInt32 enemyBoard, UInt32 possiblyEdible, int moveEven, int moveOdd, UInt32Funcs.shift_operation op)	בודק אם החתיכה הנתונה ניתנת לאכילה עבור שחקן ספציפי שאת פרטי הלוח שלו היא מקבלת ואם כן מחזיר תיאור (מבנה) של האכילה של החתיכה

מחלקה המתארת שחקן על ידי מיקום כל חייליו ומאפייני הלוח שלו – public class Player

כותרת הפונקציה	תיאור הפונקציה
public UInt32 GetPiece(int selectedPieceIndex)	מוצא מסיכה המתארת חתיכה באינדקס הנתון (הביט דולק במיקום של האינדקס וכל שאר 32 הביטים כבויים)

מחלקה המנהלת את הממשק משתמש ומחברת – public partial class UserInterface : Form
את הלוגיקה עם הגרפיקה לניהול מהלך משחק

כותרת הפונקציה	תיאור הפונקציה
public void DrawBoard(Board b)	מצייר את הלוח הנתון במצב הנוכחי של המשחק על המסך
protected override void OnClick(EventArgs e)	מה שקורה בלחיצת עכבר על המסך, תלוי באיזה מצב משחק מתקיים כרגע ואיפה הלחיצה התרחשה מגיב בצורה שונה
private void PlayerTurn(int row, int col)	מטפל בתור של שחקן על ידי קריאה לפונקציות הלוגיות הנכונות ועדכון הלוח כשצריך
private void EndTurn()	מסיים תור של שחקן ומחליף שחקן
private void PvP_Click(object sender, EventArgs e)	מאתחל מצב משחק לשחקן נגד שחקן ומתחיל את המשחק
private void BotTurn()	מטפל בתור של הבוט על ידי קריאה לפונקציות הלוגיות הנכונות ועדכון הלוח כשצריך
private void PvB_Click(object sender, EventArgs e)	מאתחל מצב משחק של שחקן נגד בוט שהגדרותיו הן לפי מה שבחר המשתמש ומתחיל את המשחק
private void BvBCourse(Ai BotWhite, Ai BotBlack)	מטפל במהלך משחק של בוט נגד בוט על ידי קריאה לפונקציות הלוגיות הנכונות ועידכון הלוח כשצריך
private void BvB_Click(object sender, EventArgs e)	מאתחל מצב משחק לבוט נגד בוט שהגדרותיו הן לפי מה שבחר המשתמש ומתחיל את המשחק

Enums

public enum TranspositionEnum – מתאר גבול תחתון, עליון או מדויק

public enum GameState – מתאר את מצב המשחק הנוכחי

public enum PlayerColor – מתאר צבע של שחקן

public enum BotHeuristics – מתאר סוג יוריסטיקה של בוט

public enum BotLevel – מתאר רמת קושי של בוט

Structs

public struct MoveAI – מתאר מהלך של הבינה המלאכותית (איזו חתיכה זזה, לאיפה והערך היוריסטי של מהלך זה)

public struct TranspositionValue – מתאר ערך בטבלת הטרנספוזיציה (העומק שבו נמצא הערך למצב הלוח, המהלך של הבינה המלאכותית למצב לוח זה ודגל הנועד לגיזום אלפא בטא)

public struct PieceMove – מתאר מהלך של חתיכה במשחק (לאיפה זזה החתיכה, אם היא אוכלת בזמן במהלך או לא ואם כן אז באיזה מיקום היא אוכלת)

public struct Coordinates – מתאר קואורדינטות על הלוח (שורה וטור)

פונקציות ראשיות בתיאור רחב

כותרת הפונקציה:

```
private MoveAI NegaMax(int depth, PlayerColor currentPlayer, Model m, int alpha,int beta)
```

הפרמטרים של הפונקציה :

- Int depth – המרחק הנוכחי בעץ משחק מהעלים (מתחיל בתור גובה עץ המשחק ויורד ל0)
- PlayerColor currentPlayer – השחקן הנוכחי שבודקים את המהלכים שלו
- Model m – המודל שמתאר את מצב המשחק הנוכחי
- Int alpha – חסם תחתון
- Int beta – חסם עליון

מה הפונקציה מחזירה :

הפונקציה מחזירה את המהלך הכי טוב שהשחקן הנוכחי יכול לבצע והניקוד של מהלך זה על ידי פונקציה היוריסטית

מה הפונקציה מבצעת :

הפונקציה מוצאת את המהלך הכי טוב לשחקן הנוכחי ואת הניקוד שלו על ידי חישוב כל מצבי המשחק depth תורות קדימה וניקוד מצב הלוח לטובת השחקן

יעילות הפונקציה :

כאשר N מייצגת את מספר המהלכים האפשריים של שחקן ,H מייצגת את עומק העץ וM מייצגת את סיבוכיות הפונקציה היוריסטית הסיבוכיות היא $O(N^H + (N^H - N^{H-1}) \times M$

להלן תיאור פסודו קוד של הפונקציה :

```
function negaMax(model, depth, α, β, color) is

    alphaOrig := α

    (* Transposition Table Lookup; model is the lookup key for ttEntry*)
    ttEntry := transpositionTableLookup(model)
    if ttEntry is valid and ttEntry.depth ≥ depth then
        if ttEntry.flag = EXACT then
            return ttEntry.value
        else if ttEntry.flag = LOWERBOUND then
            α := max(α, ttEntry.value)
        else if ttEntry.flag = UPPERBOUND then
            β := min(β, ttEntry.value)

        if α ≥ β then
            return ttEntry.value

    if depth <= 0 or model is a terminal model then
        return color × the heuristic value of model

    childModels := generateMoves(model)
    childModels := orderMoves(childModels)
    value := −∞
    for each child in childModels do
        value := max(value, −negaMax(child, depth − 1, −β, −α, −color))
        α := max(α, value)
        if α ≥ β then
            break

    (* Transposition Table Store; model is the lookup key for ttEntry *)
    ttEntry.value := value
    if value ≤ alphaOrig then
        ttEntry.flag := UPPERBOUND
    else if value ≥ β then
        ttEntry.flag := LOWERBOUND
    else
        ttEntry.flag := EXACT
    ttEntry.depth := depth
    transpositionTableStore(model, ttEntry)

    return value
```

כותרת הפונקציה:

public static string GetTranspositionKey(Model m)

הפרמטרים של הפונקציה :

- Model m – המודל שמתאר את מצב המשחק הנוכחי

מה הפונקציה מחזירה :

הפונקציה מחזירה מחרוזת ייחודית למצב הלוח במודל שנשלח שמשמשת בתור מפתח לטבלת הטרנספוזיציה

מה הפונקציה מבצעת :

הפונקציה ממירה את 3 הUInt32 שמייצגים את הלוח ואת הInt שמייצג את מספר התורות מאז האכילה האחרונה למחרוזת באורך 16 תווים שמורכבת מהבתים שלהם

יעילות הפונקציה : $O(1)$

להלן תיאור פסודו קוד של הפונקציה :

```
function getTranspositionKey(model) is

 c[16] = char array
c[0] to c[3] = 4 bytes of white player board
c[4] to c[7] = 4 bytes of black player board
c[8] to c[11] = 4 bytes of kings board
c[12] to c[15] = 4 bytes of int representing moves from last eat

    return c converted to string
```

כותרת הפונקציה:

```
public void GetPossibleMovesTurn(PlayerColor Player, Model m, PieceMove
WherePiece, UInt32 MoveAblePiece, List<PieceMove> MovesList,
List<List<PieceMove>> ListOfTurnMoves)
```

הפרמטרים של הפונקציה :

- PlayerColor Player – השחקן הנוכחי שעבורו מתבצעת הפונקציה
- Model m – המודל שמתאר את מצב המשחק הנוכחי
- PieceMove WherePiece – לאיפה להזיז את החתיכה הנתונה
- UInt32 MoveAblePiece – החתיכה הניתנת להזזה
- List<PieceMove> MovesList – מסלול נוכחי המיוצג באמצעות רשימת מהלכים לאכילת שרשרת
- List<list<PieceMove>> ListOfTurnsMoves – רשימה שבה הפונקציה שומרת את כל המסלולים האפשריים של אכילות שרשרת המיוצגות על ידי רשימות מהלכים

מה הפונקציה מחזירה :

הפונקציה לא מחזירה דבר

מה הפונקציה מבצעת :

הפונקציה שומרת בתוך ListOfTurnsMoves רשימות שכל אחת מייצגת רצף מהלכים שהחתיכה הראשונה שנשלחה (MoveAblePiece) יכולה לבצע בתור אחד עם צעד ראשון לכיוון ספציפי (WherePiece) (שרשראות אכילה שונות או צעד רגיל לכיוון הנתון).

לדוגמה חתיכה שחורה שיכולה לאכול חתיכה לבנה ימנית ראשונה ואז חתיכה לבנה שמאלית שניה או חתיכה לבנה ימנית ראשונה ואז חתיכה לבנה ימנית שניה או רק חתיכה לבנה ימנית ראשונה ישמרו עבור תזוזה שלה ימינה 3 רשימות בתוך הListOfTurnsMoves שכל אחת מייצגת רצף מהלכים אפשרי אחר.

יעילות הפונקציה : כאשר N מייצג את מספר רצפי המהלכים האפשריים שהחתיכה יכולה לבצע בכיוון מסוים אז הסיבוכיות היא $O(N)$

להלן תיאור פסודו קוד של הפונקציה :

```
function getPossibleMovesTurn(playerColor, model, wherePiece, moveAblePiece, movesList,
listOfTurnMoves)is

  If after moving moveAblePiece to wherePiece no more eating moves can be made then
          add wherePiece to movesList
          add clone of movesList to listOfTurnMoves
          return // exit recursion

   saveModel = copy of model
   add wherePiece to movesList
   add clone of movesList to listOfTurnMoves
   moveAblePiece = where the moveAblePiece moved to
   saveMovesList = copy of movesList

   If moveAblePiece can no longer make any none eating moves then
          return // exit recursion

   edibleNodes = All moveAblePiece's new possible eating moves

   For each node in edibleNodes
          getPossibleMovesTurn(playerColor, model, node, moveAblePiece, movesList,
listOfTurnMoves)
          movesList = copy of saveMovesList // restore moveslist
          model = copy of saveModel // restore model
```

מדריך למשתמש

בכדי להריץ את הפרויקט אפשר לפתוח אותו דרך סביבת העבודה visual studio 2019 ולהריץ או להריץ את קובץ הexe המצורף.

כשמריצים את הפרויקט מופיע מסך שבו יש לוח דמקה ריק ומעליו קישור לחלון הוראות על איך לשחק דמקה,בצד ימין ישנם כפתורים ל- 3 מצבי המשחק שניתן להריץ מתחת לכפתור בחירה של כל אחד מהם מופיעים קישורים לחלוניהוראות של מצב המשחק הזה ומתחת לחלקם ישנם פרמטרים ואפשרויות בחירה לבינה המלאכותית ואפשרויות בחירה לצבע השחקן.

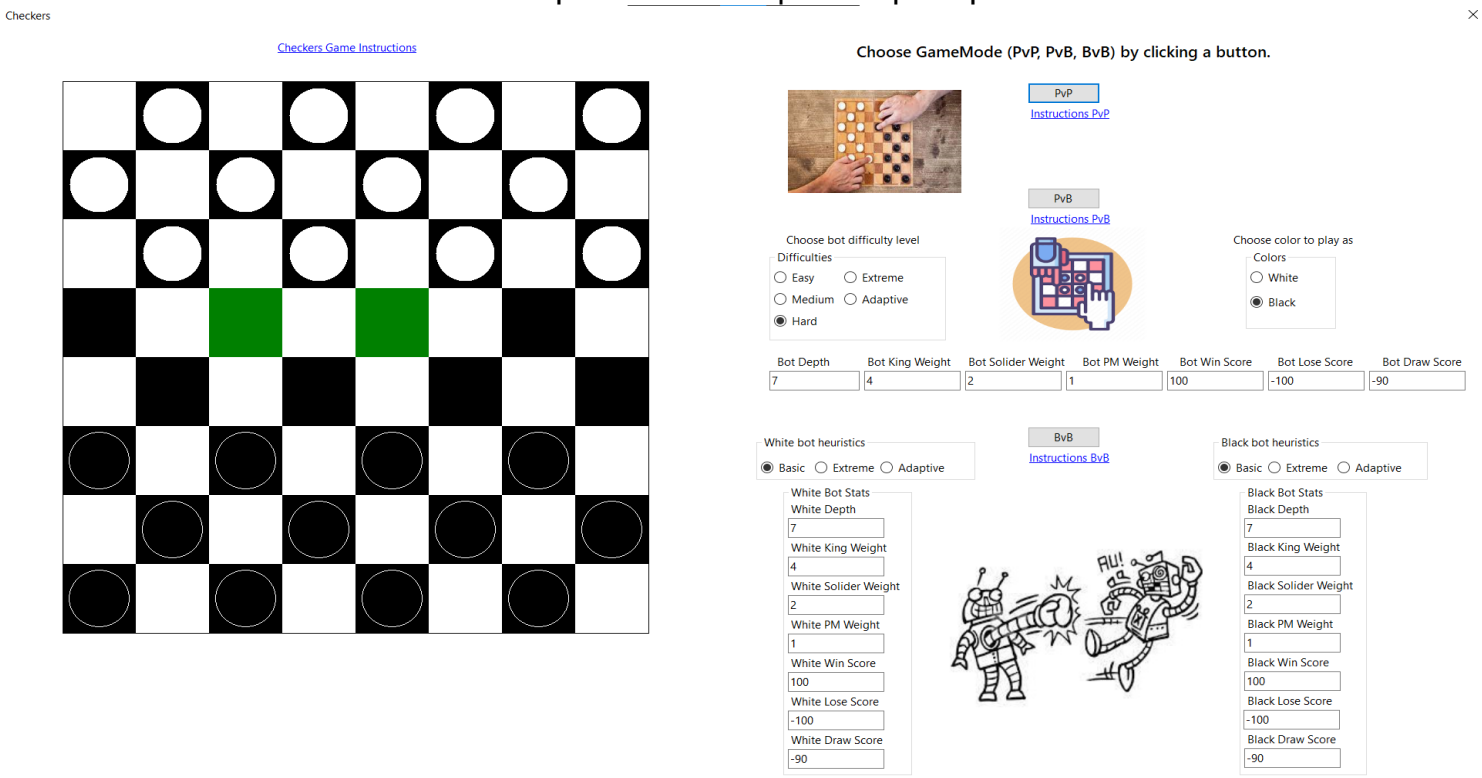
אם לוחצים על כפתור של מצב משחק מסוים המשחק מתחיל עם הפרמטרים הרשומים והאפשרויות המסומנות של מצב משחק זה, ניתן ללחוץ על כפתור זה גם במהלך משחק ובכך לאפס את המשחק ולהחליף מצבי משחק.

3 מצבי המשחק הם:

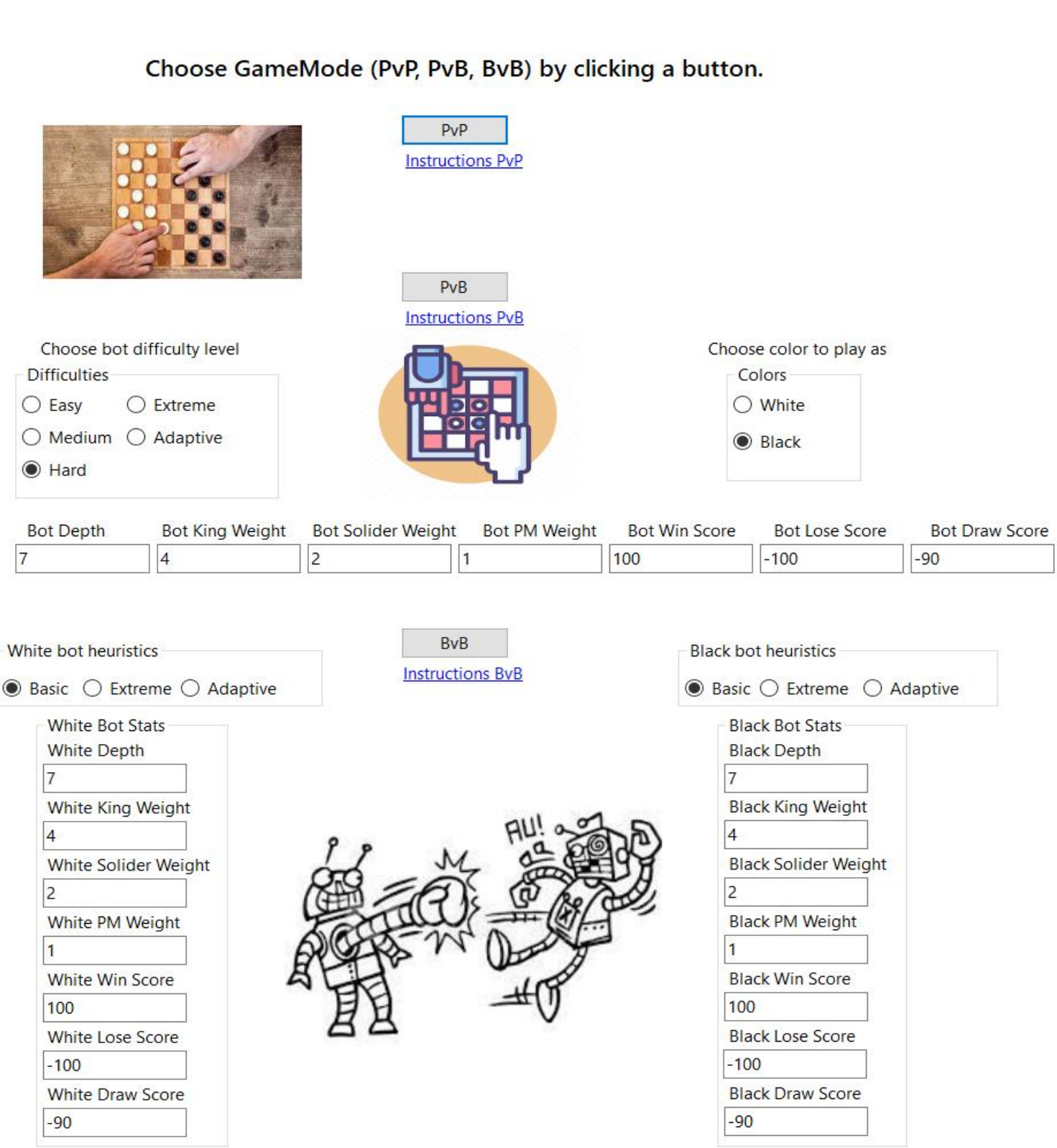
- שחקן נגד שחקן (PvP) – 2 שחקנים אנושיים משחקים
- שחקן נגד בוט (PvB) – שחקן משחק נגד בוט ויכול לבחור את הצבע שלו ואת רמת הקושי/ הפרמטרים של הבוט
- בוט נגד בוט (BvB) – בוט משחק נגד בוט וניתן לבחור עבור כל אחד מהבוטים את הפרמטרים שלו

בכדי להזיז חתיכה על הלוח בתור שחקן ניתן ללחוץ על החתיכה ואז מסומנות בירוק המשבצות שאליהן יכולה החתיכה לזוז, כשלוחצים על אחת המשבצות האלה זזה אליה החתיכה. ניתן ללחוץ גם על חתיכה אחרת שרוצים להזיז במקום והתהליך יחזור על עצמו.

בכדי לצאת מהתוכנית ניתן ללחוץ על האיקס למעלה בצד ימין.



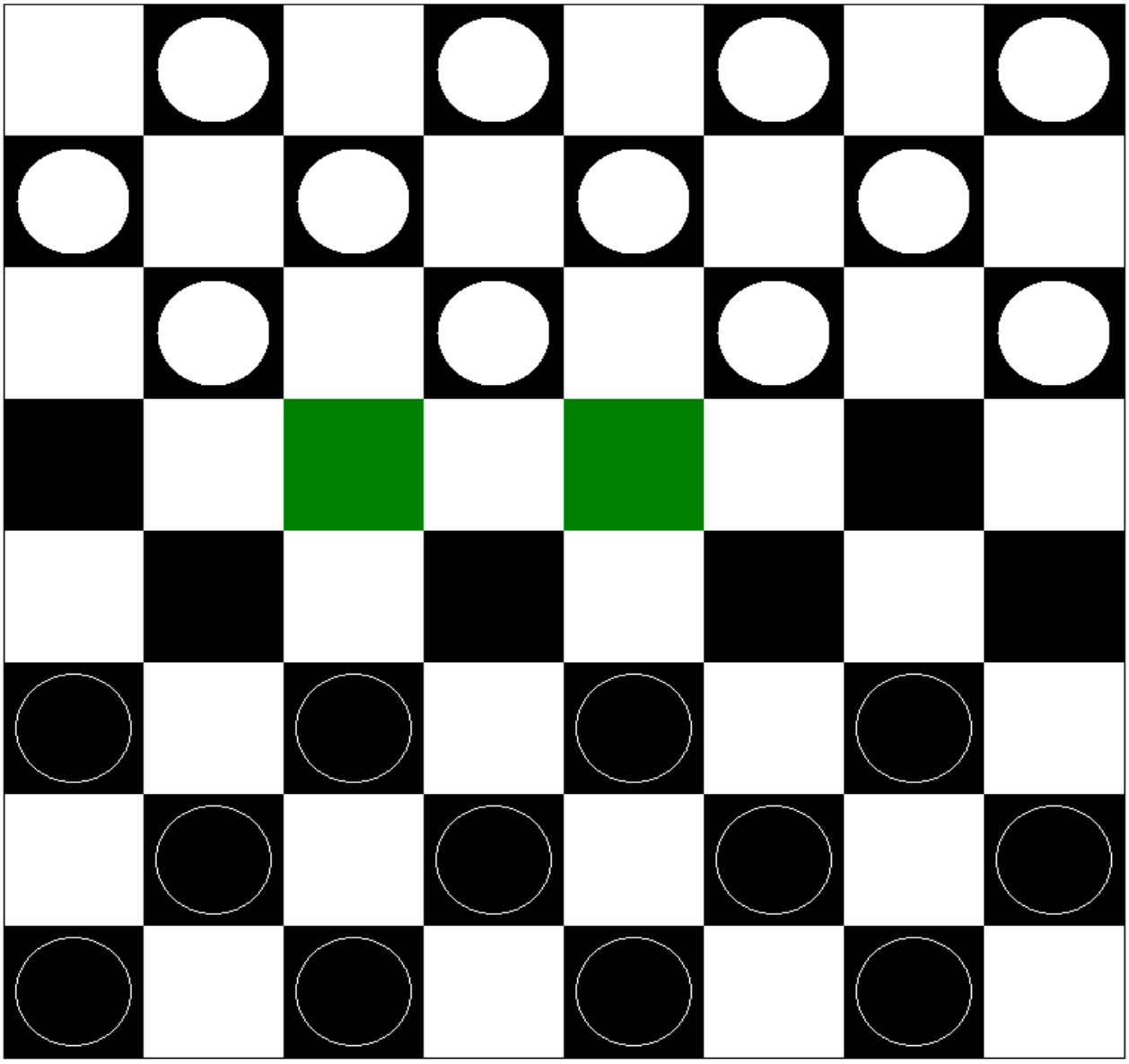
<u>זום חצי ימני</u> – תפריט והגדרות



זום חצי שמאלי – לוח משחק והוראות למשחק דמקה

Checkers

Checkers Game Instructions



קוד הפרוייקט – ייכתב על פי הסטדנטים בליווי תיעוד

```
using System.Drawing;

namespace Checkers
{
   /// <summary>
   /// Graphics Extensio functions written by me for use in the project
   /// </summary>
   public static class GraphicsExtensions
   {
      /// <summary>
      /// Draw Circle
      /// </summary>
      /// <param name="g"> Graphics object </param>
      /// <param name="pen"> Pen to draw with </param>
      /// <param name="centerX"> X Coordinate of circle center </param>
      /// <param name="centerY"> Y Coordinate of circle center </param>
      /// <param name="radius"> Radius of circle </param>
      public static void DrawCircle(this Graphics g, Pen pen,
                              int centerX, int centerY, int radius)
      {
         g.DrawEllipse(pen, centerX - radius, centerY - radius,
                  radius + radius, radius + radius);
      }

      /// <summary>
      /// Fill Circle
      /// </summary>
      /// <param name="g"> Graphics object </param>
      /// <param name="brush"> Brush to Fill circle with </param>
      /// <param name="centerX"> X Coordinate of circle center </param>
      /// <param name="centerY"> Y Coordinate of circle center </param>
      /// <param name="radius"> Radius of circle </param>
      public static void FillCircle(this Graphics g, Brush brush,
                              int centerX, int centerY, int radius)
      {
         g.FillEllipse(brush, centerX - radius, centerY - radius,
                  radius + radius, radius + radius);
      }
   }
}
```

```
using System.Collections.Generic;


namespace Checkers
{
   /// <summary>
   /// Added List Functons Written by me for use in the project
   /// </summary>
   /// <typeparam name="T"> Generic Type Of List </typeparam>
   public class ListFuncs<T>: List<T>
   {
      /// <summary>
      /// Clones list of type t
      /// </summary>
      /// <param name="ToClone"> list to clone </param>
      /// <returns> returns clone of list of type t </returns>
      public static List<T> CloneList(List<T> ToClone)
      {
         if (ToClone == null)
            return new List<T>();

         List<T> Clone = new List<T>();
         foreach(T item in ToClone)
         {
            Clone.Add(item);
         }
         return Clone;
      }
   }
}
```

```
using System;

namespace Checkers
{
```

```
    /// <summary>
    /// All supporting functions for the transposition table
    /// </summary>
    public static class TranspositionFuncs
    {
        /// <summary>
        /// Get Unique key made out of the details of the given board (the position of pieces, number of turns
since last eating and who'se turn it is now)
        /// </summary>
        /// <param name="m"> model representing the game </param>
        /// <param name="futureView"> How much further does the bot normally see </param>
        /// <returns> String representing a Zobrist key for the transposition table </returns>
        public static string GetTranspositionKey(Model m)
        {
            char[] chars = new char[16];

            Board b = m.GameBoard;

            UInt32 BB = b.BP.PB;
            UInt32 WB = b.WP.PB;
            UInt32 KB = b.K;
            int draw = m.TurnsToDraw;

            // board
            chars[0] = (char)(BB & 0xFF);
            chars[1] = (char)(BB >> 8 & 0xFF);
            chars[2] = (char)(BB >> 16 & 0xFF);
            chars[3] = (char)(BB >> 24 & 0xFF);
            chars[4] = (char)(WB & 0xFF);
            chars[5] = (char)(WB >> 8 & 0xFF);
            chars[6] = (char)(WB >> 16 & 0xFF);
            chars[7] = (char)(WB >> 24 & 0xFF);
            chars[8] = (char)(KB & 0xFF);
            chars[9] = (char)(KB >> 8 & 0xFF);
            chars[10] = (char)(KB >> 16 & 0xFF);
            chars[11] = (char)(KB >> 24 & 0xFF);

            // Add How close the game is to a draw as part of the board key
            chars[12] = (char)(draw & 0xFF);
            chars[13] = (char)(draw >> 8 & 0xFF);
            chars[14] = (char)(draw >> 16 & 0xFF);
            chars[15] = (char)(draw >> 24 & 0xFF);

            return new string(chars);

        }
    }
}



using System;

namespace Checkers
{
    /// <summary>
```

```
    /// Contains all selfmade used UInt32 functions/delegates
    /// </summary>
    public static class UInt32Funcs
    {
        /// <summary>
        /// preforms shift operation on given piece
        /// </summary>
        /// <param name="piece"> given piece to preform shift operation on </param>
        /// <param name="bits"> how many bits to shift </param>
        /// <returns> needed shift operation on given piece </returns>
        public delegate UInt32 shift_operation(UInt32 piece, int bits);


        /// <summary>
        /// converts UInt32 to a 32 character string adds 0 to the left of the board until its length is 32
        /// </summary>
        /// <param name="num"> UInt32 value to convert to string </param>
        /// <returns> num as a 32 character long string </returns>
        public static string Make32Bit(UInt32 num)
        {
            string strNum = (Convert.ToString(num, toBase: 2)); // convert to string
            while (strNum.Length != 32) // make sure string contains all 32 bits
            {
                strNum = "0" + strNum;
            }
            return strNum;
        }


        /// <summary>
        /// Count how many set bits are in given UInt32 , O(1) complexity
        /// </summary>
        /// <param name="number"> number to count set bytes of </param>
        /// <returns> return count of set bits </returns>
        public static int CountSetBits(UInt32 number)
        {
            number = number - ((number >> 1) & 0x55555555);
            number = (number & 0x33333333) + ((number >> 2) & 0x33333333);
            return (int)(((number + (number >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
        }
    }
}
```

```
using System;

using System.Collections.Generic;


namespace Checkers
```

```
}
> ///    summary<
 ///    Object Representing the AI With the given Heuristic, parameters and Ai color
/> ///    summary<
   class Ai
}
> ///       summary<
 ///       Value reaching for inifinity to assign to defaultly assign
/> ///       summary<
      private int Infinity = 999999999;



> ///       summary<
 ///       Value reaching for -inifinity to assign to defaultly assign
/> ///       summary<
      private int NegInfinity = -999999999;

> ///       summary<
 ///       The color of the bot
/> ///       summary<
      private PlayerColor AiColor;

> ///       summary<
 ///       The heuristic function that scores the board in the minimax algorithem
/> ///       summary<
      private Heuristics.HeuristicFunc Heuristic;

> ///       summary<
 ///       The depth the game tree will reach
/> ///       summary<
      private int Depth;

 //       The Heuristic funcion's parameters (Scores for diffrent situatons and weights for diffrent pieces)
```

```
> ///      summary<
 ///      heuristic parameter The weight of a king
/> ///      summary<
      private int KingWeight;

> ///      summary<
 ///      heuristic parameter The weight of a solider
/> ///      summary<
      private int SoliderWeight;

> ///      summary<
 ///      heuristic parameter the weight of a possible move
/> ///      summary<
      private int PossibleMoveWeight;

> ///      summary<
 ///      heuristic parameter The score for winning
/> ///      summary<
      private int WinScore;

> ///      summary<
 ///      heuristic parameter The score for losing
/> ///      summary<
      private int LoseScore;

> ///      summary<
 ///      heuristic parameter The score for draw
/> ///      summary<
      private int DrawScore;

> ///      summary<
 ///      Transposition table saves board states values
```

```
 ///      size set to 1 million so it wont waste time reallocating memory in smaller depths

/> ///      summary<

      private Dictionary<string, TranspositionValue> TranspositionTable = new Dictionary<string,
TranspositionValue;(1000000)<


> ///      summary<

 ///      Constructor

/> ///      summary<

> ///      param name="AColor"> The color of the bot </param<

> ///      param name="HeuristicFunction"> The heuristic function that scores the board in the minimax
algorithem </param<

> ///      param name="DepthTree"> how many moves forwards the bot will predict </param<

> ///      param name="kingWeight"> heuristic parameter The weight of a king </param<

> ///      param name="soliderWeight"> heuristic parameter The weight of a solider </param<

> ///      param name="possibleMoveWeight"> heuristc parameter the weight of a possible move </param<

> ///      param name="winScore"> heuristic parameter The score for winning </param<

> ///      param name="loseScore"> heuristic parameter The score for losing </param<

> ///      param name="drawScore"> heuristic parameter The score for a draw</param<

      public Ai(PlayerColor AColor, Heuristics.HeuristicFunc HeuristicFunction, int DepthTree, int kingWeight, int
soliderWeight, int possibleMoveWeight,

          int winScore, int loseScore, int drawScore(

}

          AiColor = AColor;


          Heuristic = HeuristicFunction;


          Depth = DepthTree;


          KingWeight = kingWeight;


          SoliderWeight = soliderWeight;


          PossibleMoveWeight = possibleMoveWeight;
```

```
        WinScore = winScore;

        LoseScore = loseScore;

        DrawScore = drawScore;
{

> ///     summary<
 ///     gets the board and makes the best move it found
/> ///     summary<
> ///     param name="m"> the game model</param<
    public void MakeMove(Model m)
}
 //         predict Depth turns into the future
        MoveAI moveToMake = NegaMax(Depth, AiColor, m, NegInfinity, Infinity);

        m.MakeMove(AiColor, moveToMake.Piece, moveToMake.Where[0]);

 //         If turn contains a chain eat, preform chain eating
        int moveNumber = moveToMake.Where.Count;
        for (int move = 1; move < moveNumber; move++)
}
          m.MakeMove(AiColor, moveToMake.Where[move - 1].Where, moveToMake.Where[move]);
{
{

> ///     summary<
 ///     Finds best move the bot can make using negamax algorithem (which goes over a game tree
representing all possible moves in each given turn in the game up to a certain depth)
/> ///     summary<
> ///     param name="depth"> current depth in the game tree (starts as max depth and goes down to 0)
</param<
> ///     param name="currentPlayer"> current player being rated </param<
> ///     param name="m"> the model of the game </param<
```

```
> ///     param name="alpha"> the lower bound </param<
> ///     param name="beta"> the upper bound </param<
> ///     returns> Returns the optimal value a maximizer can obtain and the move leading to it </returns<
    private MoveAI NegaMax(int depth, PlayerColor currentPlayer, Model m, int alpha,int beta)
}
//        Get the transposition key for given board
        string TranspositionKey = TranspositionFuncs.GetTranspositionKey(m);

//        If Heuristic For Board was already calculated at an equal or further distance from the leafes as the current depth, Return what has been calculated
        if (TranspositionTable.ContainsKey(TranspositionKey) && TranspositionTable[TranspositionKey].depth >= depth)
}
            TranspositionValue tableMove = TranspositionTable[TranspositionKey];

//            Preform alpha beta pruning with transposition table
            if (tableMove.flag == TranspositionEnum.EXACT)
}
                return tableMove.move;
{
            else if (tableMove.flag == TranspositionEnum.LOWERBOUND)
}
                alpha = Math.Max(alpha, tableMove.move.Heuristic);
{
            else if (tableMove.flag == TranspositionEnum.UPPERBOUND)
}
                beta = Math.Min(beta, tableMove.move.Heuristic);
{
            if (alpha >= beta)
}
                return tableMove.move;
{
{
```

```
//          Terminating condition. i.e leaf node is reached OR the game is over
        if ((depth <= 0) || (m.CheckGameState() != GameState.OnGoing))
}
//            Program.UI.DrawBoard(m.GameBoard);
          return new MoveAI(0, new List<PieceMove>(), Heuristic(m, currentPlayer, KingWeight, SoliderWeight,
PossibleMoveWeight, WinScore, LoseScore, DrawScore));
{

//          Define Transposition flag used to do alpha beta pruning with transposition table values
        TranspositionEnum Transflag;

//          Save alpha value at start of function
        int alphaorigin = alpha;

//          Color of next player (for next negamax iteration)
        PlayerColor nextPlayer;

//          Save the model as it was before changes
        Model SaveM = new Model;()
        m.CopyModel(SaveM);

//          Keeps details of turn made
        MoveAI turn;

//          List of all moveable pieces by the the current player
        List<UInt32> MoveAblePiecesList;

//          Bestvalue out of all possible moves return values
        MoveAI BestVal = new MoveAI(0, new List<PieceMove>(), NegInfinity);

//          Get all moveable pieces of current player
        switch (currentPlayer)
}
```

```
            case (PlayerColor.BLACK):
                MoveAblePiecesList = m.GetMoveAblePieces(m.GameBoard.BP);
                nextPlayer = PlayerColor.WHITE;
                break;
            case (PlayerColor.WHITE):
                MoveAblePiecesList = m.GetMoveAblePieces(m.GameBoard.WP);
                nextPlayer = PlayerColor.BLACK;
                break;
            default:
                throw new Exception("Player color not supported");
{

//          Go over all possiblemoves and check them out
        foreach (UInt32 MoveAblePiece in MoveAblePiecesList)
}
            foreach (PieceMove WherePiece in m.GetPossibleMoves(currentPlayer, MoveAblePiece))
}



                List<List<PieceMove>> ListOfTurnMoves = new List<List<PieceMove;()<<
                List<PieceMove> tempList = new List<PieceMove;()<

//                  Create a duplicate for getting possible moves from
                Model DuplicateM = new Model;()
                m.CopyModel(DuplicateM);

                GetPossibleMovesTurn(currentPlayer, DuplicateM, WherePiece, MoveAblePiece, tempList,
ListOfTurnMoves);

//                  Get Best value possible on perticular piece moving to a perticular index
                foreach (List<PieceMove> MoveSequence in ListOfTurnMoves)
}
```

```
//                Change module according to move sequence
            m.MakeMove(currentPlayer, MoveAblePiece, MoveSequence[0]);


//                Problem here the length of move sequence is not as expected
            for (int moveIndex = 1; moveIndex < MoveSequence.Count; moveIndex++)
}
                m.MakeMove(currentPlayer, MoveSequence[moveIndex - 1].Where,
MoveSequence[moveIndex]);
{


            turn = new MoveAI(MoveAblePiece, MoveSequence, -NegaMax(depth - 1, nextPlayer, m, -beta, -
alpha).Heuristic);


//                Undo move by copying m back to how it was
            SaveM.CopyModel(m);


//                Bestval = max(turn, Bestval)
            if (turn.Heuristic > BestVal.Heuristic)
}
                BestVal = turn;
{


//                alpha = max(alpha, Bestval)
            alpha = Math.Max(alpha, BestVal.Heuristic);


//                alpha beta pruning
            if (alpha >= beta)
}
                break;
{


{
//              alpha beta pruning
          if (alpha >= beta)
```

```
}
                break;
{
{
 //          alpha beta pruning
          if (alpha >= beta)
}
              break;
{
{

 //        Set up lookup table with alphabeta pruning of current turn
        if (BestVal.Heuristic <= alphaorigin)
           Transflag = TranspositionEnum.UPPERBOUND;
        else if (BestVal.Heuristic >= beta)
           Transflag = TranspositionEnum.LOWERBOUND;
        else
           Transflag = TranspositionEnum.EXACT;

 //        This is the move to make at given depth for the bot, goes into transposition table with the depth it
describes
        TranspositionTable[TranspositionKey] = new TranspositionValue(BestVal, depth, Transflag);

        return BestVal;
{

> ///      summary<
 ///      Puts in ListOfTurnMoves given to it all the possible plays (chain eats and regular moves) that can be
made with the given MoveAblePiece to the specific WherePiece
 ///      Using Recursion makes list of every available combination of paths in the chain eating "tree/graph "
/> ///      summary<
> ///      param name="m"> The module representing the state of the game currently </param<
> ///      param name="WherePiece"> Where to move the piece to </param<
> ///      param name="MoveAblePiece"> The piece to move </param<
```

```
> ///     param name="MovesList"> List Of current path of chain eating </param<
> ///     param name="ListOfTurnMoves"> List of all combinations of chain eating/moving routes </param<
> ///     param name="Player"> The current player's color </param<
    public void GetPossibleMovesTurn(PlayerColor Player, Model m, PieceMove WherePiece, UInt32
MoveAblePiece, List<PieceMove> MovesList, List<List<PieceMove>> ListOfTurnMoves)
}
        List<PieceMove> tempList;

//          If next move cannot be chain eaten, save current move and terminate recursion
        if (!m.MakeMove(Player, MoveAblePiece, WherePiece))
}
//              Add where to piece moving route
            MovesList.Add(WherePiece);

//              save copy of MovesList (dont want to save the same pointer every time)
            tempList = ListFuncs<PieceMove>.CloneList(MovesList);
            ListOfTurnMoves.Add(tempList);

            return;
{

//          Save the model as it was before changes
        Model SaveM = new Model;()
        m.CopyModel(SaveM);

//          Add where to piece moving route
        MovesList.Add(WherePiece);

//          save copy of MovesList (dont want to save the same pointer every time)
        tempList = ListFuncs<PieceMove>.CloneList(MovesList);
        ListOfTurnMoves.Add(tempList);

//          Save the MovesList as it was before adding moves
```

```
        List<PieceMove> SaveList = ListFuncs<PieceMove>.CloneList(MovesList);

//          Update the MoveAblePiece to where it has been moved
        MoveAblePiece = WherePiece.Where;

//          All possible Next moves for given new location of piece
        List<PieceMove> PossibleNextEats = m.GetPossibleMoves(Player, MoveAblePiece);

//          all possible eats in possiblemoves list for chain eating
        PossibleNextEats.RemoveAll(move => move.Eat == false);

//          No further eats in the chain, terminate recursion
        if (PossibleNextEats.Count == 0)
}
           return;
{

//          Go over each PossibleEat and finds its full eating route
        foreach (PieceMove tempWherePiece in PossibleNextEats)
}
//              Recurse until eating route ends
           GetPossibleMovesTurn(Player, m, tempWherePiece, MoveAblePiece, MovesList, ListOfTurnMoves);

//              restore moveslist
           MovesList = ListFuncs<PieceMove>.CloneList(SaveList);

//              restore model
           SaveM.CopyModel(m);
{

{
{
{
```

```
using System;

using System.Collections.Generic;


namespace Checkers

}

> ///    summary<

 ///    Contains Delegeate definition for a heuritstic function and all heuristic functions made

/> ///    summary<

   static class Heuristics

}


> ///       summary<

 ///       Template for herustic function (also Heuristic function type

 ///       Gets the Model representing the game and the color of the bot and returns an integer scoring how good the situation is for the bot

/> ///       summary<

      public delegate int HeuristicFunc(Model m, PlayerColor currentPlayer, int KingRatio, int SoliderRatio, int PossibleMoveRatio, int WinScore, int LoseScore, int DrawScore);



> ///       summary<

 ///       Score the game situation when called in favor of the bot, Basic heuristic modifyable by user (only uses 6 basic parameters)

/> ///       summary<

> ///       param name="m"> The Model representing the game </param<

> ///       param name="currentPlayer"> The current player to score the points of </param<

> ///       param name="KingRatio"> Importance of king piece parameter </param<

> ///       param name="SoliderRatio"> Importance of solider piece parameter </param<

> ///       param name="PossibleMovesRatio"> Importance of a possible move that can be made by the user after the turn parameter </param<

> ///       param name="WinScore"> The Score given to winning the game </param<

> ///       param name="LoseScore"> The score given to Losing the game</param<

> ///       param name="DrawScore"> The score give to a draw in the game </param<

> ///       returns> a score suggesting how good the situation is for the bot (positive is good, negative is bad) </returns<
```

```
    public static int BasicHeuristic(Model m, PlayerColor currentPlayer, int KingRatio, int SoliderRatio, int
PossibleMovesRatio, int WinScore, int LoseScore, int DrawScore)
}
//      The score of the game at its current situation
        int Score = 0,
          SoliderScore, KingScore;

//      The color of the currentPlayer's enemy
        PlayerColor enemyPlayer;

        switch (currentPlayer)
}
//          player is black player
          case (PlayerColor.BLACK):

//              Set enemyplayer color
            enemyPlayer = PlayerColor.WHITE;

//              Total soliders bot has alive minus total soliders player has alive
            SoliderScore = UInt32Funcs.CountSetBits(m.GameBoard.BP.PB ^ (m.GameBoard.K &
m.GameBoard.BP.PB))-
              UInt32Funcs.CountSetBits(m.GameBoard.WP.PB ^ (m.GameBoard.K & m.GameBoard.WP.PB));

//              Total kings bot has alive minus total pieces player has alive
            KingScore = UInt32Funcs.CountSetBits(m.GameBoard.BP.PB & m.GameBoard.K) -
UInt32Funcs.CountSetBits(m.GameBoard.WP.PB & m.GameBoard.K);

//              Check the game state and score accordingly
            switch (m.CheckGameState())
}
              case (GameState.Draw):
                Score += DrawScore;
                break;
```

```
                case (GameState.WhiteWon):
                    Score += LoseScore;
                    break;

                case (GameState.BlackWon):
                    Score += WinScore;
                    break;

{
            break;

//          player is white player
        case (PlayerColor.WHITE):

//              Set enemyplayer color
            enemyPlayer = PlayerColor.BLACK;

//              Total soliders bot has alive minus total soliders player has alive
            SoliderScore = UInt32Funcs.CountSetBits(m.GameBoard.WP.PB ^ (m.GameBoard.K &
m.GameBoard.WP.PB)) - UInt32Funcs.CountSetBits(m.GameBoard.BP.PB ^ (m.GameBoard.K &
m.GameBoard.BP.PB));

//              Total kings bot has alive minus total pieces player has alive
            KingScore = UInt32Funcs.CountSetBits(m.GameBoard.WP.PB & m.GameBoard.K) -
UInt32Funcs.CountSetBits(m.GameBoard.BP.PB & m.GameBoard.K);

//              Check the game state and score accordingly
            switch (m.CheckGameState())
}
                case (GameState.Draw):
                    Score += DrawScore;
                    break;

                case (GameState.BlackWon):
```

```
                Score += LoseScore;
                break;

            case (GameState.WhiteWon):
                Score += WinScore;
                break;

{
            break;

        default:
            throw new Exception("Player color not supported");
{

 //         Multiplied by each respective ratio
        Score += KingScore * KingRatio + SoliderScore * SoliderRatio;

 //         Add possible moves that can be made by given player minus possible moves that can be made by
enemy player
        Score += (CountPossibleMoves(m, currentPlayer) - CountPossibleMoves(m, enemyPlayer)) *
PossibleMovesRatio;

        return Score;
{

> ///       summary<
 ///       Score the game situation when called in favor of the bot, Extreme heuristic uses a lot more paremeters
and takes into consideration more factors
 ///       at the cost of being slower, used for extreme gamemode and bot level (Parameters given dont matter)
/> ///       summary<
> ///       param name="m"> The Model representing the game </param<
> ///       param name="currentPlayer"> The current player to score the points of  </param<
> ///       param name="KingRatio"> Importance of king piece parameter </param<
```

```
> ///     param name="SoliderRatio"> Importance of solider piece parameter </param<

> ///     param name="PossibleMovesRatio"> Importance of a possible move that can be made by the user
after the turn parameter </param<

> ///     param name="WinScore"> The Score given to winning the game </param<

> ///     param name="LoseScore"> The score given to Losing the game</param<

> ///     param name="DrawScore"> The score give to a draw in the game </param<

> ///     returns> a score suggesting how good the situation is for the bot (positive is good, negative is bad)
</returns<

    public static int ExtremeHeuristic(Model m, PlayerColor currentPlayer, int KingRatio, int SoliderRatio, int
PossibleMovesRatio, int WinScore, int LoseScore, int DrawScore)

}

        DrawScore = -400;

        LoseScore = -500;

        WinScore = 500;

        PossibleMovesRatio = 2;

        SoliderRatio = 5;

        KingRatio = 10;

        int DefendingPromotionLine = 3, KingSafeFromEating = 2, SoliderSafeFromEating = 1;



//          The score of the game at its current situation

        int Score = 0,

           SoliderScore, KingScore;


//          The color of the currentPlayer's enemy

        PlayerColor enemyPlayer;


        switch (currentPlayer)

}

//              player is black player

           case (PlayerColor.BLACK):


//                Set enemyplayer color

              enemyPlayer = PlayerColor.WHITE;
```

```
//          Total soliders bot has alive minus total soliders player has alive
        SoliderScore = UInt32Funcs.CountSetBits(m.GameBoard.BP.PB ^ (m.GameBoard.K &
m.GameBoard.BP.PB))-
            UInt32Funcs.CountSetBits(m.GameBoard.WP.PB ^ (m.GameBoard.K & m.GameBoard.WP.PB));

//          Total kings bot has alive minus total pieces player has alive
        KingScore = UInt32Funcs.CountSetBits(m.GameBoard.BP.PB & m.GameBoard.K) -
UInt32Funcs.CountSetBits(m.GameBoard.WP.PB & m.GameBoard.K);

//          Check the game state and score accordingly
        switch (m.CheckGameState())
}
            case (GameState.Draw):
                Score += DrawScore;
                break;

            case (GameState.WhiteWon):
                Score += LoseScore;
                break;

            case (GameState.BlackWon):
                Score += WinScore;
                break;

{

//          Add how many pieces are safe from being ate (sitting on edge of row) minus how many enemy
pieces are safe from being eaten
        Score += (CountSafePawns(m.GameBoard.BP.PB ^ (m.GameBoard.K & m.GameBoard.BP.PB)) -
CountSafePawns(m.GameBoard.WP.PB ^ (m.GameBoard.K & m.GameBoard.WP.PB))) *
SoliderSafeFromEating;
        Score += (CountSafePawns(m.GameBoard.BP.PB & m.GameBoard.K) -
CountSafePawns(m.GameBoard.WP.PB & m.GameBoard.K)) * KingSafeFromEating;
```

```
            break;

//          player is white player
        case (PlayerColor.WHITE):

//              Set enemyplayer color
            enemyPlayer = PlayerColor.BLACK;

//              Total soliders bot has alive minus total soliders player has alive
            SoliderScore = UInt32Funcs.CountSetBits(m.GameBoard.WP.PB ^ (m.GameBoard.K &
m.GameBoard.WP.PB)) - UInt32Funcs.CountSetBits(m.GameBoard.BP.PB ^ (m.GameBoard.K &
m.GameBoard.BP.PB));

//              Total kings bot has alive minus total pieces player has alive
            KingScore = UInt32Funcs.CountSetBits(m.GameBoard.WP.PB & m.GameBoard.K) -
UInt32Funcs.CountSetBits(m.GameBoard.BP.PB & m.GameBoard.K);

//              Check the game state and score accordingly
            switch (m.CheckGameState())
}
                case (GameState.Draw):
                    Score += DrawScore;
                    break;

                case (GameState.BlackWon):
                    Score += LoseScore;
                    break;

                case (GameState.WhiteWon):
                    Score += WinScore;
                    break;

{
```

```
 //                Add how many pieces are safe from being ate (sitting on edge of row) minus how many enemy
pieces are safe from being eaten

                Score += (CountSafePawns(m.GameBoard.WP.PB ^ (m.GameBoard.K & m.GameBoard.WP.PB)) -
CountSafePawns(m.GameBoard.BP.PB ^ (m.GameBoard.K & m.GameBoard.BP.PB))) * SoliderSafeFromEating;

                Score += (CountSafePawns(m.GameBoard.WP.PB & m.GameBoard.K) -
CountSafePawns(m.GameBoard.BP.PB & m.GameBoard.K)) * KingSafeFromEating;


                break;


            default:

                throw new Exception("Player color not supported");

{


 //          Multiplied by each respective ratio

        Score += KingScore * KingRatio + SoliderScore * SoliderRatio;


 //          Add possible moves that can be made by given player minus possible moves that can be made by
enemy player

        Score += (CountPossibleMoves(m, currentPlayer) - CountPossibleMoves(m, enemyPlayer)) *
PossibleMovesRatio;


 //          Add how many pieces are defending the promotion line minus how many pieces are defending the
enemy promotion line

        Score += (CountPromotionLine(m, currentPlayer) - CountPromotionLine(m, enemyPlayer)) *
DefendingPromotionLine;


        return Score;

{


> ///       summary<

 ///       Score the game situation in favor of the bot in an adaptive manner, for example the less pieces a
player has the more valueable the pieces are to the player (Parameters given dont matter)

/> ///       summary<

> ///       param name="m"> The Model representing the game </param<

> ///       param name="currentPlayer"> The current player to score the points of </param<

> ///       param name="KingRatio"> Importance of king piece parameter </param<
```

```
> ///     param name="SoliderRatio"> Importance of solider piece parameter </param<

> ///     param name="PossibleMovesRatio"> Importance of a possible move that can be made by the user
after the turn parameter </param<

> ///     param name="WinScore"> The Score given to winning the game </param<

> ///     param name="LoseScore"> The score given to Losing the game</param<

> ///     param name="DrawScore"> The score give to a draw in the game </param<

> ///     returns> a score suggesting how good the situation is for the bot (positive is good, negative is bad)
</returns<

    public static int AdaptiveHeuristic(Model m, PlayerColor currentPlayer, int KingRatio, int SoliderRatio, int
PossibleMovesRatio, int WinScore, int LoseScore, int DrawScore)

}

        DrawScore = -4000;

        LoseScore = -5000;

        WinScore = 5000;

        PossibleMovesRatio = 8;

        SoliderRatio = 5;

        KingRatio = 10;


//          The score of the game at its current situation

        int Score = 0;

        float

           WhiteSoliderScore = UInt32Funcs.CountSetBits(m.GameBoard.WP.PB ^ (m.GameBoard.K &
m.GameBoard.WP.PB))

,            WhiteKingScore = UInt32Funcs.CountSetBits(m.GameBoard.WP.PB & m.GameBoard.K)

,            BlackSoliderScore = UInt32Funcs.CountSetBits(m.GameBoard.BP.PB ^ (m.GameBoard.K &
m.GameBoard.BP.PB))

,            BlackKingScore = UInt32Funcs.CountSetBits(m.GameBoard.BP.PB & m.GameBoard.K);


//          [PlayerColor]State Is the ratio representing how much a piece is worth to the bot which depends on
how many pieces the bot has left

        float WhiteState = (12 / (WhiteSoliderScore + WhiteKingScore + 1)), BlackState = (12 /
(BlackSoliderScore + BlackKingScore + 1));

        WhiteSoliderScore *= WhiteState;

        WhiteKingScore *= WhiteState;

        BlackSoliderScore *= BlackState;

        BlackKingScore *= BlackState;
```

```
//          The color of the currentPlayer's enemy
        PlayerColor enemyPlayer;

        switch (currentPlayer)
}
//            player is black player
            case (PlayerColor.BLACK):

//                Set enemyplayer color
                enemyPlayer = PlayerColor.WHITE;

//                Add King + Solider Scores with their ratios
                Score += (int)((BlackKingScore - WhiteKingScore) * KingRatio + (BlackSoliderScore -
WhiteSoliderScore) * SoliderRatio);

//                Check the game state and score accordingly
                switch (m.CheckGameState())
}
                    case (GameState.Draw):
                        Score += DrawScore;
                        break;

                    case (GameState.WhiteWon):
                        Score += LoseScore;
                        break;

                    case (GameState.BlackWon):
                        Score += WinScore;
                        break;

{
```

```
                break;

//              player is white player
            case (PlayerColor.WHITE):

//                  Set enemyplayer color
                enemyPlayer = PlayerColor.BLACK;

//                  Add King + Solider Scores with their ratios
                Score += (int)((WhiteKingScore - BlackKingScore) * KingRatio + (WhiteSoliderScore -
BlackSoliderScore) * SoliderRatio);

//                  Check the game state and score accordingly
                switch (m.CheckGameState())
}
                    case (GameState.Draw):
                        Score += DrawScore;
                        break;

                    case (GameState.BlackWon):
                        Score += LoseScore;
                        break;

                    case (GameState.WhiteWon):
                        Score += WinScore;
                        break;

{
                break;

            default:
                throw new Exception("Player color not supported");
{
```

```
//          Add possible moves that can be made by given player minus possible moves that can be made by
enemy player

        Score += (CountPossibleMoves(m, currentPlayer) - CountPossibleMoves(m, enemyPlayer)) *
PossibleMovesRatio;


        return Score;

{


> ///       summary<

///       Counts how many legal moves the given player can preform

/> ///       summary<

> ///       param name="m"> The game model to check </param<

> ///       param name="color"> the given player color </param<

> ///       returns> the number of legal moves the given player can preform </returns<

      private static int CountPossibleMoves(Model m, PlayerColor color)

}

//          List of all moveable pieces by the bot

        List<UInt32> MoveAblePiecesList;

//          The number of legal moves the bot can make (doesn't count all diffrent types of chain eating moves)

        int NumOfLegalMoves = 0;


//          Get the list of all moveale pieces by the given bot playerColor

        switch (color)

}

            case (PlayerColor.BLACK):

                MoveAblePiecesList = m.GetMoveAblePieces(m.GameBoard.BP);

                break;

            case (PlayerColor.WHITE):

                MoveAblePiecesList = m.GetMoveAblePieces(m.GameBoard.WP);

                break;

            default:

                throw new Exception("Player color not supported");

{
```

```
//          Go over all possiblemoves and update the legal moves counter accordingly
        foreach (UInt32 MoveAblePiece in MoveAblePiecesList)
}
            NumOfLegalMoves += m.GetPossibleMoves(color, MoveAblePiece).Count;
{

        return NumOfLegalMoves;
{

> ///       summary<
 ///       Count how many pieces are on the player's promotion line
/> ///       summary<
> ///       param name="m"> the game model holding the board </param<
> ///       param name="player"> the player color </param<
> ///       returns></returns<
      private static int CountPromotionLine(Model m, PlayerColor player)
}
 //          Counter of how many pieces are on the promition line
        int counterPiecesOnLine = 0;

 //          Act differently depending on player
        switch (player)
}
 //             Need to count right most 4 bits on black board
            case (PlayerColor.BLACK):

                UInt32 checkMaskBlack = 0b_00000000000000000000000000000001;
                for(int Piece = 0; Piece < 4; Piece++)
}
                    if((m.GameBoard.BP.PB & checkMaskBlack) != 0)
}
                        counterPiecesOnLine;++
```

```
{

//                Move mask 1 bit to the left
              checkMaskBlack <<= 1;
{
            break;

//          Need to count left most 4 bits on white board
         case (PlayerColor.WHITE):
            UInt32 checkMaskWhite = 0b_10000000000000000000000000000000;
            for (int Piece = 0; Piece < 4; Piece++)
}
              if ((m.GameBoard.WP.PB & checkMaskWhite) != 0)
}
                counterPiecesOnLine;++
{

//                Move mask 1 bit to the left
              checkMaskWhite >>= 1;
{
            break;

         default:
            throw new Exception("Player color not supported");
{

      return counterPiecesOnLine;

{

> ///      summary<
 ///      Count how many pawns on the given board are safe (cannot be eaten)
/> ///      summary<
```

```
> ///      param name="Board"> Uint32 Representing the board of the player </param<
> ///      returns> how many pawns are safe and cannot be eaten </returns<
      private static int CountSafePawns(UInt32 Board)
}
//          counts how many pawns are in the corner and cannot be eaten therefor are safe
          int safePawnCounter = 0;


          UInt32 safePawnMask = 0b_00000000000000000000000000001000;


//          Go over board and count safe pawns, go couple of rows at a time one odd and one even (rows are
going form 0 - 7)
          for (int rowCoupleIndex = 0; rowCoupleIndex<4; rowCoupleIndex++)
}
//             Even row left most is safe
            if((Board & safePawnMask) != 0)
}
               safePawnCounter;++
{


            safePawnMask <<= 1;


//             Odd row right most is safe
            if ((Board & safePawnMask) != 0)
}
               safePawnCounter;++
{


            safePawnMask <<= 7;
{


          return safePawnCounter;
{
{
```

```
{

using System;
using System.Collections.Generic;

namespace Checkers
{
   /// <summary>
   /// Struct Used to represent an AI move (contains its heuristic value and the details of the move itself)
   /// </summary>
   public struct MoveAI
   {
      /// <summary>
      /// Constructor
      /// </summary>
      /// <param name="Piece"> piece to be moved </param>
      /// <param name="Where"> list of where to move the piece in current turn </param>
      /// <param name="Heuristic"> score of herustic function on game board </param>
      public MoveAI(UInt32 Piece, List<PieceMove> Where, int Heuristic)
      {
         this.Piece = Piece;
         // Copy by value not by refrence
         this.Where = ListFuncs<PieceMove>.CloneList(Where);
         this.Heuristic = Heuristic;
      }

      /// <summary>
      /// What piece to move
      /// </summary>
      public UInt32 Piece { get; }

      /// <summary>
      /// List of where to move the piece in the current turn
      /// </summary>
      public List<PieceMove> Where { get; }

      /// <summary>
      /// Score of the heuristic function on the game board
      /// </summary>
      public int Heuristic { get; }
   }
}



namespace Checkers
{
   /// <summary>
   /// Enum used for flag and alpha beta with transposition table
   /// </summary>
   public enum TranspositionEnum
   {
      EXACT,
      LOWERBOUND,
      UPPERBOUND
   }
```

```
}


namespace Checkers
{
   /// <summary>
   /// Describes transposition table value
   /// </summary>
   public struct TranspositionValue
   {
      /// <summary>
      /// Move to make in given board situation
      /// </summary>
      public MoveAI move;

      /// <summary>
      /// Depth this move was calculated at
      /// </summary>
      public int depth;

      /// <summary>
      /// flag used to help with alpha beta pruning for transposition table
      /// </summary>
      public TranspositionEnum flag;

      /// <summary>
      /// Constructor
      /// </summary>
      /// <param name="move"> Move to make in given board situation </param>
      /// <param name="depth"> Depth this move was calculated at </param>
      /// <param name="flag"> flag used to help with alpha beta pruning for transposition table </param>
      public TranspositionValue(MoveAI move, int depth, TranspositionEnum flag)
      {
         this.move = move;
         this.depth = depth;
         this.flag = flag;
      }
   }
}
using System;

namespace Checkers
{

   /// <summary>
   /// contains both players and initiates them and also contains king bit board
   /// </summary>
   public class Board
   {

      /// <summary>
      /// the length of a row on the board (how many bits represent each row)
      /// </summary>
      public readonly int RowLength = 4;


      /// <summary>
```

```
        /// a variable representing the white player (the player's piece locations, color and first row)
        /// </summary>
        public Player WP = new Player(0b_11111111111100000000000000000000, PlayerColor.WHITE,
0b_00010000000000000000000000000000, 0b_10000000000000000000000000000000); // 1111
                                                                                                                    // 1111
                                                                                                                    // 1111
                                                                                                                    // 0000
                                                                                                                    // 0000
                                                                                                                    // 0000
                                                                                                                    // 0000
                                                                                                                    // 0000


        /// <summary>
        /// a variable representing the black player (the player's piece locations, color and first row)
        /// </summary>
        public Player BP = new Player(0b_00000000000000000000111111111111, PlayerColor.BLACK,
0b_00000000000000000000000000000001, 0b_00000000000000000000000000001000); // 0000
                                                                                                                    // 0000
                                                                                                                    // 0000
                                                                                                                    // 0000
                                                                                                                    // 0000
                                                                                                                    // 1111
                                                                                                                    // 1111
                                                                                                                    // 1111


        /// <summary>
        /// a bit board representing the location of all kings alive ( 1 is a king , 0 is empty)
        /// </summary>
        public UInt32 K = 0b_00000000000000000000000000000000; // 0000
                                                // 0000
                                                // 0000
                                                // 0000
                                                // 0000
                                                // 0000
                                                // 0000
                                                // 0000

    }
}


namespace Checkers
{
    /// <summary>
    /// enum of all different game states possible (DRAW, BLACKWON, WHITEWON, ONGOING)
    /// </summary>
    public enum GameState
    {
        Draw,
        BlackWon,
        WhiteWon,
        OnGoing
    }
}
using System;
```

```
using System.Collections.Generic;


namespace Checkers

}


> ///    summary<

 ///    class containing the game's logic

/> ///    summary<

   public class Model

}

> ///       summary<

 ///       constant representing how many turns without a piece being eaten can pass until the game is declared
a draw

/> ///       summary<

       public readonly int TurnsToDraw = 40;



> ///       summary<

 ///       holds how many turns its been since a piece on the board was eaten

/> ///       summary<

       public int LastTurnEaten = 0;



> ///       summary<

 ///       holds the game board

/> ///       summary<

       public Board GameBoard { get; private set; }



> ///       summary<

 ///       initiates model properties

/> ///       summary<

       public Model()
```

```
}
        GameBoard = new Board;()
{



> ///     summary<
 ///     Copys this model into given model
/> ///     summary<
> ///     param name="newM"> Other given model </param<
    public void CopyModel(Model newM)
}
 //         Copy Kings
        newM.GameBoard.K = GameBoard.K;

 //         Copy Black player
        newM.GameBoard.BP.Color = GameBoard.BP.Color;
        newM.GameBoard.BP.PB = GameBoard.BP.PB;
        newM.GameBoard.BP.FirstRowStart = GameBoard.BP.FirstRowStart;
        newM.GameBoard.BP.FirstRowEnd = GameBoard.BP.FirstRowEnd;

 //         Copy White player
        newM.GameBoard.WP.Color = GameBoard.WP.Color;
        newM.GameBoard.WP.PB = GameBoard.WP.PB;
        newM.GameBoard.WP.FirstRowStart = GameBoard.WP.FirstRowStart;
        newM.GameBoard.WP.FirstRowEnd = GameBoard.WP.FirstRowEnd;

 //         Others
        newM.LastTurnEaten = LastTurnEaten;

{



> ///     summary<
```

```
 ///      finds the current game state
/> ///      summary<
> ///      returns> returns the current game state, whether its a draw/a player has won/the game is still
ongoing </returns<
      public GameState CheckGameState()
}
 //          if its been 30 turns since someone ate a piece or if both players have no  pieces left its a DRAW
         if ((LastTurnEaten >= TurnsToDraw) || (GameBoard.BP.PB == 0 && GameBoard.WP.PB == 0))
            return GameState.Draw;

 //          if black player has no pieces left OR black player cannot move any piece white won
         if ((GameBoard.BP.PB == 0) || GetMoveAblePieces(this.GameBoard.BP).Count == 0)
            return GameState.WhiteWon;

 //          if white player has no pieces left OR white payer cannot move any piece black won
         if ((GameBoard.WP.PB == 0) || GetMoveAblePieces(this.GameBoard.WP).Count == 0)
            return GameState.BlackWon;

 //          game is not over yet, no one has won and its not a draw
         return GameState.OnGoing;
{



> ///      summary<
 ///      Gets a list of all moveable pieces by given player
/> ///      summary<
> ///      param name="currentPlayer"> The current player </param<
> ///      returns> List of all moveable pieces by given player </returns<
      public List<UInt32> GetMoveAblePieces(Player currentPlayer)
}
         List<UInt32> moveAblePieces = new List<UInt32;()<

 //          Get all pieces
```

```
        for (int piece = 0; piece < 32; piece++)
}
          moveAblePieces.Add(currentPlayer.GetPiece(piece));
{

//        Remove all 0 where there are no pieces
        moveAblePieces.RemoveAll(piece => piece == 0);

//        Remove all pieces that cannot move
        moveAblePieces.RemoveAll(piece => GetPossibleMoves(currentPlayer.Color, piece).Count == 0);

        return moveAblePieces;
{



> ///      summary<
 ///      kings given piece at given location
/> ///       summary<
> ///      param name="kinged"> piece of the player to king </param<
      private void MakeKing(UInt32 kinged)
}
        GameBoard.K |= kinged;
{



> ///      summary<
 ///      gets the board belonging to the given player
/> ///       summary<
> ///      param name="player"> black or white the player's color </param<
> ///      returns> returns the board belonging to the given player color </returns<
      public Player GetPlayerBoard(PlayerColor player)
}
        return player switch
```

```
}
            (PlayerColor.WHITE) => GameBoard.WP,
            (PlayerColor.BLACK) => GameBoard.BP,
 <= _            throw new Exception("Invalid player"),
;{
{



> ///       summary<
 ///       makes a move on the board
/> ///       summary<
> ///       param name="player"> PlayerColor.WHITE/PlayerColor.BLACK </param<
> ///       param name="piece"> piece is an UInt32 with all bits 0 except for the one representing the piece the player wants to move </param<
> ///       param name="whereMask"> whereMask is an UInt32 with all bits 0 except for the one representing where the player wants the piece to move </param<
> ///       returns> returns true if current move was an eat and there is another possible eat for the same piece in the next move (so if chain eating is possible returns true) otherwise returns false </returns<
      public bool MakeMove(PlayerColor player, UInt32 piece, PieceMove whereMask)
}


         switch (player)
 }
            case (PlayerColor.WHITE):


                GameBoard.WP.PB ^= piece; // turns the bit at the location of the piece from 1 to 0 (indicating there is no longer a piece there)
                GameBoard.WP.PB |= whereMask.Where; // turns the bit at the location of where you move the piece from 0 to 1 (indicating there is now a piece there)


 //                reached first row of black (enemy) player, kinged piece
                if (whereMask.Where <= GameBoard.BP.FirstRowEnd && whereMask.Where >= GameBoard.BP.FirstRowStart)
}
                MakeKing(whereMask.Where);
```

```
{

//          if piece is a king change king board as well
        if ((GameBoard.K & piece) != 0)
}
            GameBoard.K ^= piece;
            GameBoard.K |= whereMask.Where;
{

//          if the move is an eating move
        if (whereMask.Eat)
}
            LastTurnEaten = 0;
            GameBoard.BP.PB ^= whereMask.WhereEaten;

//            If eaten piece was a king need to reset it on king board
            if((GameBoard.K & whereMask.WhereEaten) !=0)
              GameBoard.K ^= whereMask.WhereEaten;

//            get list of all possible move after eating move
            List<PieceMove> futureMoves = GetPossibleMoves(PlayerColor.WHITE, whereMask.Where);

//            check if any of the possible moves is an eat move
            bool possibleFutureEat = false;
            foreach (PieceMove move in futureMoves)
}
              if (move.Eat)
}
                possibleFutureEat = true;
{
{
```

```
//                returns true if there is a possible move in the future that is an eating move and this move was
also an eating move so chain eating is possible

                return possibleFutureEat;

{


            else

}

                LastTurnEaten += 1;

                return false;

{


        case (PlayerColor.BLACK):


            GameBoard.BP.PB ^= piece; // turns the bit at the location of the piece from 1 to 0 (indicating there
is no longer a piece there)

            GameBoard.BP.PB |= whereMask.Where; // turns the bit at the location of where you move the
piece from 0 to 1 (indicating there is now a piece there)


//              reached first row of white (enemy) player, kinged piece

            if (whereMask.Where <= GameBoard.WP.FirstRowEnd && whereMask.Where >=
GameBoard.WP.FirstRowStart)

}

                MakeKing(whereMask.Where);

{


//              if piece is a king change king board as well

            if ((GameBoard.K & piece) != 0)

}

                GameBoard.K ^= piece;

                GameBoard.K |= whereMask.Where;

{


//              if the move is an eating move

            if (whereMask.Eat)
```

```
}
                LastTurnEaten = 0;
                GameBoard.WP.PB ^= whereMask.WhereEaten;

//                  If eaten piece was a king need to reset it on king board
                if ((GameBoard.K & whereMask.WhereEaten) != 0)
                  GameBoard.K ^= whereMask.WhereEaten;

//                  get list of all possible move after eating move
                List<PieceMove> futureMoves = GetPossibleMoves(PlayerColor.BLACK, whereMask.Where);

//                  check if any of the possible moves is an eat move
                bool possibleFutureEat = false;
                foreach (PieceMove move in futureMoves)
}
                  if (move.Eat)
}
                    possibleFutureEat = true;
{
{

//                  returns true if there is a possible move in the future that is an eating move and this move was
also an eating move so chain eating is possible
                return possibleFutureEat;
{

              else
}
                LastTurnEaten += 1;
                return false;
{

            default:
```

```
                throw new Exception("Invalid player");
{

{

> ///      summary<
 ///      finds all the possible locations a certain piece can be moved to for given player
/> ///      summary<
> ///      param name="player"> PlayerColor.WHITE/PlayerColor.BLACK </param<
> ///      param name="piece"> piece is an UInt32 with all bits 0 except for the one representing the piece the player wants to move (a mask) </param<
> ///      returns> List of all possible moves the player can make </returns<
      public List<PieceMove> GetPossibleMoves(PlayerColor player, UInt32 piece)
}

         return player switch
}
            (PlayerColor.WHITE) => PossibleMovesSpecificPlayer(piece, GameBoard.WP.PB, GameBoard.BP.PB, PlayerColor.WHITE, 4, 3, 5, (piece, bits) => { return piece >> bits; }, (piece, bits) => { return piece << bits; }),
            (PlayerColor.BLACK) => PossibleMovesSpecificPlayer(piece, GameBoard.BP.PB, GameBoard.WP.PB, PlayerColor.BLACK, 4, 5, 3, (piece, bits) => { return piece << bits; }, (piece, bits) => { return piece >> bits; }),
 <= _            throw new Exception("Invalid player"),
;{
{

> ///      summary<
 ///      Gets the possible moves for given piece of current player by using bit operations
/> ///      summary<
> ///      param name="piece"> given piece represented as a sequence of bits </param<
> ///      param name="playerBoard"> the current player's bit board </param<
> ///      param name="enemyBoard"> the enemy player's bit board </param<
> ///      param name="player"> the player's color PlayerColor.WHITE/PlayerColor.BLACK </param<
```

```
> ///     param name="nextRowMove"> the amount of bits needed for basic next row move </param<

> ///     param name="nextRowDiagonalMoveEven"> the amount of bits needed for additional next row move ( a diagonal move) if the current row the piece is on is even </param<

> ///     param name="nextRowDiagonalMoveOdd"> the amount of bits needed for additional next row move ( a diagonal move) if the current row the piece is on is odd </param<

> ///     param name="op"> type of operation to preform on the piece with given bits to find a possible move (shift operation - shift left/shift right) </param<

> ///     param name="kingBackOp"> type of operation to preform on the piece with given bits to find a possible move (shift operation- shift left/right) the opposite operation of

 ///     op used for king piece to move backwards </param<

> ///     returns> Possible moves of given piece of current player </returns<

    private List<PieceMove> PossibleMovesSpecificPlayer(UInt32 piece, UInt32 playerBoard, UInt32 enemyBoard, PlayerColor player, int nextRowMove, int nextRowDiagonalMoveEven,

        int nextRowDiagonalMoveOdd, UInt32Funcs.shift_operation op, UInt32Funcs.shift_operation kingBackOp(

}

 //         holds all possible moves the given piece can make, also contains 0s represnting unavailable certain moves

        List<PieceMove> possibleMoves = new List<PieceMove;()<


 //         the give piece's index (right to left 0 to 31) and row (buttom to top 0 to 7)

        int pieceIndex, pieceRow;


 //         index from right to left , right most is 0 , left most is 31

        pieceIndex = System.Numerics.BitOperations.Log2(piece); // holds what number the 1 bit is in the piece


 //         get what row of the board the piece is on (stats from buttom row as row 0 up to the top as row 7)

        pieceRow = pieceIndex / 4;


 //         check if possible move for next row is available

        possibleMoves.Add(CheckPossibleMove(piece, 1, op, playerBoard, enemyBoard, player, nextRowMove, false));



 //         if the piece is a king it can also walk backwards

        if ((piece & GameBoard.K) != 0)
```

```
}
//          check if possible move for back row is available
        possibleMoves.Add(CheckPossibleMove(piece, 1, kingBackOp, playerBoard, enemyBoard, player,
(GameBoard.RowLength * 2) - nextRowMove, true));
{



//        check diagonale possible move for piece, changes depending if its on an even or an odd row
      switch (pieceRow % 2)
}
//          even row
        case:(0)
//              check if possible move for diagonale next row (if the piece is on an even row) is available
            possibleMoves.Add(CheckPossibleMove(piece, pieceIndex + 1, op, playerBoard, enemyBoard,
player, nextRowDiagonalMoveEven, false));


//              if the piece is a king it can also walk backwards
            if ((piece & GameBoard.K) != 0)
}
//                check if possible move for diagonale back row (if the piece is on an even row) is available
              possibleMoves.Add(CheckPossibleMove(piece, pieceIndex + 1, kingBackOp, playerBoard,
enemyBoard, player, (GameBoard.RowLength * 2) - nextRowDiagonalMoveEven, true));
{


            break;


//          odd row
        case:(1)
//              check if possible move for diagonale next row (if the piece is on an odd row) is available
            possibleMoves.Add(CheckPossibleMove(piece, pieceIndex, op, playerBoard, enemyBoard, player,
nextRowDiagonalMoveOdd, false));


//              if the piece is a king it can also walk backwards
            if ((piece & GameBoard.K) != 0)
```

```
}
 //                    check if possible move for diagonale back row (if the piece is on an odd row) is available
                    possibleMoves.Add(CheckPossibleMove(piece, pieceIndex, kingBackOp, playerBoard,
enemyBoard, player, (GameBoard.RowLength * 2) - nextRowDiagonalMoveOdd, true));
{

                break;
{

 //          remove all 0s reprenseing no unavailable/impossible moves for piece (other values are possible
moves)
          possibleMoves.RemoveAll(move => move.Where == 0);

          return possibleMoves;

{

> ///       summary<
 ///       checks if a certain move is possible for given piece
/> ///       summary<
> ///       param name="piece"> UInt32 represnting the piece to check move for </param<
> ///       param name="pieceIndexCheck"> used for checking the location of the piece (if it divides by 4 with
no remainder the move is not possible for the given piece) </param<
> ///       param name="op"> bit shift operation to be made on the piece for the move </param<
> ///       param name="playerBoard"> the player that the piece belongs to board </param<
> ///       param name="enemyBoard"> the enemy player's board </param<
> ///       param name="player"> the player's color PlayerColor.WHITE/PlayerColor.BLACK </param<
> ///       param name="move"> the number of bits to move the piece to in the op direction if given move is
possible </param<
> ///       param name="backWardsMove"> Specifies if the move being checked if a move forwards or a king's
move backwards(Important for sending parameters to CheckPossibleEat function) </param<
> ///       returns> returns a mask representing the possible move for given piece or 0 if no possible move for
the given piece in the given direction is available </returns<
      private PieceMove CheckPossibleMove(UInt32 piece, int pieceIndexCheck, UInt32Funcs.shift_operation
op, UInt32 playerBoard, UInt32 enemyBoard, PlayerColor player, int move, bool backWardsMove)
```

```
}
        UInt32 possibleMove = op(piece, move);
        if (((pieceIndexCheck) % 4 != 0) && ((possibleMove & playerBoard) == 0)) // Possible and location free of
player pieces
}
 //             check if there is an enemy piece already in the location
            if ((possibleMove & enemyBoard) == 0) // location free of enemy pieces
}
                return new PieceMove(possibleMove, false, 0); // return possible available regular move
{
            else // check if enemy piece is edible
}
 //                If regular forwards eat is made
                if (!backWardsMove)
}
 //                    return possbile eating move
                    return player switch
}
                    (PlayerColor.WHITE) => CheckPossibleEat(playerBoard, enemyBoard, possibleMove, move - 1,
move + 1, op),
                    (PlayerColor.BLACK) => CheckPossibleEat(playerBoard, enemyBoard, possibleMove, move + 1,
move - 1, op),
 <= _                    throw new Exception("Invalid player"),
;{
{
 //                If its a backwards eat


 //                return possbile eating move
                return player switch
}
                (PlayerColor.WHITE) => CheckPossibleEat(playerBoard, enemyBoard, possibleMove, move + 1,
move - 1, op),
                (PlayerColor.BLACK) => CheckPossibleEat(playerBoard, enemyBoard, possibleMove, move - 1,
move + 1, op),
 <= _                  throw new Exception("Invalid player"),
```

```
;{



{
{

 //         move not possible/available, return 0 representing no available location on the board for the move
        return new PieceMove(0, false, 0);
{



> ///      summary<
 ///      checks if the given enemy piece (possibleMove) is edible
/> ///       summary<
> ///      param name="playerBoard"> the current player's board </param<
> ///      param name="enemyBoard"> the enemy player's board </param<
> ///      param name="possiblyEdible"> the enemy piece that might be edible </param<
> ///      param name="moveEven"> how many bits to move past the possiblyEdible enemy piece to check if its truly edible if the possiblyEdible piece is on an even row </param<
> ///      param name="moveOdd"> how many bits to move past the possiblyEdible enemy piece to check if its truly edible if the possiblyEdible piece is on an odd row </param<
> ///      param name="op"> bit shift operation to be made on the possiblyEdible piece to check if its truly edible and a mask representing where the eating piece will mvoe to if its edible </param<
> ///      returns> if the piece is edible returns where a mask representing where the eating piece will move to and if its not returns 0 representing piece cannot be eaten </returns<
      private PieceMove CheckPossibleEat(UInt32 playerBoard, UInt32 enemyBoard, UInt32 possiblyEdible, int moveEven, int moveOdd, UInt32Funcs.shift_operation op)
}
 //         holds the index of the possibleMove (what index the 1 bit is in it)
        int possibleMoveIndex = System.Numerics.BitOperations.Log2(possiblyEdible);


 //         holds the row of the possibleMove (0 to 7)
        int possibleMoveRow = possibleMoveIndex / 4 ;


        switch (possibleMoveRow % 2)
```

```
}
 //            even row
            case:(0)
 //                check if there is an empty square in the next row so this piece can be eaten
                if (((possibleMoveIndex + 1) % 4 != 0) && ((op(possiblyEdible, moveEven) & enemyBoard) == 0) &&
((op(possiblyEdible, moveEven) & playerBoard) == 0)) // enemy piece edible
}
                    return new PieceMove((op(possiblyEdible, moveEven)), true, possiblyEdible); // return where the
piece has to move after eating
{
                break;

 //            odd row
            case:(1)
 //                check if there is an empty square in the next row so this piece can be eaten
                if ((possibleMoveIndex % 4 != 0) && ((op(possiblyEdible, moveOdd) & enemyBoard) == 0) &&
((op(possiblyEdible, moveOdd) & playerBoard) == 0)) // enemy piece edible
}
                    return new PieceMove((op(possiblyEdible, moveOdd)), true, possiblyEdible); // return where the
piece has to move after eating
{
                break;
{

 //          move not possible/available, return 0 representing no available location on the board for the move
          return new PieceMove(0, false, 0);
{

{
{
using System;

namespace Checkers
{
   /// <summary>
   /// struct describing a possible move for a certain piece
   /// </summary>
```

```
    public struct PieceMove
    {
        /// <summary>
        /// where the move leads to (as a mask)
        /// </summary>
        public UInt32 Where { get; }


        /// <summary>
        /// whether the move is an eating move or a regular move (eating move means an enemy piece is being
eaten during the move)
        /// </summary>
        public bool Eat { get; }

        /// <summary>
        /// if the move is an eating move this holds where the eaten piece is (as a mask), if no piece was eaten 0 is
to be inserted
        /// </summary>
        public UInt32 WhereEaten { get; }


        /// <summary>
        /// Constructor of PieceMove struct
        /// </summary>
        /// <param name="where"> where the move leads to (as a mask) </param>
        /// <param name="eat"> whether the move is an eating move or a regular move (eating move means an
enemy piece is being eaten during the move) </param>
        /// <param name="whereEaten"> if the move is an eating move this holds where the eaten piece is (as a
mask), if no piece was eaten 0 is to be inserted </param>
        public PieceMove(UInt32 where, bool eat, UInt32 whereEaten)
        {
            this.Where = where;
            this.Eat = eat;
            this.WhereEaten = whereEaten;
        }
    }
}
using System;

namespace Checkers
{
    /// <summary>
    /// enum representing the player color
    /// </summary>
    public enum PlayerColor
    {
        WHITE,
        BLACK
    }

    /// <summary>
    /// contains the player's piece locations, color and first row
    /// </summary>
    public class Player
    {
        /// <summary>
        /// The player's color
```

```
    /// </summary>
    public PlayerColor Color;

    /// <summary>
    /// The player's board , a 32 bit integer where every 1 bit represents this player's piece and every 0 bit
represents empty board/ other player piece
    /// </summary>
    public UInt32 PB;

    /// <summary>
    /// the value representing where the first row of the player starts
    /// </summary>
    public UInt32 FirstRowStart;


    /// <summary>
    /// the value representing where the first row of the player ends
    /// </summary>
    public UInt32 FirstRowEnd;


    /// <summary>
    /// Initiate PB, FirstRow and Color properties
    /// </summary>
    /// <param name="PB"></param>
    /// <param name="Color"></param>
    /// <param name="FirstRowStart"></param>
    /// <param name="FirstRowEnd"></param>
    public Player(UInt32 PB, PlayerColor Color, UInt32 FirstRowStart, UInt32 FirstRowEnd)
    {
      this.PB = PB;
      this.Color = Color;
      this.FirstRowStart = FirstRowStart;
      this.FirstRowEnd = FirstRowEnd;
    }

    /// <summary>
    /// gives the user a piece at a certain index (index from right to left)
    /// if there is no piece at that index returns 0
    /// </summary>
    /// <param name="selectedPieceIndex"> the index of the selected piece from right to left (0 - 31)
</param>
    /// <returns> an UInt32 that only the bit at its i location (from right to left) is 1 and the rest are 0 IF a piece
exists for player in that index , otherwise returns 0 </returns>
    public UInt32 GetPiece(int selectedPieceIndex)
    {
      // UInt32 representing the given piece (all bits are 0 except bit where the piece is which is 1)
      UInt32 piece = 1;
      for (int bitIndex = 0; bitIndex < selectedPieceIndex; bitIndex++)
      {
        piece *= 2;
      }

      // check if there is a piece at the given location , if not it resets piece to 0
      piece &= PB;

      return piece;
```

```
        }
    }
}

namespace Checkers
{
    /// <summary>
    /// Enum representing the types of Heuristics that can evaluate the game for the bot
    /// </summary>
    public enum BotHeuristics
    {
        BASIC,
        EXTREME,
        ADAPTIVE
    }
}


namespace Checkers
{

    /// <summary>
    /// Enum representing all diffrent diffculty levels a bot can have
    /// </summary>
    public enum BotLevel
    {
        EASY,
        MEDIUM,
        HARD,
        EXTREME,
        ADAPTIVE
    }
}


namespace Checkers
{
    /// <summary>
    /// Coordinates consisting of row to describe row number on the board and col to describe collumn number
on the board
    /// </summary>
    public struct Coordinates
    {
        /// <summary>
        /// Row number on the board (0 - 7)
        /// </summary>
        public int row;

        /// <summary>
        /// Collumn number on the board (0 - 7)
        /// </summary>
        public int col;

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="row"> row number on the board </param>
```

```
        /// <param name="col"> collumn number on the board </param>
        public Coordinates(int row, int col)
        {
            this.row = row;
            this.col = col;
        }
    }
}
using System;

using System.Collections.Generic;

using System.Drawing;

using System.Threading;

using System.Windows.Forms;


namespace Checkers

}

> ///    summary<

 ///    UserInterface takes care of all interaction of the game with the user and all visuals In addition to
connecting the models (the game logic) In the correct way

/> ///    summary<

    public partial class UserInterface : Form

}

> ///       summary<

 ///       Constructor initializes component

/> ///       summary<

        public UserInterface()

}

            InitializeComponent;()


{


> ///       summary<

 ///       Screen resulotion used to place widgest in the corrent place

/> ///       summary<

        private static Rectangle resolution = Screen.PrimaryScreen.Bounds;


 //       Widgets
```

```
> ///     summary<
 ///     Button designed to offer option to stop chain eating
/> ///     summary<
    private Button ChainStopButton = new Button;()

 //     logical parameters for running a game course

> ///     summary<
 ///     Model holding the game's logic
/> ///     summary<
    private Model M = new Model;()

> ///     summary<
 ///     The Color of the current Player (that player that is now making his turn)
/> ///     summary<
    private PlayerColor currentPlayer;

> ///     summary<
 ///     The game mode chosen (PvP/PvB/BvB)
/> ///     summary<
    private string Mode;

> ///     summary<
 ///     List of Possible Moves rows and collumns for drawing the board with highlighted possbile moves for
latest clicked piece
/> ///     summary<
    private List<Coordinates> PossibleMovesCoordinates = new List<Coordinates;()<

> ///     summary<
 ///     List of logical Possible moves for making a move if a possible move square has been clicked
/> ///     summary<
    private List<PieceMove> MovesList = new List<PieceMove;()<
```

```
> ///      summary<
 ///      Latest clicked valid piece (piece to move if possible move square has been clicked)
/> ///      summary<
      private UInt32 PieceToMove = 0;

> ///      summary<
 ///      True if player is in the middle of making a move (already chosen a piece and is now chossing a possible
move) false otherwise
/> ///      summary<
      private bool midMove = false;

> ///      summary<
 ///      turns true if a player is in the proccess of chain eating
/> ///      summary<
      private bool inChain = false;

> ///      summary<
 ///      used for chain eating holds the location of the eating piece
/> ///      summary<
      private UInt32 eatingPiece = 0;

 //      Graphical constants for placement/size

> ///      summary<
 ///      size of each square on the board (changes depending on resolution)
 100% ///      size
/> ///      summary<
      private static readonly int size = (int)(resolution.Width * 0.033) + (int)(resolution.Height * 0.03);

> ///      summary<
 ///      how many squares are in each row and collumn
/> ///      summary<
```

```
        private static readonly int numOfSquares = 8;

> ///       summary<
 ///       distance of board from edge of screen (changes depending on resolution)
 80% ///       of size
/> ///       summary<
        private static readonly int distanceFromEdge = (int)(size*0.8);

> ///       summary<
 ///       The diameter of each piece (changes depending on resolution)
 40% ///       of size
/> ///       summary<
        private static readonly int pieceDiameter = (int)(size*0.4);

 //       functions

> ///       summary<
 ///       Draw the game board and its pieces as it is to the screen
/> ///       summary<
> ///       param name="b"> The game board </param<
        public void DrawBoard(Board b)
}
 //          color of painted pieces/squares

 //          all back colors
            System.Drawing.SolidBrush myBrushBlackPiece = new
System.Drawing.SolidBrush(System.Drawing.Color.Black);
            System.Drawing.SolidBrush myBrushBlackKing = new
System.Drawing.SolidBrush(System.Drawing.Color.DarkRed);
            Pen myPenBlackPiece = new Pen(Brushes.White);
            Pen myPenBlackKing = new Pen(Brushes.Gold);

 //          all white colors
```

```
        System.Drawing.SolidBrush myBrushWhitePiece = new
System.Drawing.SolidBrush(System.Drawing.Color.White);

        System.Drawing.SolidBrush myBrushWhiteKing = new
System.Drawing.SolidBrush(System.Drawing.Color.LightGray);

        Pen myPenWhitePiece = new Pen(Brushes.Black);

        Pen myPenWhiteKing = new Pen(Brushes.Silver);


//          Paints empty squares

        System.Drawing.SolidBrush myBrushEmpty = new
System.Drawing.SolidBrush(System.Drawing.Color.Black);


//          Paints possible moves squares

        System.Drawing.SolidBrush myBrushPossibleMove = new
System.Drawing.SolidBrush(System.Drawing.Color.Green);


//          Set formGraphics

        System.Drawing.Graphics formGraphics = this.CreateGraphics;()


//          String form of all game boards used to place pieces on the graphical board

        string black_board = UInt32Funcs.Make32Bit(b.BP.PB), white_board =
UInt32Funcs.Make32Bit(b.WP.PB), king_board = UInt32Funcs.Make32Bit(b.K);


//          index on the translated bit board

        int indexBitBoard = 0;


//          draw board

        for (int row = 0; row < numOfSquares; row++)

}

           for (int col = 0; col < numOfSquares; col++)

}

//                if its a black square draw it, otherwise color is already white

              if (((col % 2 != 0) && (row % 2 == 0)) || ((col % 2 == 0) && (row % 2 != 0)))

}


//                  draw black square
```

```
                formGraphics.FillRectangle(myBrushEmpty, new Rectangle(col * size + distanceFromEdge, row *
size + distanceFromEdge, size, size));


 //                 black piece

                if (black_board[indexBitBoard].Equals('1'))

}

                  if (king_board[indexBitBoard].Equals('1'))

}

 //                     Draw Black King

                    formGraphics.DrawCircle(myPenBlackKing, col * size + distanceFromEdge + size / 2, row *
size + distanceFromEdge + size / 2, pieceDiameter + 1);

                    formGraphics.FillCircle(myBrushBlackKing, col * size + distanceFromEdge + size / 2, row * size
+ distanceFromEdge + size / 2, pieceDiameter);

{

                  else

}

 //                     Draw Black Solider

                    formGraphics.DrawCircle(myPenBlackPiece, col * size + distanceFromEdge + size/2, row *
size + distanceFromEdge + size/2, pieceDiameter + 1);

                    formGraphics.FillCircle(myBrushBlackPiece, col * size + distanceFromEdge + size/2, row * size
+ distanceFromEdge + size/2, pieceDiameter);

{

{


 //                 white piece

                else if (white_board[indexBitBoard].Equals('1'))

}

                  if (king_board[indexBitBoard].Equals('1'))

}

 //                     Draw White King

                    formGraphics.DrawCircle(myPenWhiteKing, col * size + distanceFromEdge + size / 2, row *
size + distanceFromEdge + size / 2, pieceDiameter+ 1);

                    formGraphics.FillCircle(myBrushWhiteKing, col * size + distanceFromEdge + size / 2, row *
size + distanceFromEdge + size / 2, pieceDiameter);

{

                  else
```

```
}
//                    Draw White Solider
                    formGraphics.DrawCircle(myPenWhitePiece, col * size + distanceFromEdge + size / 2, row *
size + distanceFromEdge + size / 2, pieceDiameter+ 1);
                    formGraphics.FillCircle(myBrushWhitePiece, col * size + distanceFromEdge + size / 2, row *
size + distanceFromEdge + size / 2, pieceDiameter);
{
{

//                Increase bit board index
              indexBitBoard;++
{
{

{

//        Dispose all used pens and brushes
        myBrushEmpty.Dispose;()
        myBrushBlackPiece.Dispose;()
        myBrushWhitePiece.Dispose;()
        myPenBlackKing.Dispose;()
        myPenWhiteKing.Dispose;()
        myPenWhitePiece.Dispose;()
        myBrushBlackPiece.Dispose;()

//        Draw border around board (black border)
        Pen pen = new Pen(Color.FromArgb(255, 0, 0, 0));
        formGraphics.DrawLine(pen, distanceFromEdge, distanceFromEdge, size * numOfSquares +
distanceFromEdge, distanceFromEdge);
        formGraphics.DrawLine(pen, distanceFromEdge, distanceFromEdge, distanceFromEdge, size *
numOfSquares + distanceFromEdge);
        formGraphics.DrawLine(pen, distanceFromEdge, size * numOfSquares + distanceFromEdge, size *
numOfSquares + distanceFromEdge, size * numOfSquares + distanceFromEdge);
        formGraphics.DrawLine(pen, size * numOfSquares + distanceFromEdge, distanceFromEdge, size *
numOfSquares + distanceFromEdge, size * numOfSquares + distanceFromEdge);
```

```
//          Dispose used pen
        pen.Dispose;()



//          Draw Possible moves
        foreach(Coordinates move in PossibleMovesCoordinates)
}
//              draw possible move square
            formGraphics.FillRectangle(myBrushPossibleMove, new Rectangle(move.col * size +
distanceFromEdge, move.row * size + distanceFromEdge, size, size));
{


//          Dispose of brush and formgraphics (finished using them)
        formGraphics.Dispose;()
        myBrushPossibleMove.Dispose;()
{


> ///       summary<
///       When mouse is clicked takes care of what happens
/> ///       summary<
> ///       param name="e"> mouse click event </param<
      protected override void OnClick(EventArgs e)
}
//          call base OnClick
        base.OnClick(e);


//          Save the current Coordinates of the mouse (x and y)
        int CurrentX = MousePosition.X;
        int CurrentY = MousePosition.Y;


//          Translate the current Coordinates to collumn and row in the board
        int col = (CurrentX - distanceFromEdge) / size;
```

```
int row = (CurrentY - distanceFromEdge) / size;

//      Check which game mode is on and act accordingly
switch (Mode)
}
//        Player Versus Player mode
case ("PvP"):

PlayerTurn(row, col);

//          Check if game is over
if(M.CheckGameState() != GameState.OnGoing)
}
//            Show game outcome
MessageBox.Show(M.CheckGameState().ToString());

//            Game over reset mode
Mode;"" =
{

break;

//        Player Versus Bot Mode
case ("PvB"):

//          If its the player's turn
if (currentPlayer == PlayerPvB && row <=7 && col <=7)
}
PlayerTurn(row, col);
{

//          Check if game is over
if (M.CheckGameState() != GameState.OnGoing)
```

```
}
//                Show game outcome
                MessageBox.Show(M.CheckGameState().ToString());

//                Game over reset mode
                Mode;"" =
{

              break;

//            Bot Versus Bot Mode
            case ("BvB"):
              MessageBox.Show("Bot Versus Bot currently running");
               break;

{

{

> ///      summary<
///      Takes care of a player's move
/> ///      summary<
> ///      param name="row"> row of clicked piece </param<
> ///      param name="col"> collumn of clicked piece </param<
      private void PlayerTurn(int row, int col)
}
//          If Square pressed is a black square that may contain a moveable piece
        if ((((col % 2 != 0) && (row % 2 == 0)) || ((col % 2 == 0) && (row % 2 != 0))) && (row < 8 && col < 8))
}
//            Get piece's index using the row and collumn its on index is from right to left buttom to top 0 to 31
only on the black squares
            int index = ((numOfSquares - 1 - row) * numOfSquares / 2) + (numOfSquares / 2 - (col / 2 + 1));
```

```
//          If in the middle of chain eating and need to select piece , only possible selectable piece is the one
doing the chain eating
        if (inChain && !midMove)
}
//              ask if he wants to quit and switch turn with pop up button or something
//              GADKNGKADHGKADNGJNADJKGADJKGHAJDHGJADHGKHADJKGHJKADHGJKAHGHJK
            index = System.Numerics.BitOperations.Log2(eatingPiece);
{

        UInt32 tempPieceToMove = M.GetPlayerBoard(currentPlayer).GetPiece(index);
//          Get Moves List for given square (if the square has no piece on it gets an empty list)
        List<PieceMove> tempMovesList = M.GetPossibleMoves(currentPlayer, tempPieceToMove);

//          Delete all none eating moves when chain eating
        if (inChain)
}
            tempMovesList.RemoveAll((PieceMove move) => move.Eat == false);
{

//          Temp list holding all moves row and col
        List<Coordinates> tempCoordinatesMovesList = new List<Coordinates;()<

//          if click was made midmove check if what was clicked was a green square (meaning in the MovesList
already)
        if (midMove)
}
//              True if a move is made
            bool isMove = false;

//              Check MovesList (from previous click)
            foreach (PieceMove piece in MovesList)
}
//                Compare to current Clicked Piece (if current clicked piece was a possiblemove of previous click)
                if (System.Numerics.BitOperations.Log2(piece.Where) == index)
```

```
}
                    isMove = true;

//                  Make Move with PieceToMove and the Piece From the MovesList
                    if (M.MakeMove(currentPlayer, PieceToMove, piece))
}
//                  If player just entered chain eating button (if statement used to prevent multiple buttons
being placed)
                    if (!inChain)
}
//                  Create button allowing to quit chain eating
                    ChainStopButton = new Button;()
                    ChainStopButton.Location = new Point(size * numOfSquares + distanceFromEdge + 5, (size
* (numOfSquares / 2) - size / 2) + distanceFromEdge);
                    ChainStopButton.Height = size;
                    ChainStopButton.Width = size;

//                  Set background and foreground
                    ChainStopButton.BackColor = Color.DarkRed;
                    ChainStopButton.ForeColor = Color.White;

                    ChainStopButton.Text = "End turn;"
                    ChainStopButton.Name = "End Chain;"
                    ChainStopButton.Font = new Font("Georgia", 16);

//                  Add a Button Click Event handler
                    ChainStopButton.Click += new EventHandler(ChainStopButton_Click);

                    Controls.Add(ChainStopButton);
{

//                  Player in chain eating turn
                    inChain = true;
                    eatingPiece = piece.Where;
```

```
{
                else
}
                  inChain = false;
{

{
{

            if (isMove)
}
              EndTurn;()
              return;
{
{

 //          Find every move's piece row and collumn
          foreach (PieceMove piece in tempMovesList)
}
            int tempIndex = System.Numerics.BitOperations.Log2(piece.Where);
            Coordinates temp;

 //             move is on an even row
            if ((tempIndex / 4 - 1) % 2 == 0)
}
              temp = new Coordinates(numOfSquares - 1 - (tempIndex / 4), (numOfSquares - 1) - (tempIndex %
4 * 2));
{

 //             move is on an odd row
            else
}
```

```
temp = new Coordinates(numOfSquares - 1 - (tempIndex / 4), (numOfSquares - 1) - (tempIndex %
4 * 2) - 1);
{

tempCoordinatesMovesList.Add(temp);
{

//          Put the tempMovesList in the PossibleMovesCoordinates for printing the board again with the
possible moves marked
PossibleMovesCoordinates = tempCoordinatesMovesList;

MovesList = tempMovesList;

PieceToMove = tempPieceToMove;

//          If the given piece has possible moves it can make redraw the game board
if (PossibleMovesCoordinates.Count > 0)
}
//          ReDraw the game board with the possiblemoves
midMove = true;
DrawBoard(M.GameBoard);
{

{
{

> ///     summary<
///     End Turn of a player, reset all variables and switch player
/> ///     summary<
private void EndTurn()
}
//          Move made reset all Move lists
PossibleMovesCoordinates = new List<Coordinates;()<
```

```
        MovesList = new List<PieceMove;()<

        PieceToMove = 0;

        midMove = false;

        DrawBoard(M.GameBoard);

//          If in middle of chain eating do not switch turns
        if (!inChain)
}
//              Switching player remove stop chain eating button
            Controls.Remove(ChainStopButton);

            currentPlayer = currentPlayer switch
}
                (PlayerColor.WHITE) => PlayerColor.BLACK,
                (PlayerColor.BLACK) => PlayerColor.WHITE,
<= _                throw new Exception("Invalid player"),
;{
{
{

> ///       summary<
 ///       Stops chain eating, switches turn and destroys itself
/> ///       summary<
> ///       param name="sender"></param<
> ///       param name="e"></param<
      private void ChainStopButton_Click(object sender, EventArgs e)
}
//          End the player's turn and switch to other player
        inChain = false;
        eatingPiece = 0;
```

```
        EndTurn;()

 //         Destroy the button
        Controls.Remove((Control)sender);
{

----------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------
 //       Player Vs Player
----------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------

> ///       summary<
 ///       Variable used to terminate threads , if its true any running thread will shut down
/> ///       summary<
      private bool killThreads = false;

> ///       summary<
 ///       Handle PvP (Player Versus Player) game course
/> ///       summary<
> ///       param name="sender"></param<
> ///       param name="e"></param<
      private void PvP_Click(object sender, EventArgs e)
}
 //         Terimate all running threads
        if ((BotTurnThread != null && BotTurnThread.IsAlive) || (BvBGameCourseThread != null &&
BvBGameCourseThread.IsAlive))
}
            killThreads = true;
 //             Wait until thread resets the command
            while(killThreads != false)
}
                Thread.Sleep;(250)
{
```

```
{

//      Reset all variables used in PvP match
        PossibleMovesCoordinates = new List<Coordinates;()<

        MovesList = new List<PieceMove;()<

        PieceToMove = 0;

        midMove = false;

        inChain = false;

        eatingPiece = 0;

//      Set game mode
        Mode = "PvP;"

//      Create game model
        M = new Model;()

//      Draw board
        DrawBoard(M.GameBoard);

//      Set starting player (always white)
        currentPlayer = PlayerColor.WHITE;

{

> ///     summary<
 ///     Display game instructions to the user for PvP mode
/> ///     summary<
> ///     param name="sender"></param<
```

```
> ///      param name="e"></param<
      private void InstructionsPvP_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
}
          MessageBox.Show("PvP (Player versus Player) is a mode where 2 players from the same computer play checkers against each other. ");
{

-------------------------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------------------
 //      Player Versus Bot
-------------------------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------------------

> ///      summary<
 ///      Any Bot Depth Must be below this depth, beyond this depth it takes too long to make a move
/> ///      summary<
      private int MaximumBotDepth = 10;

> ///      summary<
 ///      The player's Color in PvB (Default black)
/> ///      summary<
      private PlayerColor PlayerPvB = PlayerColor.BLACK;

> ///      summary<
 ///      The bot's color in PvB (Default white)
/> ///      summary<
      private PlayerColor BotPvB = PlayerColor.WHITE;

> ///      summary<
 ///      True if the player chose to be white and false if black
/> ///      summary<
      private bool playerWhite = false;

> ///      summary<
```

```
 ///      Bot in PvB
/> ///      summary<
      private Ai Bot;

> ///      summary<
 ///      Thread responsible for handling the bot's turn in PvB
/> ///      summary<
      private Thread BotTurnThread = null;

> ///      summary<
 ///      Enum holding the bot difficulty level, Medium by defauly
/> ///      summary<
      private BotLevel DifficultyLevel = BotLevel.HARD;

> ///      summary<
 ///      Chosen difficulty level is set as easy
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void EasyDifficulty_CheckedChanged(object sender, EventArgs e)
}
         DifficultyLevel = BotLevel.EASY;
         BotDepth.Text = "3;"
         BotKingWeight.Text = "4 ;"
         BotSoliderWeight.Text = "2;"
         BotPMWeight.Text= "1;"
         BotWinScore.Text = "100;"
         BotLoseScore.Text = "-100;"
         BotDrawScore.Text = "-90;"

{

> ///      summary<
```

```
 ///       Chosen difficulty level is set as medium
/> ///       summary<
> ///       param name="sender"></param<
> ///       param name="e"></param<
      private void MediumDifficulty_CheckedChanged(object sender, EventArgs e)
}
          DifficultyLevel = BotLevel.MEDIUM;
          BotDepth.Text = "5;"
          BotKingWeight.Text = "4;"
          BotSoliderWeight.Text = "2;"
          BotPMWeight.Text = "1;"
          BotWinScore.Text = "100;"
          BotLoseScore.Text = "-100;"
          BotDrawScore.Text = "-90;"
{

> ///       summary<
 ///       Chosen difficulty level is set as hard
/> ///       summary<
> ///       param name="sender"></param<
> ///       param name="e"></param<
      private void HardDifficulty_CheckedChanged(object sender, EventArgs e)
}
          DifficultyLevel = BotLevel.HARD;
          BotDepth.Text = "7;"
          BotKingWeight.Text = "4;"
          BotSoliderWeight.Text = "2;"
          BotPMWeight.Text = "1;"
          BotWinScore.Text = "100;"
          BotLoseScore.Text = "-100;"
          BotDrawScore.Text = "-90;"
{
```

```
> ///      summary<
 ///      Chosen difficulty level is set as Extreme
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void ExtremeDifficulty_CheckedChanged(object sender, EventArgs e)
}
         DifficultyLevel = BotLevel.EXTREME;
         BotDepth.Text = "7;"
         BotKingWeight.Text;"-" =
         BotSoliderWeight.Text;"-" =
         BotPMWeight.Text;"-" =
         BotWinScore.Text;"-" =
         BotLoseScore.Text;"-" =
         BotDrawScore.Text;"-" =
{

> ///      summary<
 ///      Chosen difficulty level is set as Adaptive
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void PvBAdaptive_CheckedChanged(object sender, EventArgs e)
}
         DifficultyLevel = BotLevel.ADAPTIVE;
         BotDepth.Text = "7;"
         BotKingWeight.Text;"-" =
         BotSoliderWeight.Text;"-" =
         BotPMWeight.Text;"-" =
         BotWinScore.Text;"-" =
         BotLoseScore.Text;"-" =
         BotDrawScore.Text;"-" =
{
```

```
> ///      summary<
 ///      User chose to be white player (meaning bot is black player)
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void PlayerIsWhite_CheckedChanged(object sender, EventArgs e)
}
         playerWhite = true;
{

> ///      summary<
 ///      User chose to be black player (meaning bot is white player)
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void PlayerIsBlack_CheckedChanged(object sender, EventArgs e)
}
         playerWhite = false;
{

> ///      summary<
 ///      Function handling the bot's turn in PvB
/> ///      summary<
      private void BotTurn()
}
 //          if command to kill threads is sent or if game is over terminate thread
         while (M.CheckGameState() == GameState.OnGoing && !killThreads)
}
            Thread.Sleep;(100)
            if (currentPlayer == BotPvB)
}
```

```
//              If game mode changedl or if game is already over or if command to kill threads is sent terimate
thread
            if (Mode != "PvB" || M.CheckGameState() != GameState.OnGoing || killThreads)
}
              break;
{

            Bot.MakeMove(M);
            DrawBoard(M.GameBoard);

//              Game ended terminate thread and show winner
            if (M.CheckGameState() !=GameState.OnGoing)
}
              MessageBox.Show(M.CheckGameState().ToString());
              break;
{

            currentPlayer = PlayerPvB;
{
{

//        This thread has stopped (been killed)
        killThreads = false;
{

> ///      summary<
///      Handle setting up PvB (Player versus Bot)
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void PvB_Click(object sender, EventArgs e)
}
//        Terimate all running threads
```

```
        if ((BotTurnThread != null && BotTurnThread.IsAlive) || (BvBGameCourseThread != null &&
BvBGameCourseThread.IsAlive))
}
            killThreads = true;
//              Wait until thread resets the command
            while (killThreads != false)
}
                Thread.Sleep;(250)
{
{

//          Player chose to be white
        if(playerWhite)
}
            PlayerPvB = PlayerColor.WHITE;

            BotPvB = PlayerColor.BLACK;
{
//          Player chose to be black
        else
}

            PlayerPvB = PlayerColor.BLACK;

            BotPvB = PlayerColor.WHITE;
{

//          Heuristic function chosen for the bot
        Heuristics.HeuristicFunc BotChosenHeuristic;

//          If a heuristic that only uses depth is chosen
        bool onlyDepthMatters = false;
```

```
//          Set up Bot according to chosen difficulty level (difficulty level effects depth of minimax algorithem)
        switch (DifficultyLevel)
}
            case (BotLevel.EXTREME):
//                  Extreme Uses a different heuristic which takes into account a lot of extra parameters and has its
weights already set
                BotChosenHeuristic = Heuristics.ExtremeHeuristic;
                onlyDepthMatters = true;
                break;


            case (BotLevel.ADAPTIVE):
//                  Adaptive Uses a different heuristic which scores things relative to themselves (and changes
scoring for parameters throughout the game)
                BotChosenHeuristic = Heuristics.AdaptiveHeuristic;
                onlyDepthMatters = true;
                break;


            default:
//                  Basic Heuristic (Used for easy/medium/hard difficulties)
                BotChosenHeuristic = Heuristics.BasicHeuristic;
                break;
{


//          Message in exception thrown if given depth was below 1
        string depthErrorMessage = "Depth must be 1 or higher And must be below " +
MaximumBotDepth.ToString() + " or it will take too long to calculate move;"
//          Set Ai with wanted parameters
        try
}
//              If given depth is below 1 Or above maximum bot depth
            if (int.Parse(BotDepth.Text) <= 0 || int.Parse(BotDepth.Text) >= MaximumBotDepth)
                throw new Exception(depthErrorMessage);


            if (onlyDepthMatters)
```

```
}
//              Bot parameters set as user indicated
            Bot = new Ai(BotPvB, BotChosenHeuristic, int.Parse(BotDepth.Text), 0, 0, 0, 0, 0, 0);
{
        else
}
//              Bot parameters set as user indicated
            Bot = new Ai(BotPvB, BotChosenHeuristic, int.Parse(BotDepth.Text,(
            int.Parse(BotKingWeight.Text), int.Parse(BotSoliderWeight.Text), int.Parse(BotPMWeight.Text),
int.Parse(BotWinScore.Text), int.Parse(BotLoseScore.Text), int.Parse(BotDrawScore.Text);(
{
{
      catch (Exception except)
}
//          Inform user of the error cause
        if (except.Message == depthErrorMessage)
}
//              Error was due to depth given being below 1
            MessageBox.Show(depthErrorMessage + " Default ai values set");
{

        else
//              Error was due to invalid type of value entered
            MessageBox.Show("1 or more values entered were invalid, Default ai values set);("

//          Set default values
        BotDepth.Text = "7;"
        BotKingWeight.Text = "4;"
        BotSoliderWeight.Text = "2;"
        BotPMWeight.Text = "1;"
        BotWinScore.Text = "100;"
        BotLoseScore.Text = "-100;"
        BotDrawScore.Text = "-90;"
```

```
//          default values if 1 or more of the parameter values were invalid
        Bot = new Ai(BotPvB, BotChosenHeuristic, 7, 4, 2, 1, 100, -100, -90);
{

//      Reset all variables used in PvB match
        PossibleMovesCoordinates = new List<Coordinates;()<

        MovesList = new List<PieceMove;()<

        PieceToMove = 0;

        midMove = false;

        inChain = false;

        eatingPiece = 0;

//      Set game mode
        Mode = "PvB;"

//      Create game model
        M = new Model;()

//      Draw board
        DrawBoard(M.GameBoard);

//      Set starting player (always white)
        currentPlayer = PlayerColor.WHITE;

//      Set Bot Turn handler thread
        BotTurnThread = new Thread(BotTurn);
        BotTurnThread.Start;()
```

```
{


> ///      summary<

 ///      Display game instructions to the user for PvB mode

/> ///      summary<

> ///      param name="sender"></param<

> ///      param name="e"></param<

      private void InstructionsPvB_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)

}

          MessageBox.Show("PvB (Player Versus Bot) is a mode where a player and a bot play checkers against
each other. \n\n+ "

"                    To choose what color the player is tick a color below (default color is black for player and
white for bot). \n\n+ "

"                    To Choose the difficulty of the bot tick a difficulty level below (defauly difficulty is hard)+ ",

 "                    difficulties effect the depth stat of the bot, The difficulties are: \n\n+ "

"                    Easy -> Depth 3. \n+ "

"                    Medium -> Depth 5. \n+ "

"                    Hard -> Depth 7. \n+ "

"                    Extreme -> Depth 7, uses Extreme Heuristic (Hardest Difficulty). \n+ "

"                    Adaptive -> Depth 7, uses Adaptive Heuristic. \n\n+ "

"                    The User can customize all bot stats, Takes stats from  bot stats below, the customizable stats
are as follow: \n+ "

  "                      Depth - How many moves ahead will the Bot go over, Recommended under 9. \n+ "

  "                      King Weight - How many points is a king piece worth for the bot. \n+ "

  "                      Solider Weight - How many points is a Solider piece worth for the bot. \n+ "

  "                      PM Weight - How many points is a possible move worth for the bot. \n+ "

  "                      Win Score - How many points is a win worth for the bot. \n+ "

  "                      Lose Score - How many points is a lose worth for the bot. \n+ "

  "                      Draw Score - How many points is a draw worth for the bot. \n \n+ "

  "                      Max Depth is set as " + (MaximumBotDepth - 1).ToString() + " for preformance reasons;(".

{
```

```
-----------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------
 //      Bot Versus Bot
-----------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------


> ///      summary<
 ///      Thread handling the game course of a bot versus bot in BvB
/> ///      summary<
      private Thread BvBGameCourseThread = null;


> ///      summary<
 ///      White Bot heuristic function to use
/> ///      summary<
      private BotHeuristics WhiteHeuristicBvB = BotHeuristics.BASIC;


> ///      summary<
 ///      Black Bot heuristic function to use
/> ///      summary<
      private BotHeuristics BlackHeuristicBvB = BotHeuristics.BASIC;


> ///      summary<
 ///      White Heuristic chosen as basic (regular heuristic with customer parameters)
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void BasicWhite_CheckedChanged(object sender, EventArgs e)
}
         WhiteHeuristicBvB = BotHeuristics.BASIC;
         WhiteDepth.Text = "7;"
         WhiteKingWeight.Text = "4;"
         WhiteSoliderWeight.Text = "2;"
         WhitePMWeight.Text = "1;"
         WhiteWinScore.Text = "100;"
```

```
        WhiteLoseScore.Text = "-100;"
        WhiteDrawScore.Text = "-90;"
{

> ///      summary<
 ///      White heuristic chosen as extreme (extremeHeuristic without custom parameters except for depth)
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void ExtremeWhite_CheckedChanged(object sender, EventArgs e)
}
        WhiteHeuristicBvB = BotHeuristics.EXTREME;
        WhiteDepth.Text = "7;"
        WhiteKingWeight.Text;"-" =
        WhiteSoliderWeight.Text;"-" =
        WhitePMWeight.Text;"-" =
        WhiteWinScore.Text;"-" =
        WhiteLoseScore.Text;"-" =
        WhiteDrawScore.Text;"-" =
{

> ///      summary<
 ///      White heuristic chosen as adaptive
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void BvBWhiteAdaptive_CheckedChanged(object sender, EventArgs e)
}
        WhiteHeuristicBvB = BotHeuristics.ADAPTIVE;
        WhiteDepth.Text = "7;"
        WhiteKingWeight.Text;"-" =
        WhiteSoliderWeight.Text;"-" =
        WhitePMWeight.Text;"-" =
```

```
        WhiteWinScore.Text;"-" =
        WhiteLoseScore.Text;"-" =
        WhiteDrawScore.Text;"-" =
{

> ///      summary<
 ///      Black Heuristic chosen as basic (regular heuristic with customer parameters)
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void BasicBlack_CheckedChanged(object sender, EventArgs e)
}
        BlackHeuristicBvB = BotHeuristics.BASIC;
        BlackDepth.Text = "7;"
        BlackKingWeight.Text = "4;"
        BlackSoliderWeight.Text = "2;"
        BlackPMWeight.Text = "1;"
        BlackWinScore.Text = "100;"
        BlackLoseScore.Text = "-100;"
        BlackDrawScore.Text = "-90;"
{

> ///      summary<
 ///      Black heuristic chosen as extreme (extremeHeuristic without custom parameters except for depth)
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void ExtremeBlack_CheckedChanged(object sender, EventArgs e)
}
        BlackHeuristicBvB = BotHeuristics.EXTREME;
        BlackDepth.Text = "7;"
        BlackKingWeight.Text;"-" =
        BlackSoliderWeight.Text;"-" =
```

```
        BlackPMWeight.Text;"-" =
        BlackWinScore.Text;"-" =
        BlackLoseScore.Text;"-" =
        BlackDrawScore.Text;"-" =
{

> ///      summary<
 ///      Black heuristic chosen as adaptive
/> ///      summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void BvBBlackAdaptive_CheckedChanged(object sender, EventArgs e)
}
        BlackHeuristicBvB = BotHeuristics.ADAPTIVE;
        BlackDepth.Text = "7;"
        BlackKingWeight.Text;"-" =
        BlackSoliderWeight.Text;"-" =
        BlackPMWeight.Text;"-" =
        BlackWinScore.Text;"-" =
        BlackLoseScore.Text;"-" =
        BlackDrawScore.Text;"-" =
{

> ///      summary<
 ///      Takes care of the Bot Versus Bot game course
/> ///      summary<
> ///      param name="BotWhite"></param<
> ///      param name="BotBlack"></param<
      private void BvBCourse(Ai BotWhite, Ai BotBlack)
}
 //         holds the color of the current player (white player always starts)
        PlayerColor currentPlayer = PlayerColor.WHITE;
```

```
//          while the game is ongoing (not a draw, white win or black win) and while game mode wasn't changed
and a kill thread command was not received

        while (M.CheckGameState() == GameState.OnGoing && Mode == "BvB" && !killThreads)

}

//              handle the turn of the bots

            if (currentPlayer == PlayerColor.WHITE)

                BotWhite.MakeMove(M);

            else

                BotBlack.MakeMove(M);


            DrawBoard(M.GameBoard);

//              switch the current player to the other player

            currentPlayer = currentPlayer switch

}

                (PlayerColor.WHITE) => PlayerColor.BLACK,

                (PlayerColor.BLACK) => PlayerColor.WHITE,

<= _                throw new Exception("Invalid player"),

;{

{


//          Reset Mode

        if (Mode == "BvB")

}

            Mode;"" =

{


//          Game ened because a result was reached, display the result

        if(M.CheckGameState() != GameState.OnGoing)

}

//              Game over display result

            MessageBox.Show(M.CheckGameState().ToString());

{

//          Thread terminated
```

```
        killThreads = false;
{


> ///      summary<
 ///      Handle setting up BvB (Bot Versus Bot)
/> ///       summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void BvB_Click(object sender, EventArgs e)
}
 //          Terimate all running threads
        if ((BotTurnThread != null && BotTurnThread.IsAlive) || (BvBGameCourseThread != null &&
BvBGameCourseThread.IsAlive))
}
           killThreads = true;
 //             Wait until thread resets the command
           while (killThreads != false)
}
              Thread.Sleep;(250)
{
{


 //          Reset all variables used in PvB/PvP match
        PossibleMovesCoordinates = new List<Coordinates;()<


        MovesList = new List<PieceMove;()<


        PieceToMove = 0;


        midMove = false;


        inChain = false;
```

```
        eatingPiece = 0;

//      Set game mode
        Mode = "BvB;"

//      Create game model
        M = new Model;()

//      Draw board
        DrawBoard(M.GameBoard);

//      Black bot Ai Set with default values
        Ai BotBlack = new Ai(PlayerColor.BLACK, Heuristics.BasicHeuristic, 7, 4, 2, 1, 100, -100, -90);

//      White bot Ai Set with default values
        Ai BotWhite = new Ai(PlayerColor.WHITE, Heuristics.BasicHeuristic, 7, 4, 2, 1, 100, -100, -90);

//      Message in exception thrown if given depth was below 1
        string depthErrorMessage = "Depth must be 1 or higher And must be below " +
MaximumBotDepth.ToString() + " or it will take too long to calculate move;"

//      BLACK BOT SET UP

//      Heuristic function chosen for the bot
        Heuristics.HeuristicFunc BlackChosenHeuristic;

//      If a heuristic that only uses depth is chosen
        bool BlackOnlyDepthMatters = false;

//      Set Black bot according to heuristic chosen
        switch (BlackHeuristicBvB)
}
```

```
            case (BotHeuristics.BASIC):
                BlackChosenHeuristic = Heuristics.BasicHeuristic;
                break;

            case (BotHeuristics.EXTREME):
                BlackChosenHeuristic = Heuristics.ExtremeHeuristic;
                BlackOnlyDepthMatters = true;
                break;



            case (BotHeuristics.ADAPTIVE):
                BlackChosenHeuristic = Heuristics.AdaptiveHeuristic;
                BlackOnlyDepthMatters = true;
                break;

            default:
                throw new NotSupportedException("Heuristic type is not supported");



{

 //         Set black Ai with wanted parameters
        try
}
 //            If given depth is below 1 Or above maximum bot depth
            if (int.Parse(BlackDepth.Text) <= 0 || int.Parse(BlackDepth.Text) >= MaximumBotDepth)
                throw new Exception(depthErrorMessage);

            if (BlackOnlyDepthMatters)
}
 //                Bot parameters set as user indicated
                BotBlack = new Ai(PlayerColor.BLACK, BlackChosenHeuristic, int.Parse(BlackDepth.Text), 0, 0, 0, 0,
0, 0);
```

```
{
        else
}
//          Bot parameters set as user indicated
          BotBlack = new Ai(PlayerColor.BLACK, BlackChosenHeuristic, int.Parse(BlackDepth.Text,(
          int.Parse(BlackKingWeight.Text), int.Parse(BlackSoliderWeight.Text),
int.Parse(BlackPMWeight.Text), int.Parse(BlackWinScore.Text), int.Parse(BlackLoseScore.Text),
int.Parse(BlackDrawScore.Text);(
{


{
      catch (Exception except)
}
//        Inform user of the error cause
        if (except.Message == depthErrorMessage)
}
//          Error was due to depth given being below 1
          MessageBox.Show(depthErrorMessage + " Default ai values set (Black Ai)");
{


        else
//          Error was due to invalid type of value entered
          MessageBox.Show("1 or more values entered were invalid, Default ai values set (Black Ai)");


//        Set values to default
        BlackDepth.Text = "7;"
        BlackKingWeight.Text = "4;"
        BlackSoliderWeight.Text = "2;"
        BlackPMWeight.Text = "1;"
        BlackWinScore.Text = "100;"
        BlackLoseScore.Text = "-100;"
        BlackDrawScore.Text = "-90;"


//        default values if 1 or more of the parameter values were invalid
```

```
        BotBlack = new Ai(PlayerColor.BLACK, BlackChosenHeuristic, 7, 4, 2, 1, 100, -100, -90);
{

//      WHITE SET UP

//      Heuristic function chosen for the White bot
      Heuristics.HeuristicFunc WhiteChosenHeuristic;

//      If a heuristic that only uses depth is chosen
      bool WhiteOnlyDepthMatters = false;

//      Set White bot according to heuristic chosen
      switch (WhiteHeuristicBvB)
}

        case (BotHeuristics.BASIC):
            WhiteChosenHeuristic = Heuristics.BasicHeuristic;
            break;

        case (BotHeuristics.EXTREME):
            WhiteChosenHeuristic = Heuristics.ExtremeHeuristic;
            WhiteOnlyDepthMatters = true;
            break;



        case (BotHeuristics.ADAPTIVE):
            WhiteChosenHeuristic = Heuristics.AdaptiveHeuristic;
            WhiteOnlyDepthMatters = true;
            break;

        default:
            throw new NotSupportedException("Heuristic type is not supported");
```

```
{

//        Set White Ai with wanted parameters
        try
}
//            If given depth is below 1 Or above maximum bot depth
          if (int.Parse(WhiteDepth.Text) <= 0 || int.Parse(WhiteDepth.Text) >= MaximumBotDepth)
             throw new Exception(depthErrorMessage);

          if (WhiteOnlyDepthMatters)
}
//                Bot parameters set as user indicated
             BotWhite = new Ai(PlayerColor.WHITE, WhiteChosenHeuristic, int.Parse(WhiteDepth.Text), 0, 0, 0,
0, 0, 0);
{
          else
}
//                Bot parameters set as user indicated
             BotWhite = new Ai(PlayerColor.WHITE, WhiteChosenHeuristic, int.Parse(WhiteDepth.Text,(
             int.Parse(WhiteKingWeight.Text), int.Parse(WhiteSoliderWeight.Text),
int.Parse(WhitePMWeight.Text), int.Parse(WhiteWinScore.Text), int.Parse(WhiteLoseScore.Text),
int.Parse(WhiteDrawScore.Text);(
{

{
        catch (Exception except)
}
//            Inform user of the error cause
          if (except.Message == depthErrorMessage)
}
//                Error was due to depth given being below 1
             MessageBox.Show(depthErrorMessage + " Default ai values set (White Ai)");
{
```

```
            else
//                Error was due to invalid type of value entered
              MessageBox.Show("1 or more values entered were invalid, Default ai values set (White Ai)");

//              Set values to default
            WhiteDepth.Text = "7;"
            WhiteKingWeight.Text = "4;"
            WhiteSoliderWeight.Text = "2;"
            WhitePMWeight.Text = "1;"
            WhiteWinScore.Text = "100;"
            WhiteLoseScore.Text = "-100;"
            WhiteDrawScore.Text = "-90;"

//              default values if 1 or more of the parameter values were invalid
            BotWhite = new Ai(PlayerColor.WHITE, WhiteChosenHeuristic, 7, 4, 2, 1, 100, -100, -90);
{




        BvBGameCourseThread = new Thread(() => BvBCourse(BotWhite, BotBlack));
        BvBGameCourseThread.Start;()
{

> ///       summary<
 ///       Display game instructions to the user for BvB mode
/> ///       summary<
> ///       param name="sender"></param<
> ///       param name="e"></param<
      private void InstructionsBvB_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
}
        MessageBox.Show("BvB (Bot Versus Bot) is a mode where 2 Bots play checkers against each other. \n
\n+ "
```

```
"                    Pick Heuristic function type: \n+ "

"                    Basic - takes all parameters and only uses them.\n+ "

"                    Extreme - only takes depth and takes into consideration more parameters than the ones
present.\n+ "

"                    Adaptive - only takes depth, Scores relative to the progress of the game, For example the
more soliders the bot has the less each solider is worth to it. \n\n+ "

"                    It is possible to change the bot's heuristic function stats, the default values are the ones in the
text boxes below and may be changed. \n \n+ "

"                    Depth - How many moves ahead will the Bot go over, Recommended under 9. \n+ "

"                    King Weight - How many points is a king piece worth for the bot. \n+ "

"                    Solider Weight - How many points is a Solider piece worth for the bot. \n+ "

"                    PM Weight - How many points is a possible move worth for the bot. \n+ "

"                    Win Score - How many points is a win worth for the bot. \n+ "

"                    Lose Score - How many points is a lose worth for the bot. \n+ "

"                    Draw Score - How many points is a draw worth for the bot.\n \n+ "

"                    Max Depth is set as " + (MaximumBotDepth - 1).ToString() + " for preformance reasons;(".

{


-------------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------------------

 //      Other

-------------------------------------------------------------------------------------------------------------------------------------------//
-------------------------------------------------------------------



> ///      summary<

 ///      Draws an empty checkers board

/> ///      summary<

      private void DrawEmptyBoard(System.Drawing.Graphics formGraphics)

}

 //          Color of painted squares

          System.Drawing.SolidBrush myBrushEmpty = new
System.Drawing.SolidBrush(System.Drawing.Color.Black);


 //          Draw board (black squares)
```

```
        for (int row = 0; row < numOfSquares; row++)
}
          for (int col = 0; col < numOfSquares; col++)
}
//              if its a black square draw it, otherwise color is already white (background is white)
             if (((col % 2 != 0) && (row % 2 == 0)) || ((col % 2 == 0) && (row % 2 != 0)))
}
//                  draw black square
                  formGraphics.FillRectangle(myBrushEmpty, new Rectangle(col * size + distanceFromEdge, row *
size + distanceFromEdge, size, size));
{
{
{
        myBrushEmpty.Dispose;()

//          Draw border around board
        Pen pen = new Pen(Color.FromArgb(255, 0, 0, 0));
        formGraphics.DrawLine(pen, distanceFromEdge, distanceFromEdge, size * numOfSquares +
distanceFromEdge, distanceFromEdge);
        formGraphics.DrawLine(pen, distanceFromEdge, distanceFromEdge, distanceFromEdge, size *
numOfSquares + distanceFromEdge);
        formGraphics.DrawLine(pen, distanceFromEdge, size * numOfSquares + distanceFromEdge, size *
numOfSquares + distanceFromEdge, size * numOfSquares + distanceFromEdge);
        formGraphics.DrawLine(pen, size * numOfSquares + distanceFromEdge, distanceFromEdge, size *
numOfSquares + distanceFromEdge, size * numOfSquares + distanceFromEdge);
        pen.Dispose;()
{

> ///      summary<
///      Override OnPaint to paint empty checkers board to screen when the Form loads up
/> ///      summary<
> ///      param name="e"></param<
      protected override void OnPaint(PaintEventArgs e)
}
        base.OnPaint(e);
```

```
        DrawEmptyBoard(e.Graphics);
{

> ///      summary<
 ///      Display Instructions on how to play checkers
/> ///       summary<
> ///      param name="sender"></param<
> ///      param name="e"></param<
      private void CheckersInstructions_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
}
        MessageBox.Show("Checkers Game Instructions \n \n+ "
"                    Set Up - \nWhen a gamemode is chosen (BvB/PvB/PvP) the pieces are placed in the 12 dark spots on the bottom and top of the board.\n+ "
"                    Each of these three rows has a total of 4 checkers.\n+ "
"                    The rows at the top and the bottom are called the King Rows. \n \n+ "

"                    Interaction - \nTo move a piece click on it and than click on one of the green squares representing the possible moves the clicked piece can make \n \n+ "

"                    Rules- \nThe opponent with the White pieces moves first. \n+ "
"                    Solider pieces may only move one diagonal space forward(towards their opponents pieces). \n+ "
"                    Pieces must stay on the dark squares.\n+ "
"                    To eat an opposing piece, jump over it by moving two diagonal spaces in the direction of the opposing piece. \n+ "
"                    A piece may jump forward over an opponent's pieces in multiple parts of the board to eat them (chain eating, may also be stopped at any point using the EndTurn button that will apear). \n+ "
"                    the space on the other side of your opponent’s piece must be empty for it to be captured. \n "
+
"                    If a piece reaches the last row on its opponent's side, It is crowned as a king piece (black turns red and white turns light gray). \n+ "
"                    King pieces may still only move one space at a time during a non - eating move. However, they may move diagonally forward or backwards. \n + "
"                    There is no limit to how many king pieces a player may have. \n \n+ "
```

```
"                    The Goal - \n To win one must eat all enemy pieces Or have the enemy have no possible
moves left .\n+ "

"                    A draw is declared if the number of pieces on the board hasn't changed in 40 turns;(" .

{

{

{



using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Checkers
{
   public static class Program
   {
      // Setting up the UserInterface
      public static UserInterface UI = new UserInterface();

      /// <summary>
      ///  The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
         // Set up and Run The User interface
         Application.SetHighDpiMode(HighDpiMode.SystemAware);
         Application.EnableVisualStyles();
         Application.SetCompatibleTextRenderingDefault(false);
         Application.Run(UI);


         // Terimnates all running threads and exists program when application window is closed
         System.Environment.Exit(1);
      }
   }
}
```

סיכום אישי / רפלקציה

שלום אני אמיר וולברג, למדתי במהלך השנה האחרונה הנדסת תוכנה במכללת אורט הרמלין. הפרויקט שמוצג בפניכם הוא הפרויקט גמר שלי, פרויקט אשר שקדתי עליו במשך חודשים תוך כדיי ניסיון לאזן בין העבודה על הפרויקט לבין למידה למבחני המכללה.

בחרתי לשלב בפרויקט מגוון דברים חדשים שלמדתי השנה במסגרת המכללה, כגון עבודה עם ביטים שלמדתי בקורסי סי ושפת סף, עבודה עם מבני נתונים כמו מילון, רשימות ועצי חיפוש שלמדתי בקורס מבנה נתונים ושיקולי יעילות זמן ריצה שלמדתי בקורס סיבוכיות.

מכשול אחד בפן האישי שנתקלתי בו במהלך העבודה היה הקושי לשלב בין העבודה על הפרויקט והלימודים למבחני המכללה במהלך השנה, התגברתי על מכשול זה בעזרת ניהול זמנים נכון והערכה מוצלחת של כמות הזמן שיקח לי לעשות דברים שונים בפרויקט.

מכשול אחד בפן הטכני שנתקלתי בו בפרויקט היה לעבוד עם לוחות ביטים, ההתעסקות היחידה שהייתה לי עם פעולות על ביטים לפני הייתה במקצועות סי ושפת סף במכללה שעזרו לי עם ידע כללי אך לממש את הידע הזה בפרויקט ובשפה גבוהה יותר (סי שארפ) לא היה קל. ולמרות זאת אחרי שחילקתי את העבודה שלי והצבתי כל פעם מטרה קטנה אחרת שיכלתי להתרכז בה הצלחתי לאט לאט להתרגל, להבין ולהשתפר בעבודה עם ביטים עד שהצלחתי לעשות את כל מה שתכננתי לעשות.

למדתי מהפרויקט איך לעבוד ולהתנהל עם פרויקט בסדר גודל רציני וכיצד לחלק אותו למחלקות בצורה מודולרית. כמו כן למדתי הרבה על אלגוריתמים של בינה מלאכותית, גם כאלה שלא יצא לי להשתמש בהם בסוף אבל עדיין קראתי עליהם והבנתי אותם. אני גם מאמין שהשתפרתי בחקירת ומציאת אלגוריתמים במרשתת.

אם הייתי מתחיל את הפרויקט מחדש היום הייתי מתכנן וקובע את היעדים שלי בצורה יותר ברורה בהתחלה, קובע דדליין לכל חלק בפרויקט ועומד בו בצורה יותר טובה. בנוסף לכל גם הייתי מתחיל מלקרוא על כל האלגוריתמים שאני רוצה להשתמש בהם עוד לפני כתיבת קוד הפרויקט.

לסיכום אני מרגיש שלמדתי והעשרתי את הידע שלי הרבה בפרויקט הזה ומאוד נהניתי, גיליתי שיש הרבה יותר לבינה מלאכותית משחשבתי והשתפרתי בלחשוב מחוץ לקופסא בשביל לפתור בעיות יעילות ואלגוריתמים.

בבליוגרפיה

קישור	מקור
https://stackoverflow.com/questions/20009796/transposition-tables	StackOverflow
https://en.wikipedia.org/wiki/Bitboard	וויקיפדיה
https://www.geeksforgeeks.org/minimax-algorithm-in-game-/theory-set-1-introduction	GeeksForGeeks
https://pages.mini.pw.edu.pl/~mandziukj/PRACE/es_init.pdf	Jacek Ma´ndziuk∗ , Magdalena Kusiak and Karol Wal¸edzik
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning	וויקיפדיה