

עבודת גמר

לקבלת תואר

טכנאי תוכנה

הנושא: משחק "דמקה סינית"

המגיש: מאור מילקנדוב.

**המנחים: מיכאל צ'רנובילסקי, אלון
חיימוביץ'.**

תש"ף

אפריל 2020

תוכן עניינים

7	מבוא	1.
7	מטרה	1.1.
7	תאור המערכת (תקציר, כולל רציונל)	1.2.
8	שפת התכנות ופירוט סביבת העבודה והכלים	1.3.
9	מפרטי תוכנה	2.
9	תאור כללי	2.1.
13	ניסוח וניתוח הבעיה האלגוריתמית	2.2.
14	פיתוח הפתרון ויישומו	2.3.
22	תאור אלגוריתמים	2.4.
29	מבנה נתונים	2.5.
31	תכנון	3.
32	חלוקה למודולים	3.1.
50	רשימת הנתונים	3.2.
54	מדריך למשתמש	4.
58	ביבליוגרפיה	5.
59	נספח – קוד התוכנית	6.

הצעת נושא פרויקט

שם המנחה: מיכאל צ'רנובלסקי.

שם הסטודנט: מאור מילקנדוב.

מספר ת.ז.: 212256291

תאריך: 17/11/2019.

נושא הפרויקט: משחק חשיבה דמקה סינית.

רקע תיאורטי:

היסטוריה:

דמקה סינית או הלמה כוכב הוא משחק לוח המיועד לשניים עד שישה שחקנים. המשחק הוא גרסה של המשחק הלמה שפותח בשלהי המאה ה-19, והמטרה בו היא להעביר את כלי המשחק, על פי רוב גולות או יתדות, לצד המנוגד של לוח המשחק.

למרות שמו, המשחק דמקה סינית לא הומצא בסין או באסיה כלל. הוא אף אינו גרסה של המשחק דמקה. המשחק הומצא בגרמניה בשנת 1892 תחת השם "שטרן-הלמה" (הלמה כוכב), כגרסה למשחק הלמה האמריקאי. שמו הנוכחי ניתן לו בארצות הברית ב-1928, כאמצעי שיווקי של ביל וג'ק פרסמן, אך בטרם זאת נקרא דמקה הופ צ'ינג.



לוח דמקה סינית עם שש ערכות משחק

תיאור המשחק:

לוח המשחק עשוי 121 שקערוריות (גומות) המסודרות בצורת מגן דוד, כאשר המשולשים הקטנים המרכיבים את פינותיו מכילים עשר שקערוריות כל אחד. בתחילת המשחק מציב כל שחקן עשרה כלי משחק מאותו צבע באחד מקדקדיו של הלוח.

כל שחקן בתורו מזיז כלי משחק אחד, או לשקערורית סמוכה פנויה או תוך כדי דילוג מעל כלי משחק אחד אחר, אין זה משנה של איזה שחקן, אל השקערורית הפנויה הבאה אחריה.

אסור לדלג מעל שני כלי משחק צמודים, אך ניתן לבצע באותו מהלך מספר דילוגים, ובתנאי שבכל דילוג בשרשרת יונח כלי המשחק בשקערורית פנויה.

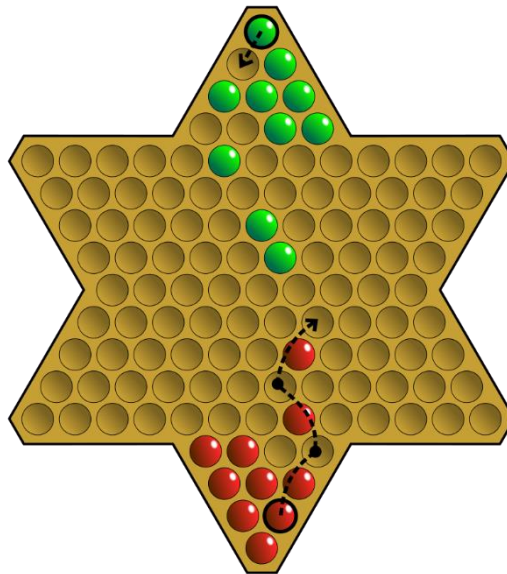
הדילוג יכול להתבצע בכל כיוון, וכלי המשחק שמדלגים מעליו אינו מושפע מכך.

מצבי משחק:

- כאשר משחקים שני שחקנים, ניתן לשחק עם ערכה אחת, עם שתיים או עם שלוש, ובהתאם להציב את כלי המשחק. הצבה שונה של כלי המשחק דורשת טקטיקה שונה, שכן שחקן המניע את כליו לעבר פינה נגדית התפוסה בכליו של היריב נוהג אחרת משחקן המניע כליו לפינה התפוסה בכליו שלו עצמו או לפינה ריקה.
- כאשר משחקים ארבעה שחקנים דומה למשחק בשישה, אלא ששתי פינות אינן בשימוש.
- כאשר ישנם שלושה שחקנים, ניתן לשחק עם ערכה אחת או עם שתי ערכות לכל שחקן. הצבת כלי המשחק במקרה של ערכה אחת לשחקן מתבצעת כך שכל שחקן נע לעבר פינה נגדית ריקה. במקרה של שתי ערכות מוצבים כלי המשחק של כל שחקן בפינות מנוגדות.
- כאשר משחקים חמישה שחקנים נוהג שהשחקן המזיז את כליו אל עבר הפינה הריקה, דבר המעניק לו יתרון, יהיה השחקן החלש ביותר.

סיום המשחק:

מטרת המשחק היא להעביר את כל עשרת כלי המשחק של השחקן מפינה משולשת אחת אל הפינה המשולשת הנגדית בטרם יעשה זאת שחקן אחר. המנצח הוא הראשון שמצליח להעביר את כל כליו לפינה הנגדית.



איור עם מהלכים אפשריים: הזזה לשקע רורית סמוכה פנויה (ירוק) ושרשרת דילוגים (אדום)

מטרות:

1. בניית לוח המשחק (צורת מגן דוד) בצורה חכמה.
2. אפשרות של שניים עד שישה שחקנים אמיתיים לשחק אחד נגד השני - על אותו המחשב, או על מחשבים שונים.
3. אפשרות שהמשתמש ישחק כנגד המחשב במשחק של אחד על אחד.

דרישות מערכת:

1. מימוש אלגוריתם חכם ויעיל של השחקן הממוחשב.
2. מימוש אלגוריתם חכם ויעיל אשר מטפל בכל חוקי המשחק.
3. מימוש אלגוריתם היכול לנהל את תורות השחקנים אם הם על אותו המחשב.
4. מימוש אלגוריתם היכול לנהל את תורות השחקנים אם הם על מחשבים נפרדים.
5. מימוש אלגוריתם חכם לשחקן ממוחשב במשחק של אחד על אחד (שחקן ממוחשב נגד שחקן אמיתי).
6. המשחק צריך להיות בנוי לפי כללי הנדסת תוכנה, תוך שימוש בתכנות מונחה עצמים הכולל בניית מחלקות לעצמים השונים במשחק (למשל שחקן, כלים).

משאבים:

סביבת פיתוח:

- NetBeans IDE 8.2
- Java Development Kit 8 Update 181

סביבת משתמש:

- Java SE Runtime Environment 8

לוח זמנים:

משימה	תאריך יעד
הגשת הצעת נושא פרויקט	01/11/2019
הגשת נוסח ראשוני – הצעת פרויקט	12/11/2019
הגשת נוסח סופי – הצעת פרויקט	19/11/2019
הגשה למשרד החינוך ואישור סופי	01/12/2019
סיום בניית לוח משחק	08/12/2019
סיום בניית אפשרות של עד 6 שחקנים לשחק אחד נגד השני על אותו מחשב	15/12/2019
סיום בניית אפשרות של עד 6 שחקנים לשחק אחד נגד השני על מחשבים שונים	15/01/2019
הגשת דו"ח ביניים – התקדמות הפיתוח עד כה	01/02/2019
שיודע לשחק נגד שחקן אחד סיום בניית	20/03/2019
סיום פיתוח ראשוני	01/04/2019
סיום תיקון שגיאות לאחר בדיקות	15/04/2019
הגשת תיק מלווה של פרויקט	01/05/2019
הגנה על עבודת הגמר – פנימית	מאי 2019
הגנה על עבודת הגמר - חיצונית	מאי 2019

1. מבוא

1.1 מטרה

לבחון את הידע והכישורים התכנותיים שלי בפרוייקט בסדר גודל כזה. לפתח אלגוריתמים חכמים אשר יפתרו בעיות שונות תוך כדי שימוש בכלי תכנות וחשיבה שברשותי. בנוסף בניית מודל של אינטלגנציה מלאכותית, דבר שלא נתקלתי בו בעבר.

1.2 תאור המערכת

לוח המשחק עשוי 121 שקערוריות (גומות) המסודרות בצורת מגן דוד, כאשר המשולשים הקטנים המרכיבים את פינותיו מכילים עשר שקערוריות כל אחד. בתחילת המשחק מציב כל שחקן עשרה כלי משחק מאותו צבע באחד מקדקדיו של הלוח.

כל שחקן בתורו מזיז כלי משחק אחד, או לשקערורית סמוכה פנויה או תוך כדי דילוג מעל כלי משחק אחד אחר, אין זה משנה של איזה שחקן, אל השקערורית הפנויה הבאה אחריה.

אסור לדלג מעל שני כלי משחק צמודים, אך ניתן לבצע באותו מהלך מספר דילוגים, ובתנאי שבכל דילוג בשרשרת יונח כלי המשחק בשקערורית פנויה.

הדילוג יכול להתבצע בכל כיוון, וכלי המשחק שמדלגים מעליו אינו מושפע מכך.

המערכת כוללת 3 אפשרויות משחק:

1. לשחק על אותו המחשב (בין 2 עד 6 שחקנים).
2. לשחק אונליין עם מחשבים אחרים (בין 2 עד 6 שחקנים).
3. שחקן אנושי נגד שחקן מחשב תוך כדי בחירת סוג הAI של המחשב – AlphaBeta או Monte Carlo.

מטרת המשחק היא להעביר את כל עשרת כלי המשחק של השחקן מפינה משולשת אחת אל הפינה המשולשת הנגדית בטרם יעשה זאת שחקן אחר. המנצח הוא הראשון שמצליח להעביר את כל כליו לפינה הנגדית.

1.3 שפת התכנות ופירוש סביבת העבודה והכלים

סביבת פיתוח:

- NetBeans IDE 8.2
- Java Development Kit 8 Update 181
- GlassFish Server 4.1.1

סביבת משתמש:

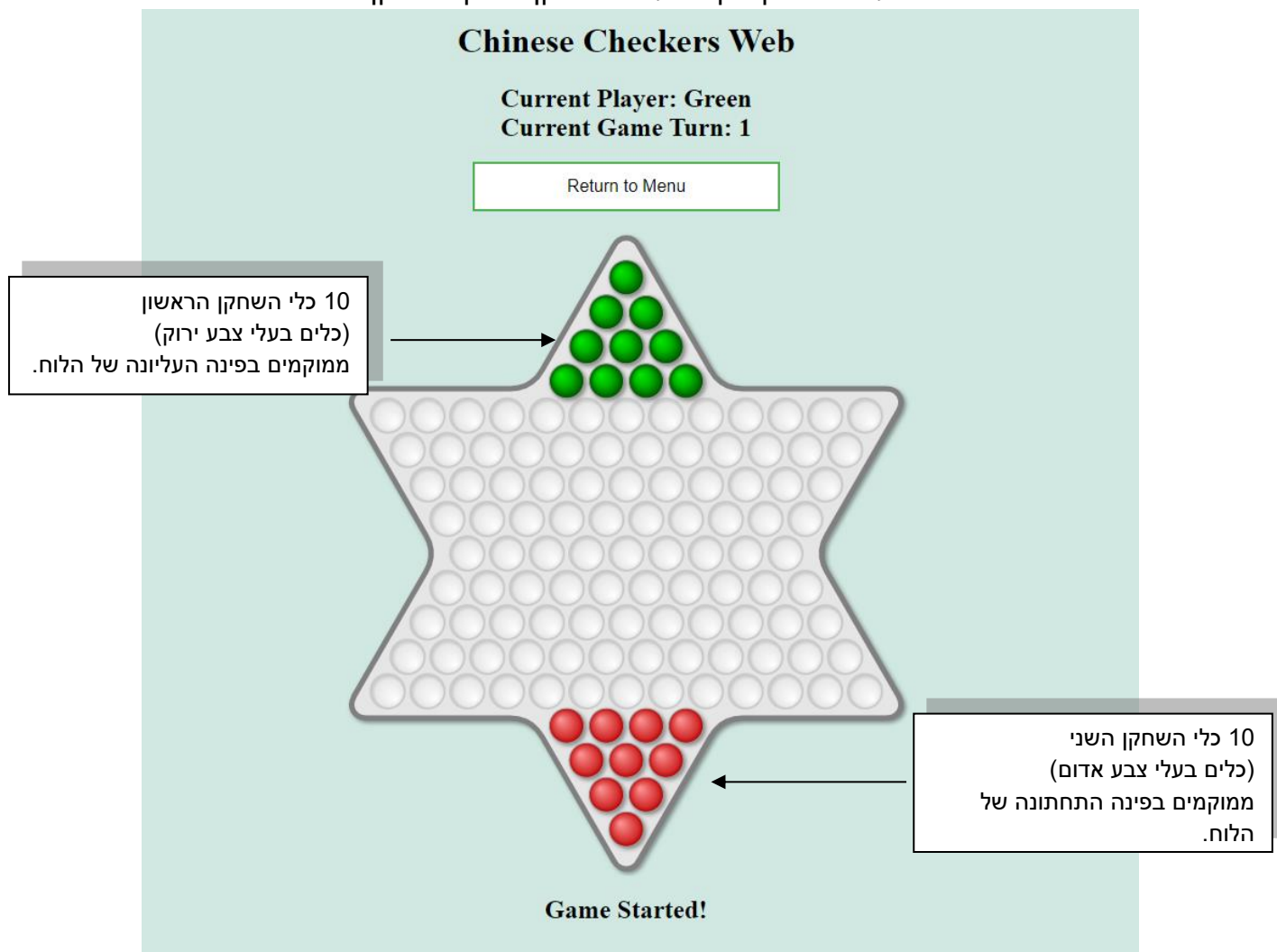
- Java SE Runtime Environment 8
- Google Chrome / Firefox Mozilla

2. מפרטי תוכנה

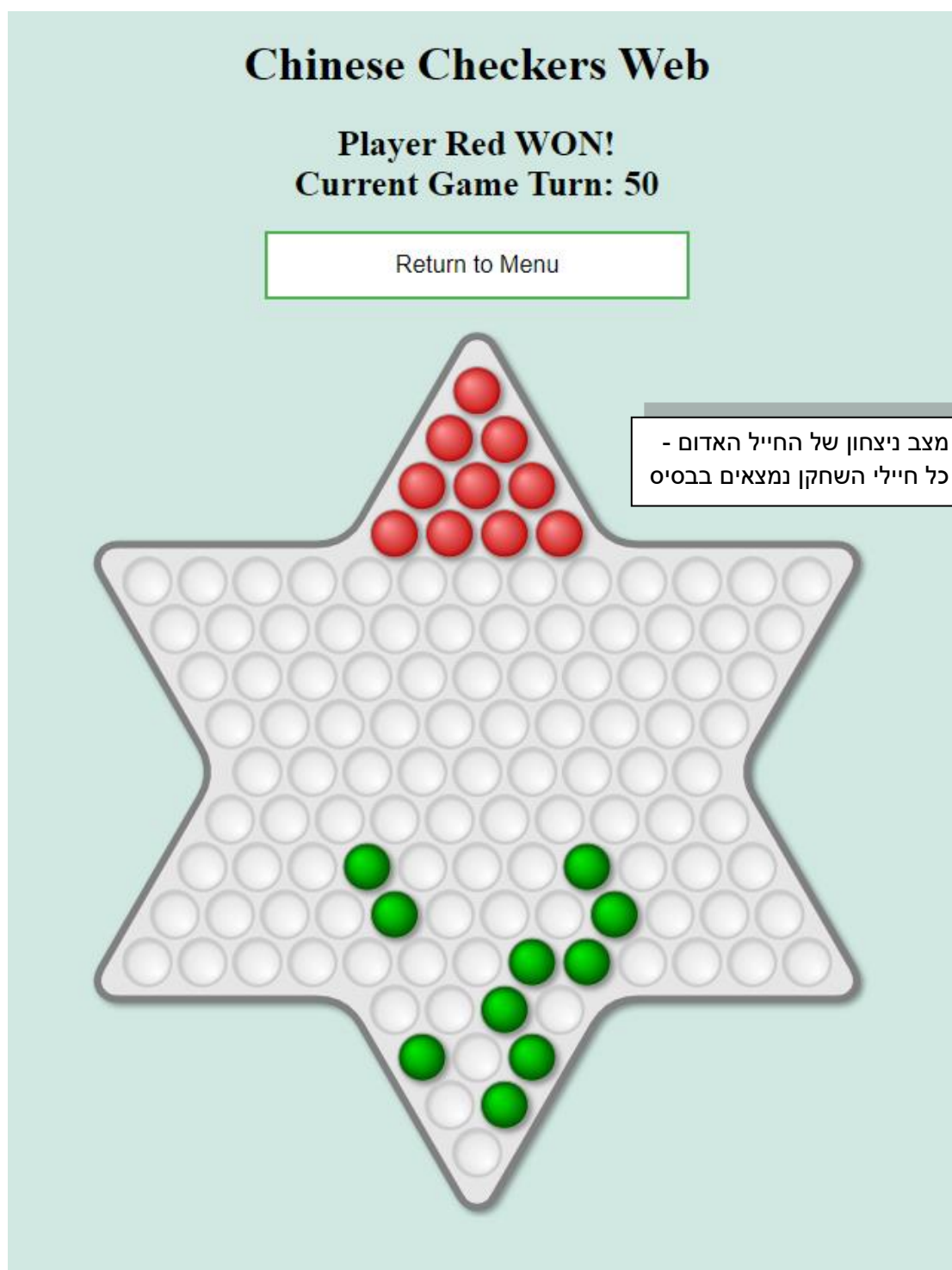
2.1 תאור כללי

לוח המשחק עשוי 121 שקערוריות (גומות) המסודרות בצורת מגן דוד, כאשר המשולשים הקטנים המרכיבים את פינותיו מכילים עשר שקערוריות כל אחד. בתחילת המשחק מציב כל שחקן עשרה כלי משחק מאותו צבע באחד מקדקדיו של הלוח.

*מראה הלוח בתחילת עבור משחק לוקאלי עבור השחקן הירוק והשחקן האדום



המשחק מסתיים כאשר כל חייליו של אחד מהשחקנים נמצא באיזור ההתחלתי של השחקן היריב.

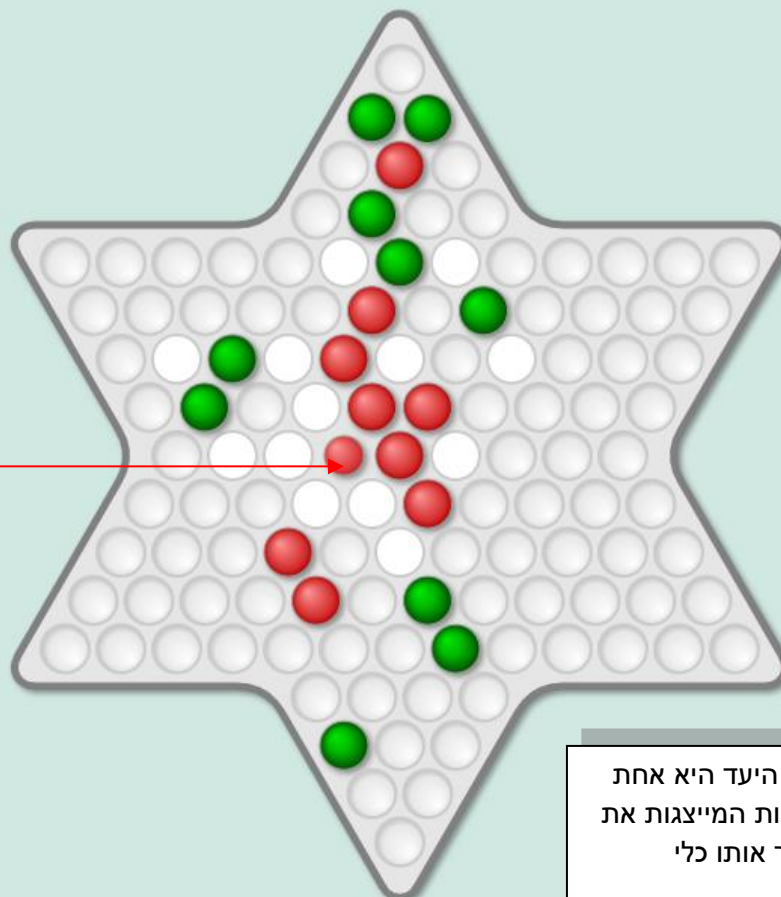


ביצוע מהלך בתוך המערכת מתנהל בצורה הבאה:
כאשר מגיע תורו של השחקן הוא בוחר בחייל שאותו הוא רוצה להזיז (בחירת החייל מתבצעת באמצעות המקש השמאלי של העכבר) ולאחר מכן מוצגות בפניו כל האפשרויות החוקיות של תזוזת החייל.

Chinese Checkers Web

Current Player: Red
Current Game Turn: 22

Return to Menu



בחירת החייל
שברצוננו להזיז.

בחירת משבצת היעד היא אחת
מהנקודות הלבנות המייצגות את
מהלך חוקי עבור אותו כלי
שנבחר.

מבט על המסך הראשוני של האתר:



1. התחלת משחק של בוט.

2. התחלת משחק אונליין.

3. הצטרפות למשחק אונליין קיים.

4. התחלת משחק לוקאלי על המחשב.

הסבר מפורט על כל אחד מהמצבים יהיה במדריך למשתמש.

2.2 ניסוח וניתוח הבעיה האלגוריתמית

במהלך בניית הפרוייקט נתקלתי במספר בעיות אלגוריתמיות אשר היו מהותיות לקיום המערכת. כעת אמנה אותם, אנתח אותם ובפרק הבא אציע פיתרון לבעיות.

בעיה אלגוריתמית מספר 1: תיאום מערכת לבחירות השחקן

לפני כל התחלת משחק, השחקן נתקל ב-4 אפשרויות משחק שונות:

1. התחלת משחק של בוט.

2. התחלת משחק אונליין.

3. הצטרפות למשחק אונליין קיים.

4. התחלת משחק לוקאלי על המחשב.

בנוסף – לפעולות 1, 2, 4 יש תתי בחירות נוספות:

- אם השחקן בוחר במשחק של בוט, יש לדעת האם הוא רוצה להשתתף במשחק או שמא הוא מעוניין לצפות בבוט נגד בוט.
- אם השחקן רוצה להצטרף למשחק אונליין או להתחיל משחק לוקאלי על המחשב, יש לשאול אילו צבעים הוא רוצה שישתתפו במשחק.

אם השחקן בוחר ב-3 (בהצטרפות למשחק אונליין קיים) יש להציג לו רשימה של כל המשחקים הקיימים כעת ודרושים להם שחקנים נוספים על מנת להתחילם. כמו-כן יש לטפל במצב בו הלווי התמלא/נסגר לפני שהשחקן יכל להצטרף למשחק.

בעיה אלגוריתמית מספר 2: הצגת כל אפשרויות התזוזה עבור כלי שנבחר

כל מערכת של משחק נשנעת על חוקי המשחק, כי בלי החוקים המערכת חסרת ערך ומשמעות. במשחק "דקמה סינית" המשחק מתקדם באמצעות מהלכים, לכל שחקן בתורו יש מהלך ובו הוא מזיז את אחד מחייליו למשבצת מסויימת בלוח המשחק. על המערכת לטפל בכל לחיצה של כלי מסוים של השחקן, ולהציג לו את כל אפשרויות התזוזה החוקיות עבור כלי זה.

בעיה אלגוריתמית מספר 3: אלגוריתם לשחקן ממוחשב (artificial intelligence)

שחקן ממוחשב הינו שחקן הפועל על דעת עצמו ולא בעזרת החלטות המשתמש. כל החלטותיו של המשחקן הממוחשב נקבעות על פי שיקולים מתמטיים, שיקולים מתמטיים אלו נקבעו מראש באלגוריתם בצורה כזו שהשחקן הממוחשב יבצע את המהלך הטוב ביותר. לשם כך על השחקן הממוחשב להתחשב בנתוני המצב הקיים של לוח המשחק ולהגיב בהתאם לשינויים המתרחשים במהלך המשחק כך שבכל שלב יהיה בידו המהלך הטוב ביותר עבורו.

על המערכת לספק 3 אפשרויות לשחקן ממוחשב: Alpha Beta, AlphaBeta Hard, Monte Carlo.

בעצם בעיה זו מכילה 2 אלגוריתמים מרכזיים שהם Alpha Beta ו-Monte Carlo.

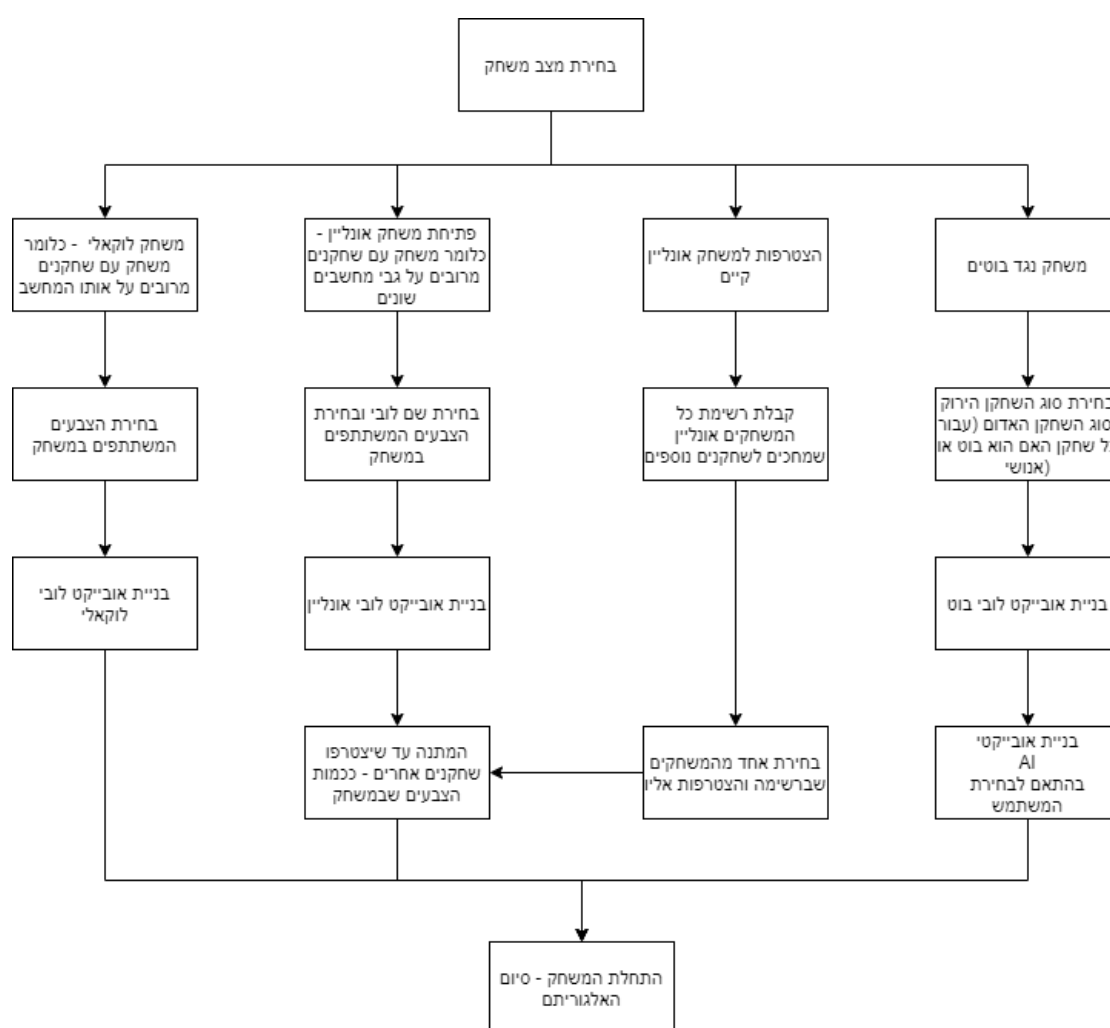
2.3 פיתוח הפתרון ויישומו

פיתוח פתרון לבעיה אלגוריתמית מספר 1 ויישומו: תיאום מערכת לבחירות השחקן

לשם פתרון בעיה זו החלטתי שיהיה על המערכת לייצג מחלקת "לובי" – Lobby שתהיה אבסטרקטית, ותכיל את שלושת סוגי הלובי האפשריים: לובי עבור משחק עם בוט, לובי עבור משחק אונליין ולובי עבור משחק לוקאלי (על אותו המחשב). מחלקת לובי מכילה את מחלקת Board שהיא מכילה מערך עם שחקני המשחק.

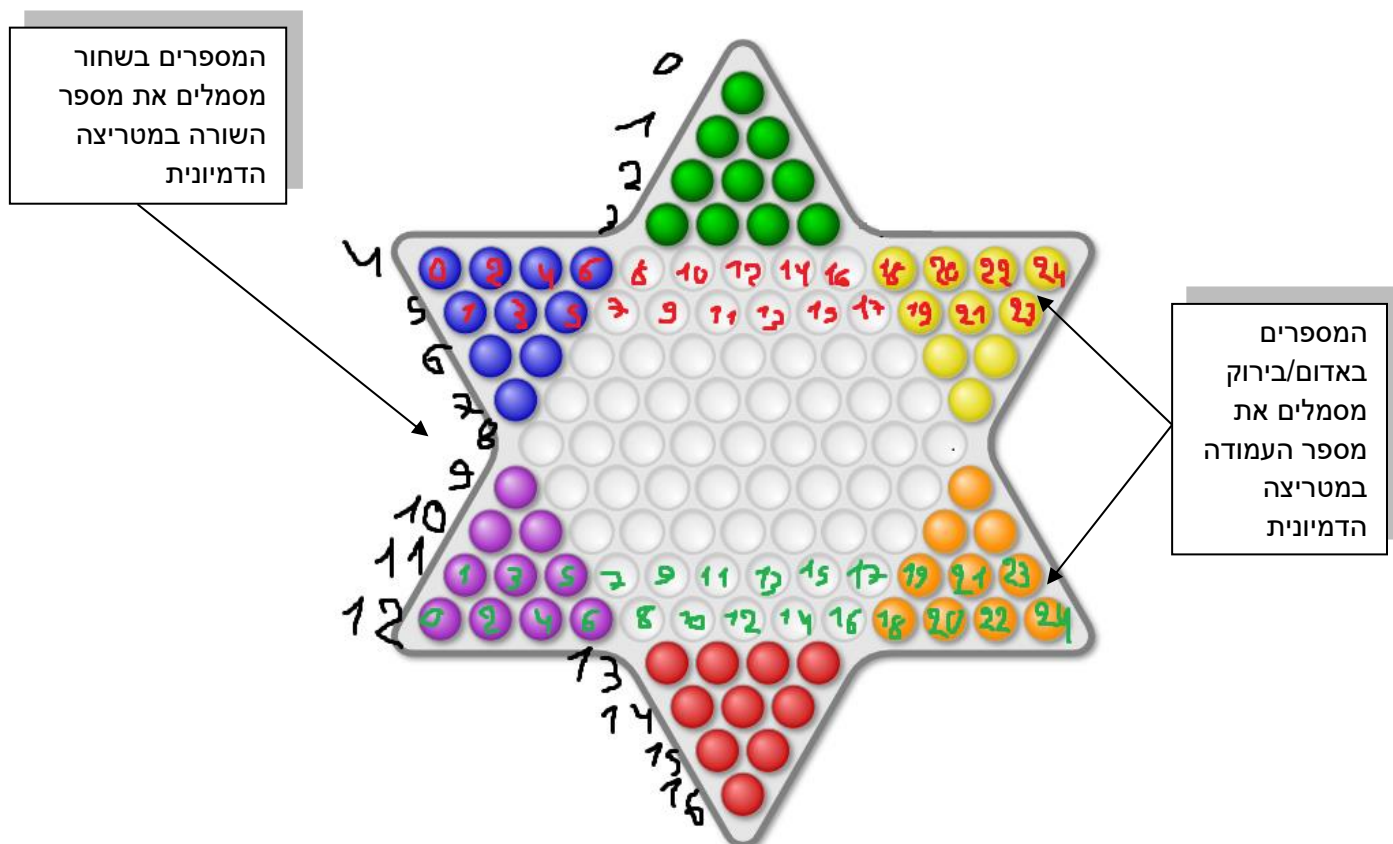
לאחר בחירת המשתמש עבור סוג המשחק, מתחיל המשחק עבור כיוון השעון כאשר השחקן הירוק (העליון) הוא השחקן הפותח את המשחק.

תרשים זרימת המערכת לבחירת מצב משחק:



פיתוח פתרון לבעיה אלגוריתמית מספר 2: הצגת כל אפשרויות התזוזה עבור כלי שנבחר

במשחק "דמקה סינית" כל חייל יכל להתקדם בכל אחד מהכיוונים האפשריים שלידו (סך הכל שישה כיוונים). הבעיה בלוח המשחק היא שכל שורה מכילה מספר עמודות שונות מהשורה הקודמת לה או שלאחריה, כפי שניתן לראות באיור הבא:

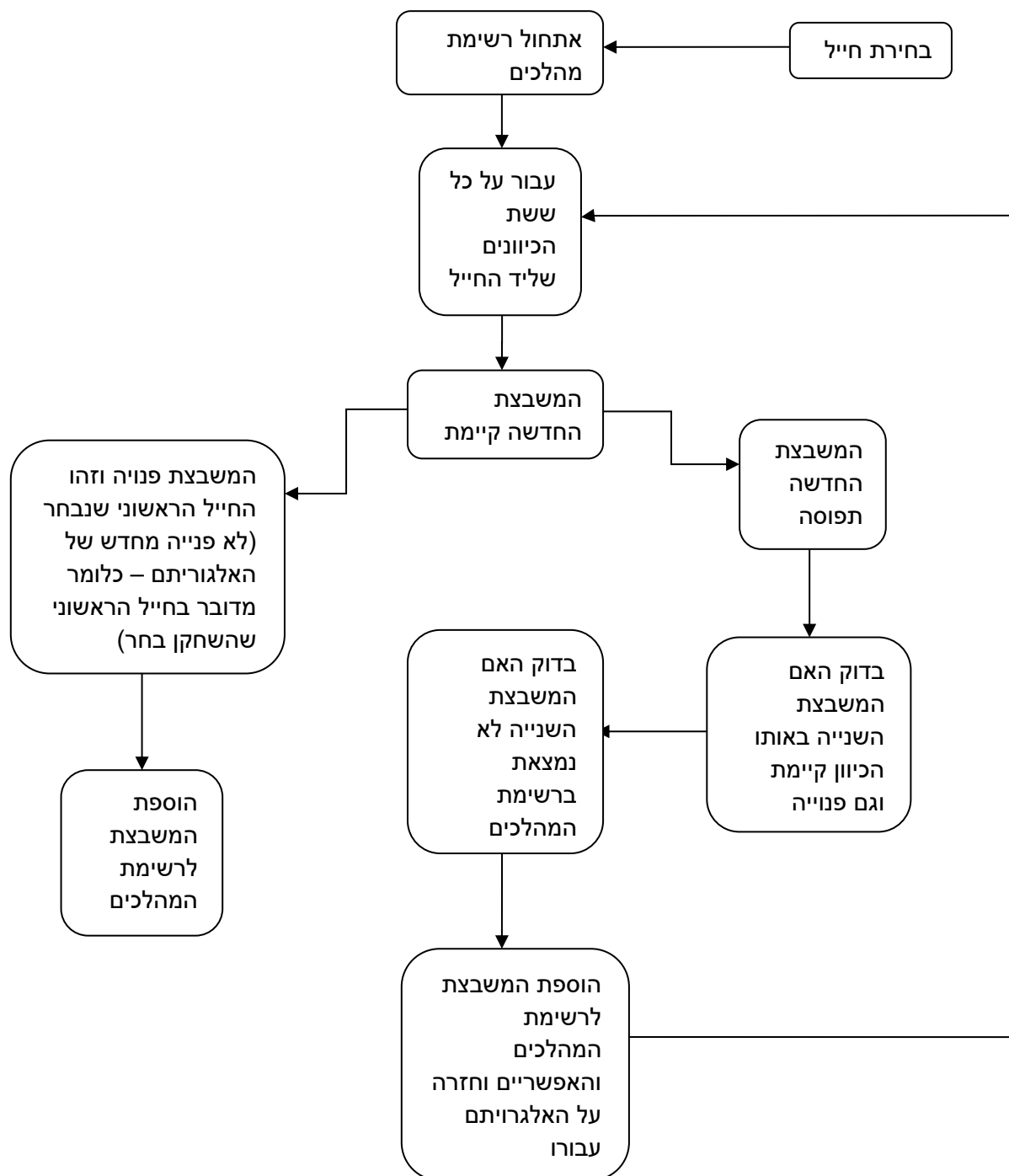


בשביל לפתור את בעיה זו הייתי צריך להכין "מסכה" – אותה הכנתי מביטים שתהווה מעין מטריצה דמיונית של גומות המשחק. במסיכה, כל ביט דלוק מייצג תא הקיים בלוח המשחק, וכל ביט כבוי מייצג תא שאינו קיים בלוח המשחק. כך אני יכל לעבור עבור מיקום נתון על כל אחד מששת הצדדים שלצידו, ולבדוק האם כל אחד מהם באמת קיים על לוח המשחק או לא.

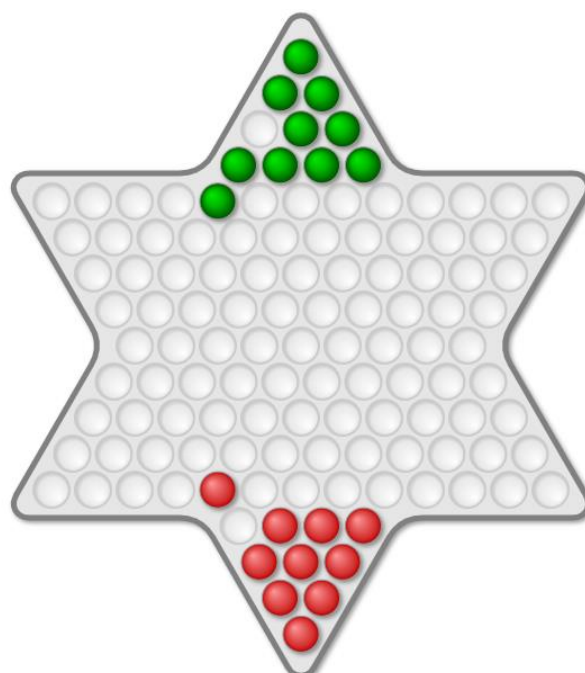
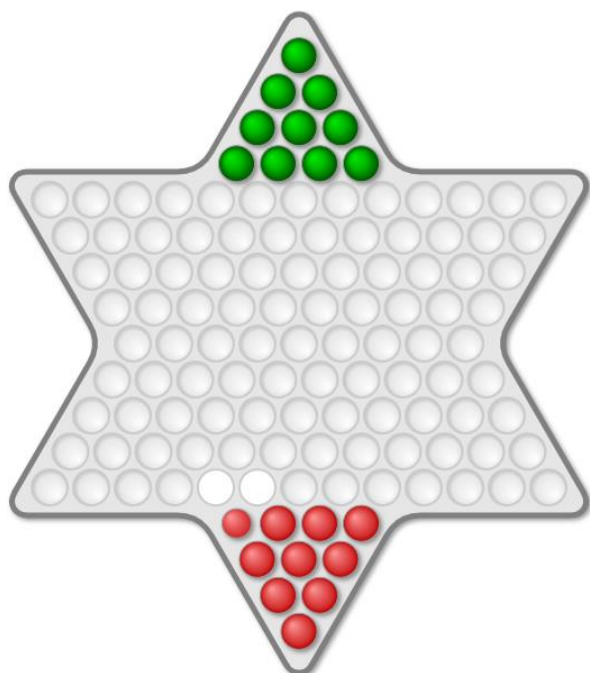
בנוסף, השתמשתי במטריצת כיוונים בגודל 6, כאשר כל מערך מכיל את היסט הכיוון הרצוי:

היסט שורה	היסט עמודה	כיוון רצוי במילים
-1	-1	למעלה-שמאלה
-1	1	למעלה-ימינה
0	2	ימינה
1	1	למטה-ימינה
1	-1	למטה-שמאלה
0	-2	שמאלה

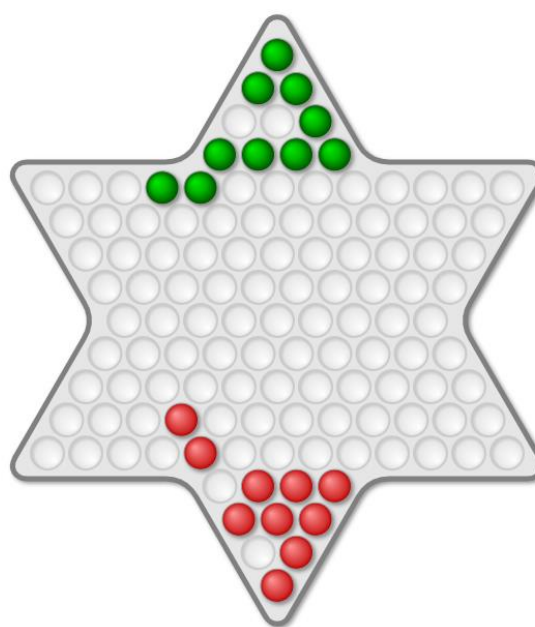
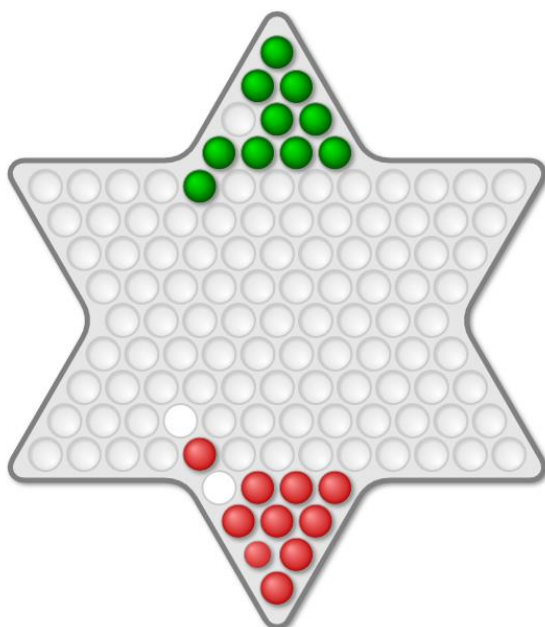
תרשים זרימת האלגוריתם:



צעד פשוט (אין חזרה רקורסיבית של האלגוריתם):



צעד מורכב (יש חזרה רקורסיבית של האלגוריתם):



פיתוח פתרון לבעיה אלגוריתמית מספר 3: אלגוריתם לשחקן ממוחשב (artificial intelligence)

בשביל לפתח AI לדמקה סינית החלטתי כמה דברים:

(1) אעשה AI רק עבור משחק בין שני שחקנים – שחקן אנושי נגד בוט או בוט נגד בוט. הצדדים שבחרתי הם ירוק (השחקן העליון) ואדום (השחקן התחתון), כי יותר קל לנתח את ניקוד המשחק שלהם מאשר שאר השחקנים, בגלל הייצוג של המשחק שלי במטריצה הדמיונית. בניגוד לשאר השחקנים, ניתן לזהות תזוזה נכונה שלהם בקלות רק על ידי השינוי במספר השורה של הכלים שלהם.

(2) קראתי והבנתי שיש פתיחות מקובלות למשחק. החלטתי לממש חלק מהפתיחות אשר מבטיחות התחלת משחק חזקה יותר ובכך משחק טוב יותר מאשר אם הייתי נותן לאלגוריתם כלשהו לבצע את פתיחת המשחק.

(3) החלטתי לממש שני אלגוריתמים שונים של AI עבור המשחק – Monte Carlo ו AlphaBeta, עליהם אפרט כעת:

אלגוריתם AlphaBeta:

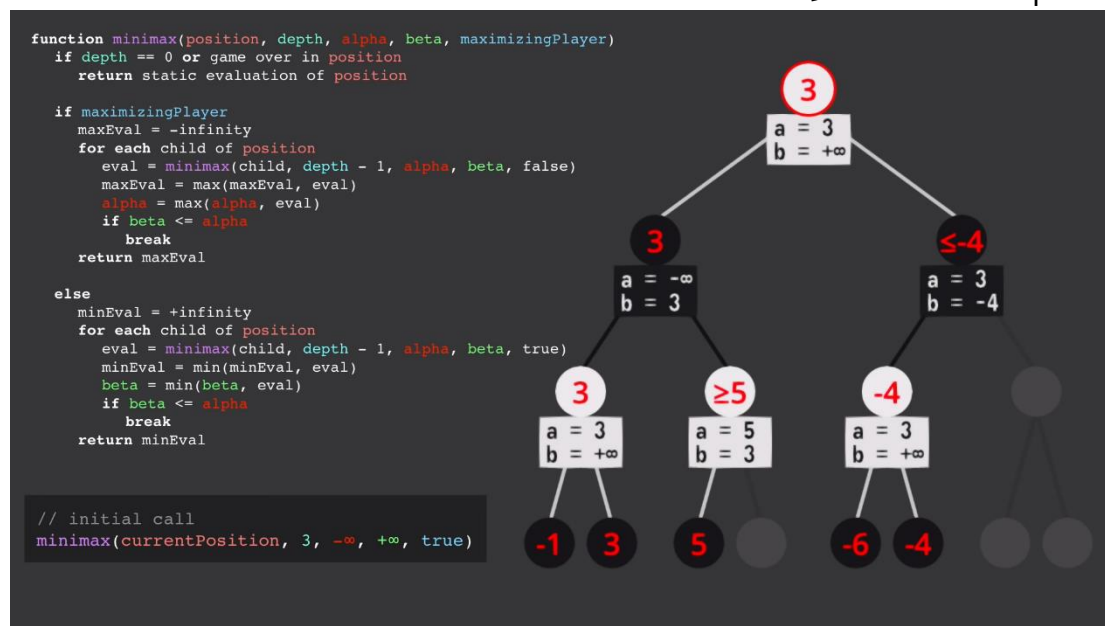
החלטתי להשתמש ב Alpha Beta כי זהו AI שמאוד מתאים לפתרון הבעיה. כעת אסביר על AlphaBeta אך לפני זה חשוב להבין מהו minimax כי AlphaBeta הוא בסך הכל שיפור של minimax.

עץ מינימקס הוא עץ הפורס את האפשרויות למשחק של שחקן א', את התגובות של שחקן ב' לכל פעולה של שחקן א', את תגובותיו של א' לתגובותיו של ב' וכך הלאה. מעשית מוגבל עומקו של העץ על ידי הזמן וזיכרון המחשב העומדים לרשותנו.

נסרוק את העץ החל בעלים, דרך הקודקודים (קודקוד הוא צומת פנימי בעץ) שמעליהם עד השורש (מלמטה למעלה) הציון שניתן לכל קודקוד הוא הערך הגבוה ביותר של העלים/הקודקודים שתחתיו, אם אותו קודקוד מסמל מהלך שלי, והערך הנמוך ביותר של העלים/קודקודים שתחתיו אם אותו קודקוד מסמל מהלך של היריב (כי ברור שהוא יבחר באפשרות הטובה ביותר בשבילו - הכי גרועה בשבילי).

כשנגיע לשורש, נבחר בקודקוד שמתחת לשורש שיש לו את הציון הטוב ביותר.

פסודו קוד ותרשים מפורט על אלפא בטא:



האלגוריתם AlphaBeta עובד כך:

אלפא מייצג את התוצאה הטובה ביותר שלנו ברמה זאת ומעלה כאשר מטרתנו למקסם את אלפא כמה שאפשר.

בטא מייצג את התוצאה הטובה ביותר של היריב ברמה זאת ומעלה כאשר מטרת היריב לצמצם את בטא כמה שאפשר.

הסיבה לכך היא שהניקוד מתייחס ללוח המשחק עבורנו (ניקוד גבוה יותר = מצב טוב יותר עבורנו).

בכל קריאה לפונקציה עבור השחקן אליו הפונקציה מתייחסת, נשנה את אלפא ובטא אם יש שינוי בתוצאה הטובה ביותר של השחקן.

אם באיזשהו שלב בטא קטן או שווה לאלפא, סימן שקיימת פעולה טובה יותר עבורנו ברמה למעלה ואין צורך להמשיך את החיפוש למטה.

כמה נקודות בנוגע למימוש האלגוריתם:

(1) למרות שAlphaBeta מייעל את minimax, עדיין במשחק שלי יש המון אפשרויות עבור כל תור, ולכן החלטתי לצמצם אפשרויות לא הגיוניות של מהלכים אפשריים של השחקנים כמו הבאת כלים אחורה ואפילו הזזת כלים לאותה השורה. נכון שבחלק מהמקרים ייתכן שזהו דווקא יהיה מהלך טוב אך ברוב המוחלט, בשחקן נגד שחקן זה כמעט ולא קורה. כך ניתן להגיע לעומקים גדולים יותר בשימוש הפונקציית AlphaBeta.

(2) שיטת הניקוד שלי תהיה מבוססת על כמה שיטות:

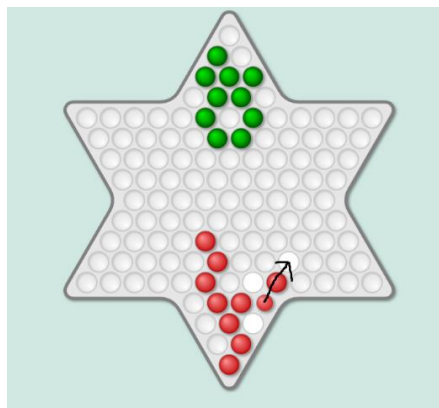
1. עבור כל מהלך לכיוון היריב, הוסף נקודה.

2. במידה ויש לשחקן כלי שאין אף כלי אחר לידו של אותו השחקן, הורד ניקוד לשחקן.

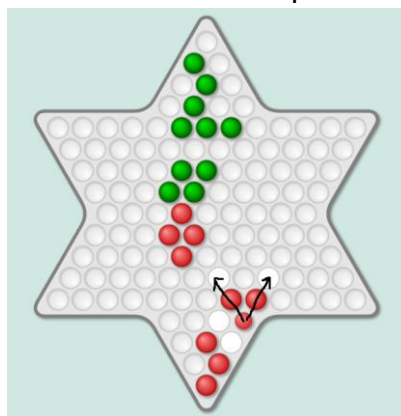
3. במידה והכלי נמצא בבית של השחקן, הורד לשחקן נקודה.

4. חפש צורות שיש לשחקן עם הכלים שלו:

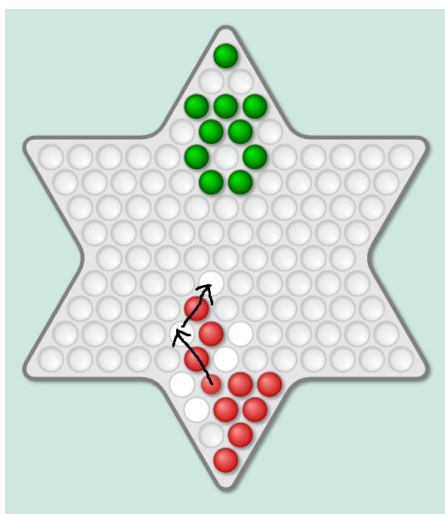
2.1 אם לשחקן יש רצף של שני כלים ואחריהם אין אף כלי, הוסף קצת ניקוד לשחקן.
דוגמה לרצף רגיל:



2.2 אם לשחקן יש שני רצפים של שני כלים הצמודים אחד לשני, הוסף ניקוד לשחקן.
דוגמה לרצף של שני כלים:



4.3 אם לשחקן יש רצף כפול של שני כלים המאפשר מעבר של 4 שורות ויותר במהלך אחד, הוסף הרבה ניקוד לשחקן. דוגמה לרצף כפול:



החלטתי ששיטה 4, שכן היא שיטה חזקה תהיה רק עבור AI שהוגדר Hard AlphaBeta.

(3) החלטתי לממש עבור AlphaBeta גם Iterative Deepening. המשמעות היא שעבור כל עומק שהאלגוריתם השלים, אנו ננסה לבצע את אותו האלגוריתם שוב עבור אותו עומק + 1, כל עוד לא

עבר הזמן שלנו לפעולה (חשוב לזכור שאי אפשר לתת לאלגוריתם לעבוד יותר מידי זמן כי המשחק לא יהיה כיף והשחקן ישתעמם).

אלגוריתם MonteCarlo:

העיקרון של אלגוריתם Monte Carlo הוא החזרת התוצאה ביותר לאחר משחקים רבים מהמצב הנוכחי. את התוצאה הטובה ביותר נקבל בהסתמך על הסתברות מתמטית של נוסחת UCT אותה שהיא:

$$\frac{w_i}{s_i} + c \sqrt{\frac{\ln s_p}{s_i}}$$

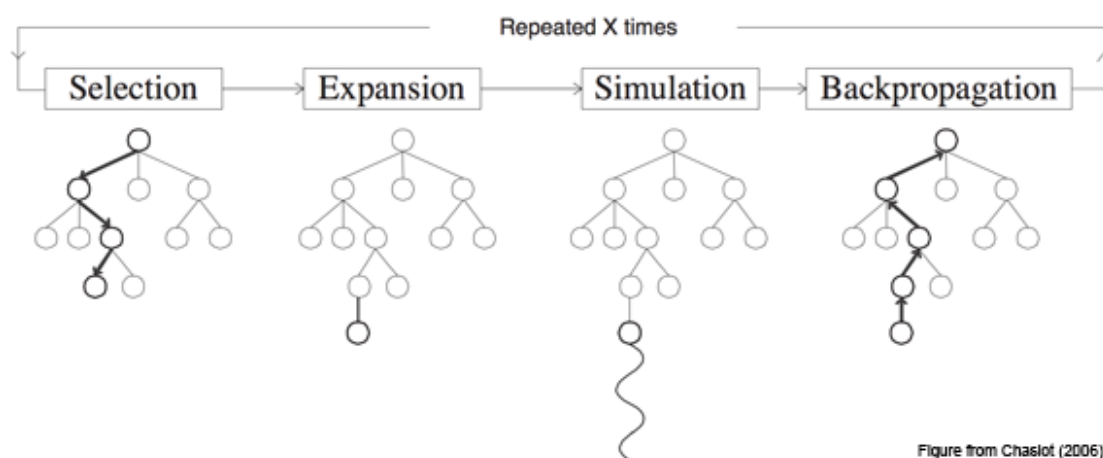
W_i = number of wins after the i -th move

S_i = number of simulations after the i -th move

c = exploration parameter (theoretically equal to $\sqrt{2}$)

S_p = total number of simulations for the parent node

הפונקציה מתחלקת ל-4 תתי אלגוריתמים המתרחשים אחד השני, והשאיפה היא שהם יקרו כמה שיותר כדי שנוכל לקבל תוצאה טובה יותר עבור המהלך הטוב ביותר:



1. בחירה - Selection: בשלב זה בוחרים את הצומת בעלת הניקוד הגבוה ביותר לפי המשוואה הנ"ל.

2. התרחבות - Expansion: לאחר שבחרנו את הצומת שנרצה להמשיך לחקור, נרחיב את צאצאיו לפי המהלכים האפשריים של השחקן.

3. סימולציה - Simulation: בשלב זה יוצרים סימולציות של משחקים רנדומליים מהמצב של הלוח שמייצג עד שנגמר הזמן שהוקצב לסימולציה או עד שהגענו למצב של ניצחון או הפסד עבור השחקן.

4. Backpropagation - לפי הערך שקיבלנו בסימולציה נעלה למעלה במעלי העץ ונעדכן כל צומת שבה עברנו בהתאם לתוצאות (כי התוצאות של האב תלוי בתוצאות של הבן).

2.4 תאור אלגוריתמים

תאור אלגוריתם מספר 1: תיאום מערכת לבחירות השחקן

קלט – סוג מצב שהשחקן רוצה להיכנס אליו.

פלט – השרת מחזיר ללקוח הודעה מתאימה עבור סוג הבקשה שלו.

בנה לובי חדש (סוג לובי) - $O(1)$:

1. אם סוג לובי = 'בוט לובי':

1.1 בניית לובי של בוט (שחקן ירוק, שחקן אדום)

2. אחרת אם סוג לובי = 'בוט אונליין'

2.1 בניית לובי אונליין (מחרוזת צבעים שמשתתפים במשחק)

3. אחרת אם סוג לובי = 'בוט אונליין'

3.1 בניית לובי לוקאלי (מחרוזת צבעים שמשתתפים במשחק)

בניית לובי של בוט (סוג שחקן ירוק, סוג שחקן אדום) - $O(1)$:

1. אם 'סוג שחקן ירוק' וגם 'סוג שחקן אדום' שווים ל'בן אדם' – בנה לובי לוקאלי (מחרוזת של ירוק ואדום)

2. אחרת

2.1 אתחול-לוח (כמות שחקנים = 2)

2.2. בנה סוג בוט (סוג שחקן ירוק)

2.3. בנה סוג בוט (סוג שחקן אדום)

בנה סוג בוט (סוג שחקן) - $O(1)$:

1. אם 'סוג שחקן' שונה מ'בן אדם':

1.1 אם אלפא-בטא – בנה אלפא בטא

1.2 אחרת אם אלפא-בטא-קשה – בנה אלפא בטא קשה

1.3 אחרת בנה מונטה קרלו

בניית לובי אונליין (מחרוזת צבעים שמשותפים במשחק) - $O(1)$:

1. **אתחל-לוח (כמות שחקנים = כאורך המחרוזת)**

2. אתחל מערך `Session[]` שיכיל את כל השחקנים במשחק

3. עבור כל צבע במחרוזת:

3.1 בנה שחקן חדש של צבע X

4. צרף את הלובי החדש לרשימת הלובים שהם אונליין וצריכים עוד שחקנים

בניית לובי לוקאלי (מחרוזת צבעים שמשותפים במשחק) - $O(1)$:

1. **אתחל-לוח (כמות שחקנים = כאורך המחרוזת)**

2. עבור כל צבע במחרוזת:

2.1 בנה שחקן חדש של צבע X

קבל את רשימת הלובים - $O(n)$ (כאשר n מספר הלובים שהם אונליין המחכים לעוד שחקנים):

1. **החזר את כל הלובים שהם אונליין המחכים לעוד שחקנים.**

הוסף שחקן ללובי אונליין קיים - $O(1)$:

1. תשיג את הלובי מרשימת הלובים שהם אונליין ובהמתנה למשחק (ב $O(1)$ כי אנו שומרים את הרשימה כ`HashSet`).

2. בדוק האם הלובי אינו `NULL` (במידה ומישהו הצטרף/לא נשארו עוד שחקנים בלובי לפני שהמשתמש ביקש מהסרבר בקשה להצטרפות ללובי).

2.1 הוסף את המשתמש ללובי.

2.1.1 בדוק האם הלובי מלא כעת ויכל להתחיל משחק

2.1.1.1 שלח הודעה על כך לכל שאר השחקנים

תאור אלגוריתם מספר 2: הצגת כל אפשרויות התזוזה עבור כלי שנבחר - $O(n)$ - כאשר n מייצג את כמות האפשרויות עבור כל חייל שנבחר

האלגוריתם מקבל כקלט:

- נקודת משבצת המוצא של הכלי

פלט האלגוריתם:

- מחזיר את כל המהלכים החוקיים הבאים עבור הכלי שקיבלנו בקלט

עבור על ששת הצדדים (מיקום, האם נכנס מתוך האלגוריתם עצמו):

1. אתחל רשימת מהלכים אפשריים ריקה.

2. עבור על כל ששת הכיוונים במטריצת הכיוונים

2.1. עבור על אפשרויות של צד אחד(מיקום, היסט כיוון, האם נכנס מתוך האלגוריתם עצמו)

עבור על אפשרויות של צד אחד(מיקום, היסט כיוון, האם נכנס מתוך האלגוריתם עצמו):

1. בנה משבצת חדשה עם המיקום ההתחלתי והיסט הכיוון.

2. האם המשבצת קיימת במסיכת לוח המשחק.

2.1. האם המשבצת החדשה תפוסה.

2.1.1 בדוק האם המשבצת השנייה באותו הכיוון קיימת וגם פנוייה.

2.1.1.1 בדוק האם המשבצת השנייה לא נמצאת ברשימת המהלכים.

2.1.1.1.1 הוסף את המשבצת השנייה לרשימת המהלכים.

2.1.1.1.2 עבור על ששת הצדדים (המשבצת השנייה, אמת).

2.2. האם נכנס מתוך האלגוריתם עצמו = שקר.

2.2.1 הוסף את המשבצת השנייה לרשימת המהלכים.

תאור אלגוריתם מספר 3: אלגוריתם לשחקן ממוחשב (artificial intelligence) - Iterative Deepening AlphaBeta

ישנם שני AI במשחק והם MonteCarlo ו AlphaBeta:

:AlphaBeta

יעילות האלגוריתם: אנו מחליטים על משך הזמן של האלגוריתם, כי אם האלפא-בטא חורג מהזמן המוקצב לו, הפעילות שלו תיפסק ונחזיר את התוצאה הטובה ביותר שנמצאה עד לפני שהופסקה פעולת החיפוש. לכן היעילות היא $O(1)$.

האלגוריתם מתחלק לשני תתי-אלגוריתמים:

אלגוריתם: Iterative Deepening

1. בצע אלפא-בטא (עומק ראשוני, שחקן נוכחי, מינוס אין סוף, פלוס אין סוף) ושמור אותו במהלך הבטוח והכי טוב.
2. כל עוד נשאר זמן למהלך בצע אלפא-בטא (עומק + 1, מינוס אין סוף, פלוס אין סוף) ושמור אותו במהלך הכי טוב.
3. אם 2 הושלם במלואו שמור את המהלך במהלך הבטוח והכי טוב וחזור על 2.
4. אחרת החזר מהלך הבטוח והכי טוב.

פלט התת-אלגוריתם:

- מחזיר את המהלך האופטימלי לתור הנוכחי עבור השחקן הממוחשב בזמן שהוקצב.

אלגוריתם: אלפא-בטא (עומק, שחקן, אלפא, בטא)

1. אם הזמן עבר

1.1 החזר את התוצאה הטובה ביותר שנמצאה עד כה.

2. אם הגענו לעומק 0

2.1 בצע ניקוד לוח משחק נוכחי

3. אם שחקן = 'שחקן נוכחי'

3.1 השג רשימה של כל המהלכים ההגיוניים עבור השחקן הנוכחי.

3.2 עבור על כל אחד מהמהלכים שברשימה.

3.2.1 בצע את המהלך ועדכן את הלוח.

3.2.2 בצע אלפא-בטא (עומק - 1, 'שחקן יריב', אלפא, בטא) ושמור את התוצאה בניקוד מהלך.

3.2.3 החזר את מצב הלוח לפני עשיית המהלך.

3.2.4 אם ניקוד המהלך גדול מאלפא.

3.2.4.1. עדכן את אלפא כך שיהיה שווה לניקוד המהלך.

3.2.4.2 אם מדובר בעומק ש**Iterative Deepening** זימן ממנו את
אלפא-בטא

3.2.4.2.1. שמור את המהלך במהלך הכי טוב.

3.2.5. אם אלפא גדול שווה לבטא

3.2.5.1. החזר את אלפא.

3.2.1. החזר את אלפא.

4. אחרת

4.1. השג רשימה של כל המהלכים ההגיוניים עבור השחקן היריב.

4.2. עבור על כל אחד מהמלכים שברשימה.

4.2.1. בצע את המהלך ועדכן את הלוח.

4.2.2. בצע **אלפא-בטא** (עומק - 1, 'שחקן יריב', אלפא, בטא) ושמור את התוצאה
בניקוד המהלך.

4.2.3. החזר את מצב הלוח לפני עשיית המהלך.

4.2.4. אם ניקוד המהלך קטן מבטא.

4.2.4.1. עדכן את בטא כך שיהיה שווה לניקוד המהלך.

4.2.5. אם אלפא גדול שווה לבטא

4.2.5.1. החזר את בטא.

4.3. החזר את בטא.

פלט התת-אלגוריתם:

- מחזיר את המהלך האופטימלי עבור העומק הרצוי.

:Monte Carlo

יעילות האלגוריתם: למטה ארשום את היעילות של כל אחד מהאלגוריתמים אך למעשה היעילות של האלגוריתם כולו היא $O(1)$ כי אנו קובעים עד מתי האלגוריתם נמשך עבור רצף הפעולות שלו, וכאשר עובר הזמן אנו נחזיר את התוצאה הטובה ביותר שמצאנו.

האלגוריתם מתחלק ל-4 תתי-אלגוריתמים:

אלגוריתם: בחירה (root) – $O(n)$ כאשר n עומק העץ

1. $childNode = root$

2. כל עוד מערך הילדים של $childNode$ הנוכחי לא ריק:

2.1. בחר את הילד הטוב ביותר באמצעות UCT ושומר אותו ב- $childNode$.

3. החזר את $childNode$.

אלגוריתם: הרחבה (node) – $O(n)$ כאשר n מספר המהלכים האפשריים ממצב הלוח של $node$

1. צור רשימה של כל המהלכים האפשריים עבור $node$ שהתקבל.

2. עבור כל מהלך ברשימת המהלכים:

2.1. צור $newNode$ עם לוח המשחק ומהלך החדש

2.2. הוסף אותו לרשימת הילדים של $node$.

אלגוריתם: סימולציה (node) – $O(1)$ כי אנו קובעים את כמות הזמן שתתמשך הסימולציה:

1. כל עוד לא נגמר הזמן וגם אף אחד לא ניצח, בצע:

1.1. בצע מהלך רנדומלי.

1.2. החלף לשחקן השני.

2. אם השחקן שלנו ניצח החזר 1

3. אם השחקן היריב ניצח החזר -1.

4. החזר 0.

אלגוריתם: **(leaf)Backpropagation** – $O(n)$ כאשר n עומק העץ:

1. `parent = leaf.getParent()`

2. כל עוד `parent` שונה מ-`null` בצע:

2.1 העלה ב-1 את מספר הביקורים ב-`parent`.

2.2 הוסף את תוצאת המשחק ב-`parent` ל-`gameScore`.

2.3 `parent = leaf.getParent()`

2.5 מבנה נתונים

מבנה הנתונים שלי בפרויקט הוא מחלקת Board. ניתן לחלק את המחלקה לשני חלקים:

1. לשמור את כלי המשחק של השחקנים. לא את המשבצות הריקות, אלא רק כלים ממשיים של שחקנים. לצורך כך פתחתי גם את מחלקות Player, Marble, Location – עליהן אפרט בהמשך.
2. לייצג את לוח המשחק שבצורת 'מגן דוד' על ידי "מסכה" – אותה הכנתי מביטים שתהווה מעין מטריצה דמיונית של גומות המשחק. במסיכה, כל ביט דלוק מייצג תא קייים בלוח המשחק, וכל ביט כבוי מייצג תא שאינו קיים בלוח המשחק. כך אני יכל לעבור עבור מיקום נתון על כל אחד מששת הצדדים שלצידו, ולבדוק האם כל אחד מהם באמת קיים על לוח המשחק או לא.

כלומר מטרת המחלקה היא לשמור רק כלי משחק ממשיים ובכך לחסוך זיכרון מקום (כי אנו שומרים רק את הכלים הקיימים במשחק).

בנוסף במחלקה יש מטריצת כיוונים, שמכילה את ההיסט עבור כל כיוון.

מחלקת Board:

```
public class Board {
    // =====
    // Consts Board details:
    // =====
    // The CELLS_COUNT represents the ammount of bits in the boardMask.
    // The last 9 bits are 0 so I didn't wrote them to the board mask, so we
    // have to sub them from the CELLS_COUNT.
    private final static short CELLS_COUNT = 425-9;
    // There are 17 rows in the board
    public final static byte ROWS = 17;
    // There are 25 cols in the board
    public final static byte COLS = 25;

    // The boardMask the represents by bits the real cells in the Board.
    private final static byte[] boardMask = {
        0x00, 0x08, 0x00, 0x00,
        0x0A, 0x00, 0x00, 0x0A,
        (byte) 0x80, 0x00, 0x0A, (byte) 0xA0,
        0x0A, (byte) 0xAA, (byte) 0xAA, (byte) 0xAA,
        (byte) 0xAA, (byte) 0xAA, (byte) 0xA8, (byte) 0xAA,
        (byte) 0xAA, (byte) 0xA8, 0x2A, (byte) 0xAA,
        (byte) 0xA8, 0x0A, (byte) 0xAA, (byte) 0xA8,
        0x0A, (byte) 0xAA, (byte) 0xAA, 0x0A,
        (byte) 0xAA, (byte) 0xAA, (byte) 0x8A, (byte) 0xAA,
        (byte) 0xAA, (byte) 0xAA, (byte) 0xAA, (byte) 0xAA,
        (byte) 0xA8, 0x02, (byte) 0xA8, 0x00,
        0x00, (byte) 0xA8, 0x00, 0x00,
        0x28, 0x00, 0x00, 0x08};

    // The diraction array.
    public final static byte[][] dirArr = {
        {-1, -1}, // Top-Left
        {-1, 1}, // Top-Right
        {0, 2}, // Right
        {1, 1}, // Bottom-Right
        {1, -1}, // Bottom-Left
        {0, -2} // Left
    };

    // The players array
    private final Player[] playersArr;
```

בדיקה האם מיקום נמצא על גומה חוקית בלוח:

```
/**
 * The function returns true if given location exists in the game board,
 * otherwise returns false.
 * @param loc the given location
 * @return true if given location exists in the game board,
 *         otherwise returns false.
 */
public static boolean isCellExists(Location loc) {
    byte row = loc.getRow();
    byte col = loc.getCol();

    if (row < 0 || col < 0)
        return false;
    if (col >= COLS || row >= ROWS)
        return false;
    if (row*COLS + col > CELLS_COUNT)
        return false;
    return (((boardMask[(row*COLS + col) / 8]) << (row*COLS + col) % 8) & 0x80) != 0);
}
```

Player – כל אובייקט של שחקן מכיל מערך (בגודל 10) של Marble, את המיקומים ההתחלתיים של הכלים שלו ואת הצבע שלו.

מחלקת Player:

```
public class Player {
    // currMarblesPos have the marbles of the player
    private final Marble[] currMarblesPos;
    // startingMarblesPos is const and contains the starting locations of the
    // player's marbles
    private final Location[] startingMarblesPos;
    // The color of the player
    private final char color;
}
```

Marble – מכיל אובייקט Location, ושני משתנים בוליאניים – האחד האם הכלי נמצא בביתו, והשני האם הכלי נמצא בבסיס של יריבו הנגדי.

מחלקת Marble:

```
public class Marble {
    // The location of the Marble (contains row and col)
    private final Location loc;
    // Flag to indicate if this marble is on the starting spot of the player
    // it belongs to
    private boolean isOnStartingSpot;
    // Flag to indicate if this marble is on the winning spot of the player
    // it belongs to
    private boolean isOnWinningSpot;
}
```

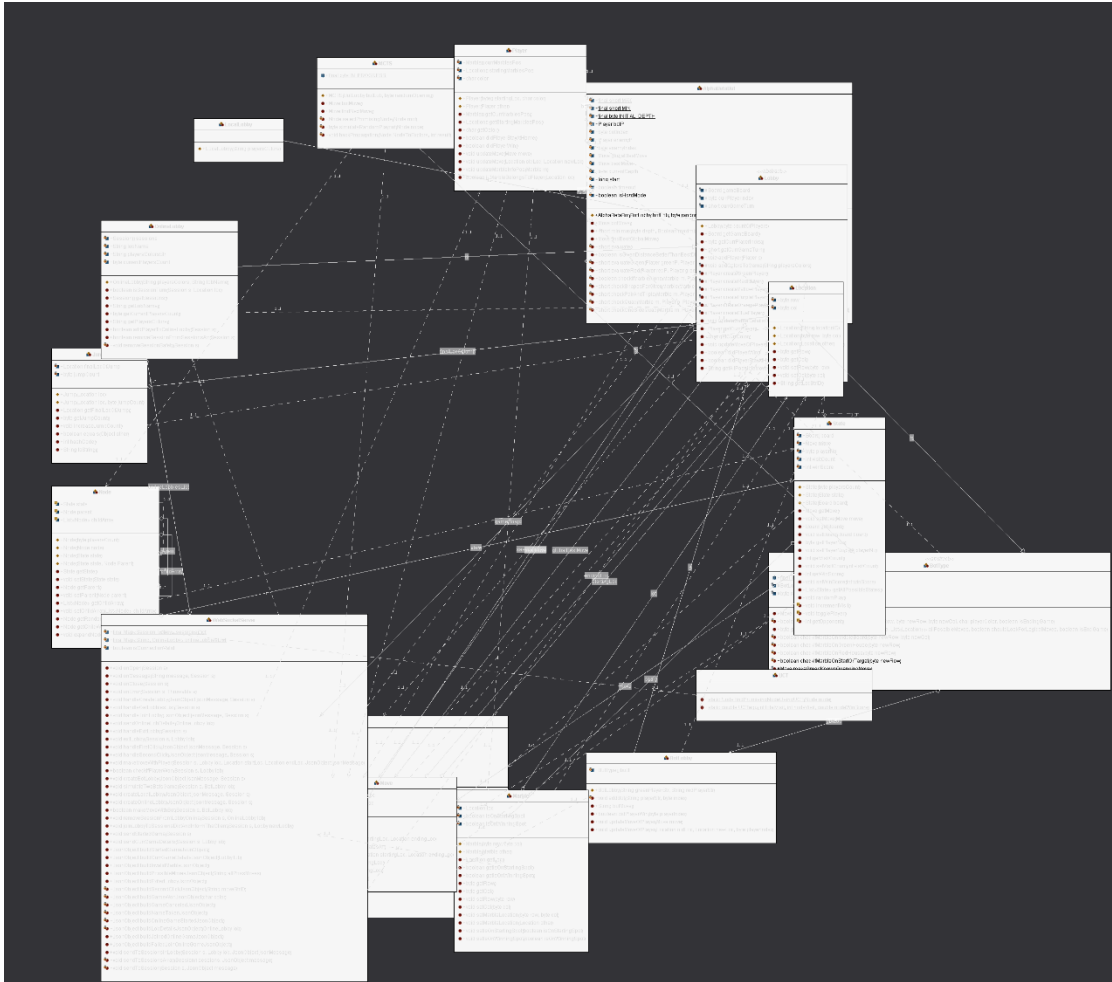
Location – מכילה row ו- col ומייצגת מיקום על הלוח.

מחלקת Location:

```
public class Location {
    private byte row;
    private byte col;
}
```

3. תכנון

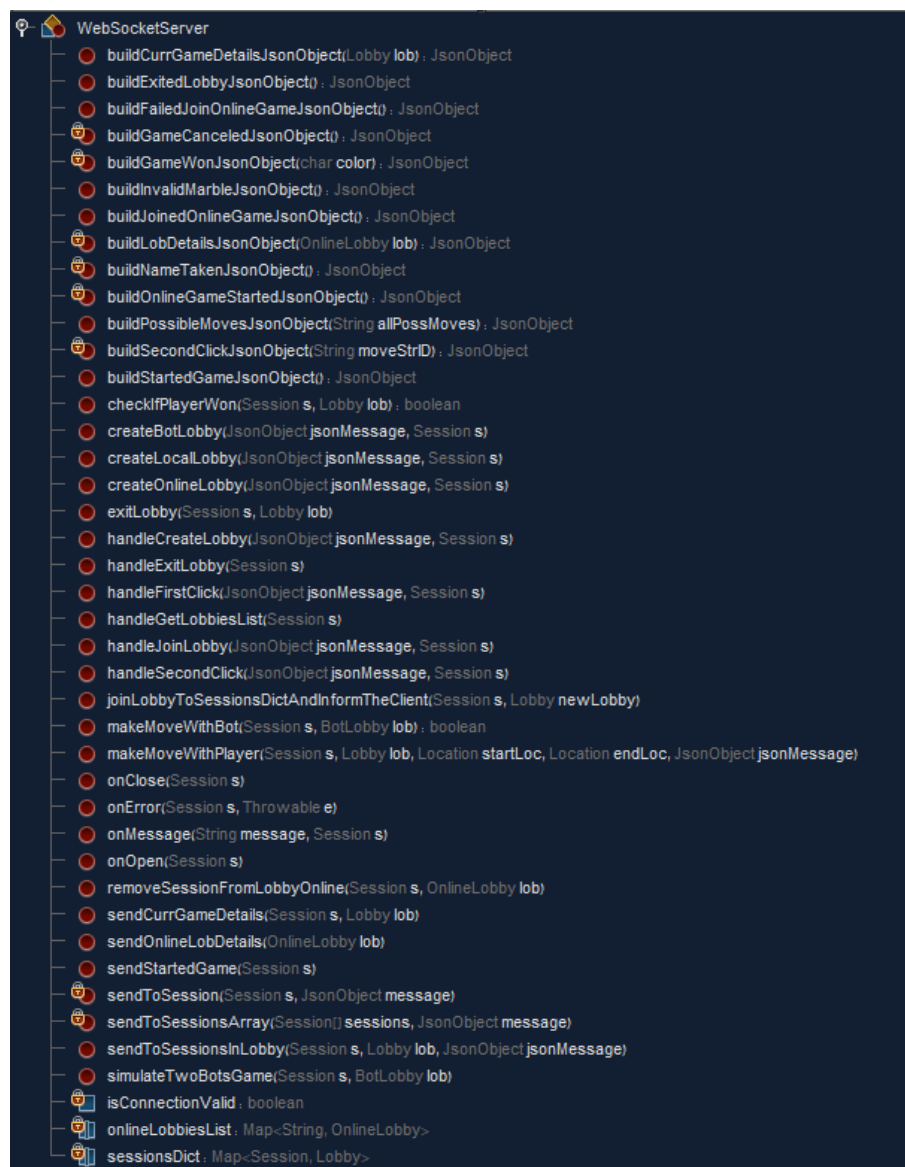
תרשים UML:



3.1 חלוקה למודולים

מחלקת `WebSocketServer`:

מחלקה זו היא בעצם `Server`, ואחראית על כל התקשורת בין השחקנים השונים בינם לבין הסרבר ולבין שחקנים אחרים אם נדרש.



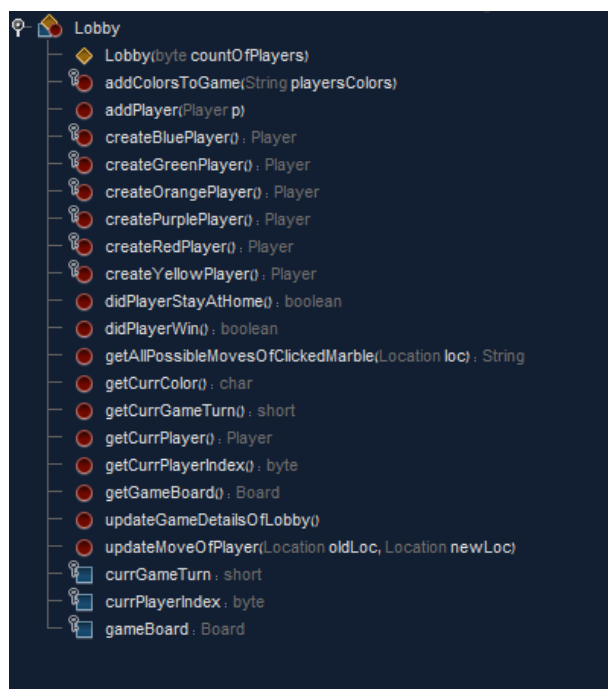
שם הפעולה	תיעוד הפעולה
<code>@OnOpen</code> <code>public void onOpen(Session s)</code>	<p>הפעולה מקבלת את כל הכניסות השונות לסרבר, ומטפלת בהם על ידי הכנסת <code>Session</code> החדש ל-<code>sessionsDict</code>.</p>

הפעולה אחראית על כל טיפול הבקשות של הקליינטים השונים לסרברים, על ידי העברת כל בקשה לפונקציה המתאימה לה.	<code>@OnMessage</code> <code>public void onMessage(String message, Session s)</code>
הפעולה סוגרת Session בצורה בטוחה ומעדכנת את sessionsDictn.	<code>@OnClose</code> <code>public void onClose(Session s)</code>
הפעולה מדפיסה את stackTrace עבור שגיאה שהתרחשה, ומעדכנת את false! isConnectedValidn.	<code>@OnError</code> <code>public void onError(Session s, Throwable e)</code>
הפעולה יוצרת לובי חדש עבור קליינט מסויים לפי הבקשה שלו.	<code>public void</code> <code>handleCreateLobby(JsonObject jsonMessage, Session s)</code>
הפעולה מחזירה לקליינט את כל רשימת הלובים שהם אונליין שניתן להצטרף אליהם.	<code>public void</code> <code>handleGetLobbiesList(Session s)</code>
פעולה זו מכניסה לקוח מסויים ללובי שהוא ביקש. במידה ולא התאפשר הפעולה תחזיר לקליינט הודעת שגיאה מתאימה על כך.	<code>public void</code> <code>handleJoinLobby(JsonObject jsonMessage, Session s)</code>
הפעולה שולחת את הפרטים (צבע שחקן נוכחי ומספר תור) של אונליין לובי לכל הקליינטים בלובי.	<code>public void</code> <code>sendOnlineLobDetails(OnlineLobby lob)</code>
הפעולה מטפלת בבקשה של קליינט ליציאה מהלובי הנוכחי בו הוא נמצא.	<code>public void</code> <code>handleExitLobby(Session s)</code>
הפעולה מוציאה קליינט שהתקבל מלובי שהתקבל.	<code>public void</code> <code>exitLobby(Session s, Lobby lob)</code>
הפעולה מטפלת בלחיצה ראשונה של השחקן על הלוח. במידה ולא התאפשר לשחקן ללחוץ על הכלי (למשל כי הכלי לא שלו), הפונקציה תחזיר לו על כך הודעה מתאימה. במידה והכל התרחש כצפוי, תישלח ללקוח הודעת json ובה פרטים על כל המהלכים האפשריים עבור כלי המשחק שהתקבל.	<code>public void</code> <code>handleFirstClick(JsonObject jsonMessage, Session s)</code>
הפעולה מטפלת בלחיצה שנייה של השחקן על הלוח, כלומר במהלך מלא. הפונקציה תעדכן את המהלך בסרבר תשלח הודעה מתאימה לכל השחקנים בלובי כדי שידעו כיצד לעדכן את לוח המשחק שלהם.	<code>public void</code> <code>handleSecondClick(JsonObject jsonMessage, Session s)</code>
פעולה מבצעת מהלך של שחקן מסויים בלובי מסויים ומעדכנת את כל שאר השחקנים בלובי על כך.	<code>public void</code> <code>makeMoveWithPlayer(Session s, Lobby lob, Location startLoc, Location endLoc, JsonObject jsonMessage)</code>

הפעולה בודקת האם שחקן נוכחי בלובי ניצח. אם כן הפעולה תעדכן את כל שאר השחקנים בלובי על כך.	<code>public boolean checkIfPlayerWon(Session s, Lobby lob)</code>
הפעולה יוצרת בוט לובי חדש. במידה ומדובר בשני בוטים הפעולה תזמן את הפעולה <code>simulateTwoBotsGame</code> .	<code>public void createBotLobby(JsonObject jsonMessage, Session s)</code>
הפעולה מדמה משחק שלם בין שני בוטים ומעדכנת את הקליינט בכל פעולה שלהם כך שיוכל לצפות במשחק עד לניצחון של אחד מהם.	<code>public void simulateTwoBotsGame(Session s, BotLobby lob)</code>
הפעולה יוצרת לוקאל לובי חדש עבור ה- <code>Session</code> שהתקבל.	<code>public void createLocalLobby(JsonObject jsonMessage, Session s)</code>
הפעולה יוצרת אונליין לובי חדש עבור ה- <code>Session</code> שהתקבל.	<code>public void createOnlineLobby(JsonObject jsonMessage, Session s)</code>
הפעולה מבצעת צעד עם בוט מסויים בבוט לובי ומעדכנת את ה- <code>Session</code> על מצב המשחק.	<code>public boolean makeMoveWithBot(Session s, BotLobby lob)</code>
הפעולה מוציאה <code>Session</code> מסויים מ- <code>OnlineLobby</code> .	<code>public void removeSessionFromLobbyOnline(Session s, OnlineLobby lob)</code>
הפעולה מכניסה <code>Session</code> שהתקבל לתוך <code>Lobby</code> שהתקבל, ושולחת הודעת לקליינט שהמשחק התחיל ולאחר מכן את פרטי המשחק.	<code>public void joinLobbyToSessionsDictAnd InformTheClient(Session s, Lobby newLobby)</code>
הפעולה שולחת ל- <code>Session</code> שהתקבל שהמשחק התחיל.	<code>public void sendStartedGame(Session s)</code>
הפעולה שולחת ל- <code>Session</code> את פרטי <code>Lobby</code> הנוכחיים (צבע שחקן נוכחי, ומספר התור הנוכחי של המשחק).	<code>public void sendCurrGameDetails(Session s, Lobby lob)</code>
הפעולה שולחת לכל השחקנים בלובי הודעת <code>json</code> מסויימת.	<code>public void sendToSessionsInLobby(Session s, Lobby lob, JsonObject jsonMessage)</code>
הפעולה שולחת לכל השחקנים במערך <code>Session</code> הודעת <code>json</code> מסויימת.	<code>private void sendToSessionsArray(Session[] sessions, JsonObject message)</code>
הפעולה שולחת ל- <code>Session</code> מסויים הודעת <code>json</code> מסויימת.	<code>private void sendToSession(Session s, JsonObject message)</code>

מחלקת Lobby – מחלקה אבסטרקטית:

מחלקה זו מהווה בסיס לשאר סוגי הלובי הממשיים – BotLobby, LocalLobby, OnlineLobby.

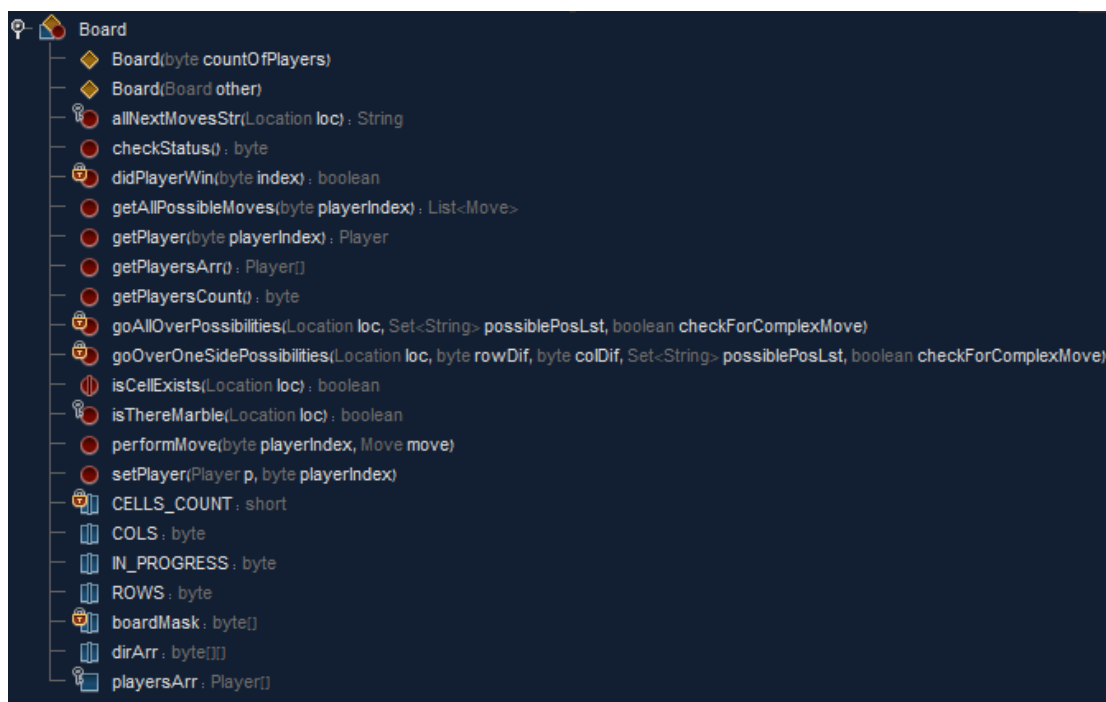


שם הפעולה	תיעוד הפעולה
<code>public Lobby (byte countOfPlayers)</code>	פעולה בונה המקבלת כמות שחקנים מסויימת ויוצרת לובי חדש.
<code>public Board getGameBoard()</code>	הפעולה מחזירה את לוח המשחק של הלובי.
<code>public byte getCurrPlayerIndex()</code>	הפעולה מחזירה את האינדקס של השחקן הנוכחי.
<code>public short getCurrGameTurn()</code>	הפעולה מחזירה את התור הנוכחי.
<code>public void addPlayer(Player p)</code>	הפעולה מוסיפה שחקן מסויים ללובי ומעדכנת את פרטי הלובי.
<code>protected void addColorsToGame(String playersColors)</code>	הפעולה מוסיפה שחקנים חדשים ללוח המשחק לפי מחרוזת המכילה את הצבעים שיש להוסיף ללוח משחק.
<code>protected Player createGreenPlayer()</code>	הפעולה יוצרת שחקן ירוק חדש ומחזירה אותו.
<code>protected Player createRedPlayer()</code>	הפעולה יוצרת שחקן אדום חדש ומחזירה אותו.

הפעולה יוצרת שחקן צהוב חדש ומחזירה אותו.	<code>protected Player createYellowPlayer()</code>
הפעולה יוצרת שחקן סגול חדש ומחזירה אותו.	<code>protected Player createPurplePlayer()</code>
הפעולה יוצרת שחקן כתום חדש ומחזירה אותו.	<code>protected Player createOrangePlayer()</code>
הפעולה יוצרת שחקן כחול חדש ומחזירה אותו.	<code>protected Player createBluePlayer()</code>
הפעולה מעדכנת את ה- <code>currentIndex</code> ואת ה- <code>currentGameTurn</code> (אם יש צורך).	<code>public void updateGameDetailsOfLobby()</code>
הפעולה מחזירה את השחקן הנוכחי.	<code>public Player getCurrPlayer()</code>
הפעולה מחזירה את הצבע של השחקן הנוכחי.	<code>public char getCurrColor()</code>
הפעולה מחזירה אמת אם השחקן הנוכחי ניצח, שקר אחרת.	<code>public boolean didPlayerWin()</code>
הפעולה מחזירה אמת אם יש כלי לשחקן הנוכחי שנשאר בביתו, אחרת מחזירה שקר.	<code>public boolean didPlayerStayAtHome()</code>
הפעולה מחזירה כמחרוזת את כל המהלכים הבאים עבור מיקום של תא מסויים. הפעולה משתמשת ב- <code>allNextMovesStr</code> של מחלקת <code>Board</code> .	<code>public String getAllPossibleMovesOfClickedMarble(Location loc)</code>

מחלקת Board:

מחלקה זו מייצגת את לוח המשחק.

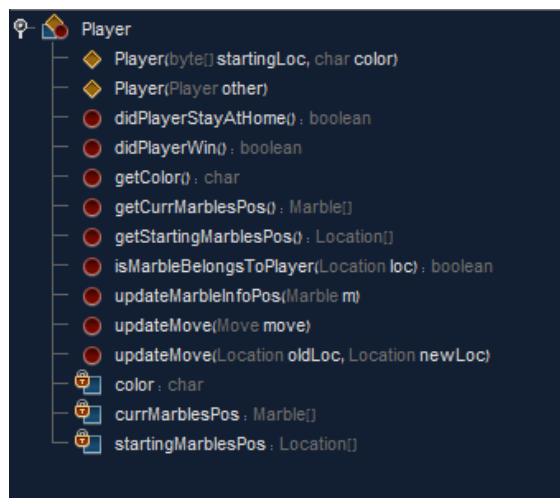


שם הפעולה	תיעוד הפעולה
public Board(byte countOfPlayers)	פעולה בונה, המקבלת כמות שחקנים ומאתחלת את playersArr בגודל כמות השחקנים שהתקבלה.
public Board(Board other)	פעולה בונה מעתיקה.
public Player getPlayer(byte playerIndex)	פעולה מחזירה את השחקן במקום האינדקס שהתקבל.
public void setPlayer(Player p, byte playerIndex)	פעולה מאתחלת את השחקן במקום האינדקס שהתקבל.
public byte getPlayersCount()	פעולה מחזיר את כמות השחקנים על לוח המשחקים (את אורך המערך playersArr).
public Player[] getPlayersArr()	פעולה מחזירה את מערך השחקנים.
public void performMove(byte playerIndex, Move move)	פעולה מקבלת אינדקס של שחקן, מהלך מסוים ומבצעת אותו.
public byte checkStatus()	פעולה מחזירה את האינדקס של השחקן שניצח. אם אף שחקן לא ניצח, הפעולה תחזיר את הערך של IN_PROGRESS.

פעולה מחזירה אמת אם שחקן שנמצא באינדקס שהתקבל בתוך מערך השחקנים ניצח, אחרת מחזירה שקר.	<code>private boolean didPlayerWin(byte index)</code>
פעולה מחזירה את רשימת כל המהלכים האפשריים עבור שחקן מסויים.	<code>public List<Move> getAllPossibleMoves(byte playerIndex)</code>
פעולה מחזירה מחרוזת עם כל המהלכים האפשריים עבור מיקום שהתקבל.	<code>protected String allNextMovesStr(Location loc)</code>
פעולה מעדכנת את <code>possiblePosLst</code> כך שיכיל את כל המהלכים החוקיים עבור <code>loc</code> שהתקבל	<code>private void goAllOverPossibilities(Location loc, Set<String> possiblePosLst, boolean checkForComplexMove)</code>
פעולה מעדכנת את <code>possiblePosLst</code> כך שיכיל עוד מהלכים אפשריים עבור צד אחד שהתקבל שהיסטו הוא <code>colDif</code> ו <code>rowDif</code>	<code>private void goOverOneSidePossibilities(Location loc, byte rowDif, byte colDif, Set<String> possiblePosLst, boolean checkForComplexMove)</code>
פעולה סטטית שתפקידה להחזיר אמת אם המיקום שהתקבל הינו גומה תקנית על הלוח, שקר אחרת.	<code>public static boolean isCellExists(Location loc)</code>
פעולה מחזירה אמת אם מיקום שהתקבל שייך לאחד השחקנים שבמשחק, אחרת מחזירה שקר.	<code>protected boolean isThereMarble(Location loc)</code>

מחלקת Player:

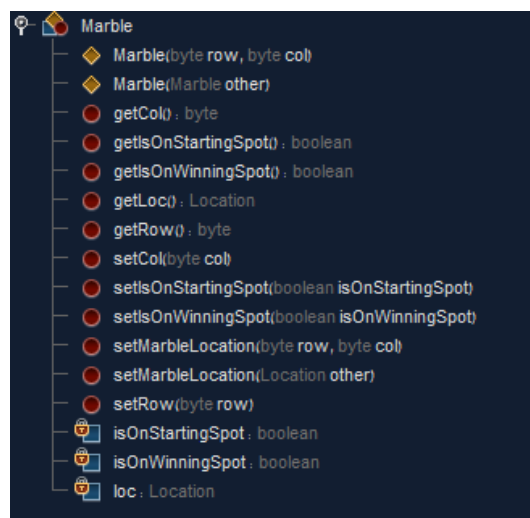
מחלקה זו מייצגת אובייקט שחקן.



שם הפעולה	תיעוד הפעולה
public Player(byte[] startingLoc, char color)	בנאי מחלקת Player, הבנאי מקבל את מיקום הכלים ההתחלתיים שלו ואת צבעו.
public Player(Player other)	פעולה בונה מעתיקה.
public Marble[] getCurrMarblesPos()	הפעולה מחזירה את מערך הכלים.
public Location[] getStartingMarblesPos()	הפעולה מחזירה את מערך המיקומים ההתחלתי של השחקן.
public char getColor()	הפעולה מחזירה את צבע השחקן.
public boolean didPlayerStayAtHome()	הפעולה מחזירה אמת אם לשחקן יש כלי בביתו, אחרת מחזירה שקר.
public boolean didPlayerWin()	הפעולה מחזירה אמת אם השחקן ניצח אחרת מחזירה שקר.
public void updateMove(Move move)	הפעולה מעדכנת את מערך הכלים של השחקן לפי ה Move שהתקבל.
public void updateMove(Location oldLoc, Location newLoc)	הפעולה מעדכנת את מערך הכלים של השחקן לפי המיקומים שהתקבלו.
public void updateMarbleInfoPos(Marble m)	הפעולה מעדכנת את isOnStartingSpot ו isOnWinningSpot של Marble שהתקבל
public boolean isMarbleBelongsToPlayer(Location loc)	הפעולה מחזירה אמת אם ישנו כלי במיקום שהתקבל אחרת מחזירה שקר.

מחלקת Marble:

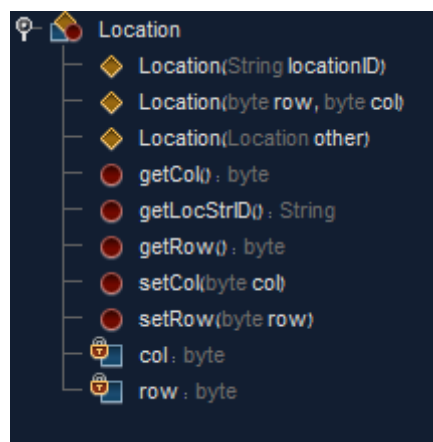
מחלקה זו מייצגת כלי משחק של אחד השחקנים.



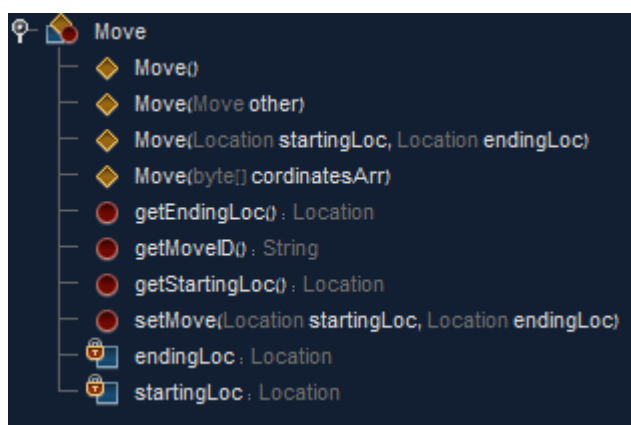
שם הפעולה	תיעוד הפעולה
public Marble(byte row, byte col)	בנאי מחלקת Marble, הבנאי מקבל שורה ועמודה ובונה Marble חדש.
public Marble(Marble other)	פעולה בונה מעתיקה.
public Location getLoc()	הפעולה מחזירה מיקום הכלי.
public boolean getIsOnStartingSpot()	הפעולה מחזירה האם הכלי בביתו של השחקן שלו.
public boolean getIsOnWinningSpot()	הפעולה מחזירה האם הכלי בבסיס של היריב הנגדי של השחקן שלו.
public byte getRow()	הפעולה מחזירה את השורה של הכלי.
public byte getCol()	הפעולה מחזירה את העמודה של הכלי.
public void setRow(byte row)	הפעולה מעדכנת את השורה של הכלי.
public void setCol(byte col)	הפעולה מעדכנת את העמודה של הכלי.
public void setMarbleLocation(byte row, byte col)	הפעולה מעדכנת את הLocation של הכלי.
public void setMarbleLocation(Location other)	הפעולה מעדכנת את הLocation של הכלי.
public void setIsOnStartingSpot(boolean isOnStartingSpot)	הפעולה מעדכנת את ה- isOnStartingSpot של הכלי.
public void setIsOnWinningSpot(boolean isOnWinningSpot)	הפעולה מעדכנת את ה- isOnWinningSpot של הכלי.

מחלקת Location:

מחלקה המייצגת מיקום – בעלת row ו col ופעולות get ו set בסיסיות.



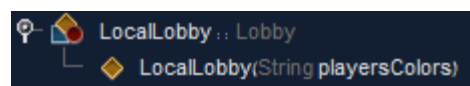
שם הפעולה	תיעוד הפעולה
public Location(String locationID)	בנאי מחלקת Location. הפעולה מקבלת locationID, כלומר מחרוזת שידוע שהיא מייצגת מיקום ובונה ממנו Location חדש.
public Location(byte row, byte col)	בנאי מחלקת Location. הפעולה מקבלת row ו col ובונה מהם Location חדש.
public Location(Location other)	פעולה בונה מעתיקה.
public byte getRow()	הפעולה מחזירה את השורה של המיקום.
public byte getCol()	הפעולה מחזירה את העמודה של המיקום.
public void setRow(byte row)	הפעולה מעדכנת את השורה של המיקום.
public void setCol(byte col)	הפעולה מעדכנת את העמודה של המיקום.
public String getLocStrID()	הפעולה מחזירה locationID, כלומר מחרוזת שידוע שהיא מייצגת מיקום.

מחלקת Move:

שם הפעולה	תיעוד הפעולה
public Move()	בנאי היוצר Move חדש.
public Move(Move other)	פעולה בונה מעתיקה.
public Move(Location startingLoc, Location endingLoc)	בנאי היוצר Move חדש עם שני מיקומים המתקבלים.
public Move(byte[] coordinatesArr)	בנאי היוצר Move חדש ממערך ביטים של קורדינאטות.
public void setMove(Location startingLoc, Location endingLoc)	פעולה זו שמה את המיקומים המתקבלים כפי שהם (ולא יוצרת מיקומים חדשים).
public Location getStartingLoc()	פעולה המחזירה את המיקום ההתחלתי.
public Location getEndingLoc()	פעולה המחזירה את המיקום הסופי.
public String getMoveID()	פעולה המחזירה את Move כמחרוזת.

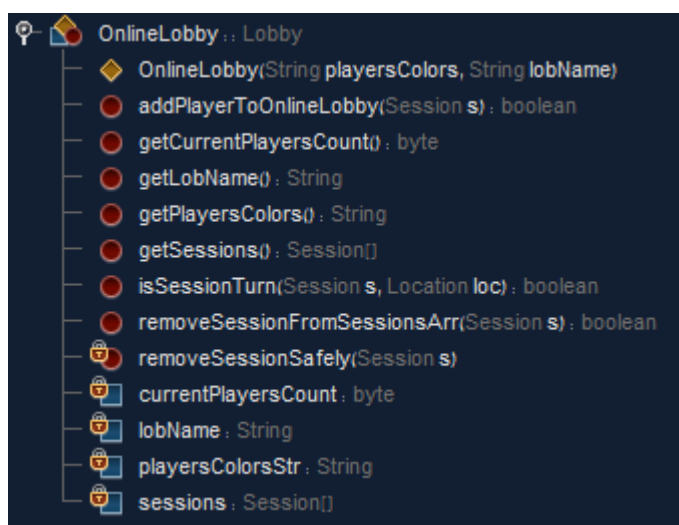
מחלקת LocalLobby:

מחלקה זו יורשת מLobby.

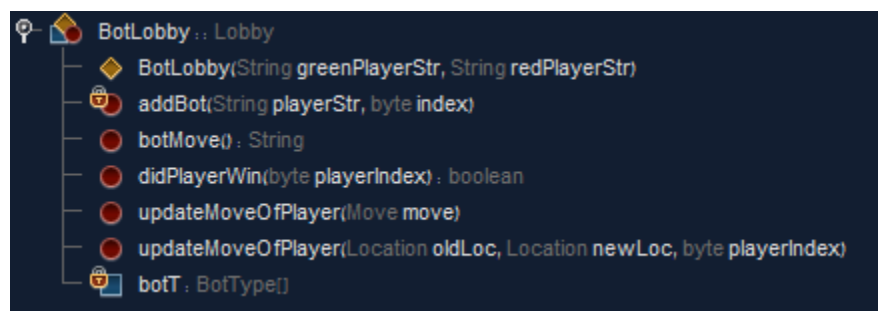


שם הפעולה	תיעוד הפעולה
public LocalLobby(String playersColors)	בנאי מחלקת LocalLobby. בונה LocalLobby חדש עם כמות השחקנים כאורך מחרוזת צבעי השחקנים שהתקבלה.

מחלקת OnlineLobby:



שם הפעולה	תיעוד הפעולה
public OnlineLobby(String playersColors, String lobName)	בנאי היוצר OnlineLobby חדש.
public boolean isSessionTurn(Session s, Location loc)	פעולה מחזירה אמת אם זהו התור של השחקן, אחרת מחזירה שקר.
public Session[] getSessions()	הפעולה מחזירה את מערך sessions
public String getLobName()	הפעולה מחזירה את שם הלובי lobName
public byte getCurrentPlayersCount()	הפעולה מחזירה את מספר השחקנים שבלובי currentPlayerCount כעת
public boolean addPlayerToOnlineLobby(Session s)	הפעולה מוסיפה Session למערך sessions ומחזירה אמת אם לאחר ההוספה צריך להתחיל במשחק, שקר אחרת.
public boolean removeSessionFromSessionsArr(Session s)	הפעולה מוציאה session ממערך sessions ומחזירה אמת אם לאחר ההוצאה שלו יש לסגור את הלובי (מצב זה קורה כאשר המשחק כבר התחיל, או כאשר המשחק עוד לא התחיל ולא נשאר אף שחקן בלובי).
private void removeSessionSafely(Session s)	הפעולה מוציאה session ממערך sessions.

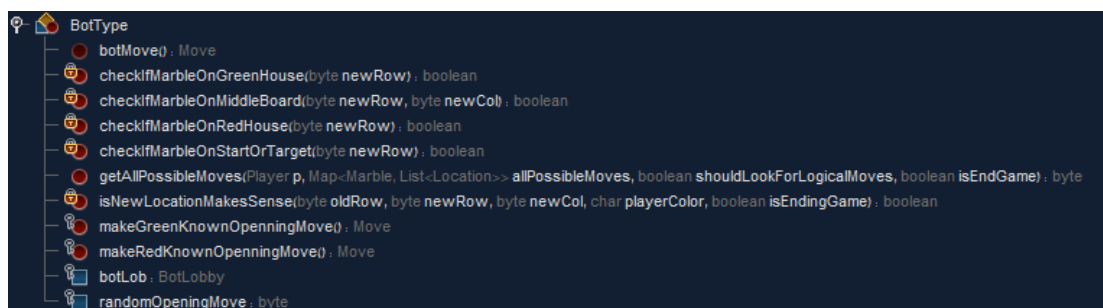
מחלקת BotLobby:

מחלקה זו יורשת מ Lobby ומכילה מערך של BotType – כלומר מערך שמייצג לוגיקה של בוט ים שנבחרו.

שם הפעולה	תיעוד הפעולה
public BotLobby(String greenPlayerStr, String redPlayerStr)	בנאי מחלקת BotLobby. בונה BotLobby בגודל 2, עבור הצבעים ירוק ואדום. בנוסף מזמן עבור כל צבע את הפעולה addBot להוספת לוגיקה למי שנדרש.
private void addBot(String playerStr, byte index)	מקבלת אינדקס וסטרינג שמייצג איזה סוג שחקן מדובר, ובמידה ומדובר בסוג כלשהו של בוט, הפעולה תוסיף למערך BotType במיקום המתאים את הבוט שהתבקש על ידי השחקן.
public String botMove()	מבצעת פעולה של הבוט עבור השחקן הנוכחי, ומחזירה מחרוזת המייצגת את המהלך.
public boolean didPlayerWin(byte playerIndex)	פעולה מחזירה אמת אם שחקן במקום האינדקס ניצח אחרת מחזירה שקר.
public void updateMoveOfPlayer(Move move)	פעולה מקבלת Move ומבצעת עדכון בעבור השחקן הנוכחי.
public void updateMoveOfPlayer(Location oldLoc, Location newLoc, byte playerIndex)	פעולה מקבלת מיקום ישן, מיקום חדש, ואינדקס של השחקן, ומבצעת עדכון פעולה לאותו השחקן הנדרש.

מחלקת BotType – מחלקה אבסטרקטית:

מחלקה זו מהווה בסיס לשאר סוגי הלוגיקות לבוטים הממשיים – AlphaBetaBot | MonteCarloBot.

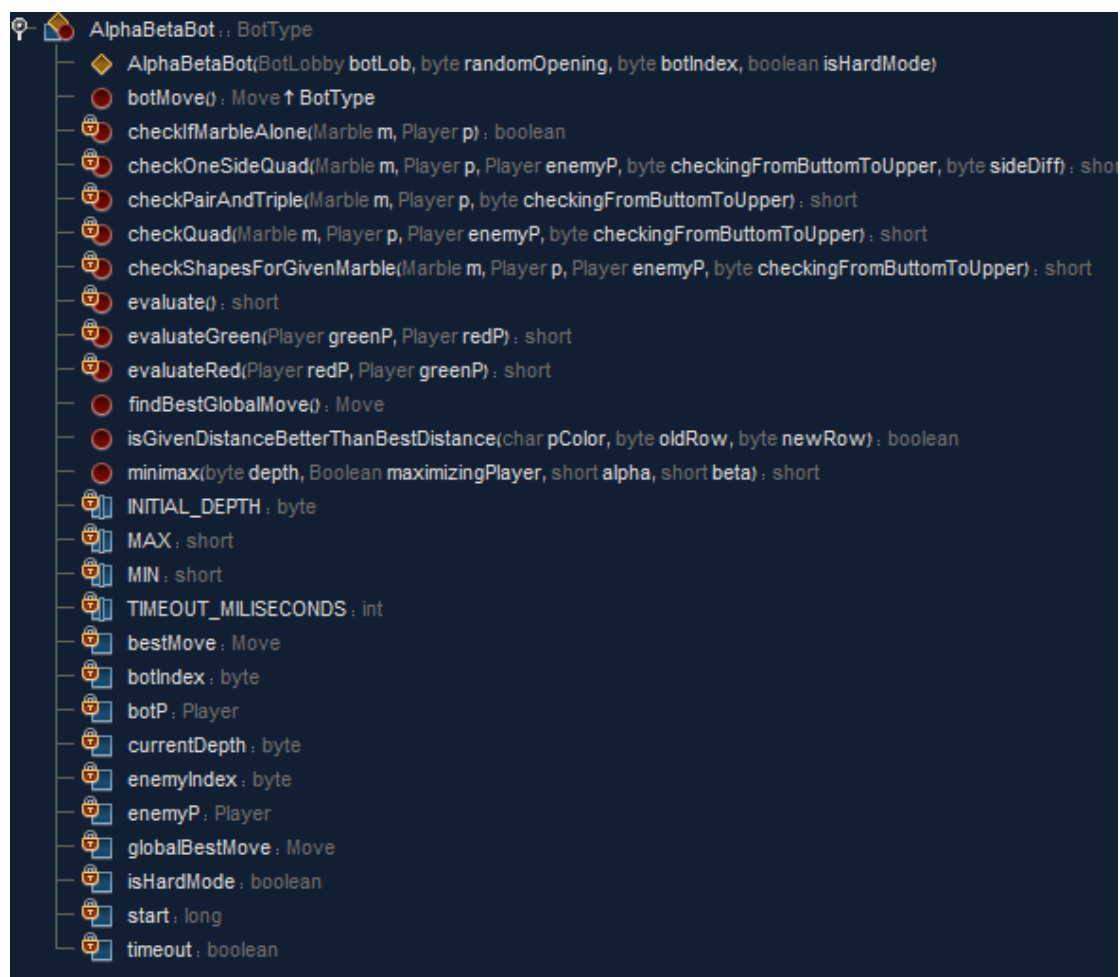


שם הפעולה	תיעוד הפעולה
public abstract Move botMove()	פעולה אבסטרקטית שכל מי שירש מ BotType צריך לממש. הפעולה מחזירה Move, כלומר תזוזה של הבוט
private boolean isNewLocationMakesSense(byte oldRow, byte newRow, byte newCol, char playerColor, boolean isEndingGame)	הפעולה מקבלת את צבע השחקן: אם הוא ירוק, הפעולה בודקת שהשורה החדשה אליה השחקן הולך גדולה יותר מהשורה הישנה בה הוא היה – כלומר השחקן התקדם קדימה. ההפך הוא הנכון אם הצבע הוא אדום. כמו-כן במידה isEndingGame הוא אמת, הפעולה תאפשר גם תזוזה בשורה הראשונה מהיציאה של הבית של היריב. מטרת הפעולה היא לצמצם אפשרויות תזוזה לא הגיוניות כדי שנוכל לחקור אפשרויות הגיוניות.
public byte getAllPossibleMoves(Player p, Map<Marble, List<Location>> allPossibleMoves, boolean shouldLookForLogicalMoves, boolean isEndingGame)	הפעולה מחזירה את כמות האפשרויות שהפעולה שמה בתוך allPossibleMoveses. אם shouldLookForLogicalMoves שקר הפעולה תחזיר את כל האפשרויות שיש לשחקן. אם הוא אמת הפעולה תחזיר את כל האפשרויות ההגיוניות שיש לשחקן בהתאם לפעולה isNewLocationMakesSense ובהתאם ל isEndGame שמשפיע בתוך isNewLocationMakesSense
private boolean checkIfMarbleOnMiddleBoard(byte newRow, byte newCol)	פעולה מחזירה אמת אם השורה והעמודה שהתקבלו נמצאים באמצע הלוח אחרת מחזירה שקר.
private boolean checkIfMarbleOnGreenHouse(byte newRow)	פעולה מחזירה אמת אם השורה נמצאת בבית של השחקן הירוק.

פעולה מחזירה אמת אם השורה נמצאת בבית של השחקן האדום.	private boolean checkIfMarbleOnRedHouse(byte newRow)
פעולה מחזירה אמת אם השורה נמצאת בבית של השחקן הירוק או של השחקן האדום.	private boolean checkIfMarbleOnStartOrTarget(byte newRow)
הפעולה מכילה מטריצה של Move בה בכל מערך יש רצף פעולות התחלתיות שידוע שהן פתיחות טובות עבור השחקן הירוק.	protected Move makeGreenKnownOpeningMove()
הפעולה מכילה מטריצה של Move בה בכל מערך יש רצף פעולות התחלתיות שידוע שהן פתיחות טובות עבור השחקן האדום.	protected Move makeRedKnownOpeningMove()

מחלקת AlphaBetaBot:

המחלקה יורשת מBotType ומכילה את הלוגיקה עבור בוט של AlphaBeta.

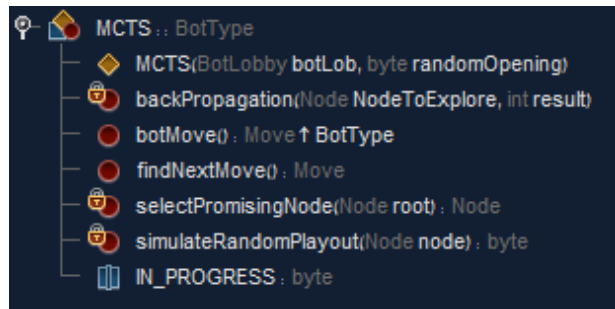


שם הפעולה	תיעוד הפעולה
public AlphaBetaBot(BotLobby botLob, byte randomOpening, byte botIndex, boolean isHardMode)	בנאי מחלקת AlphaBetaBot. הבנאי מקבל את לובי הבוט ממנו זומן, אינדקס של מהלך פתיחה רנדומלי, את האינדקס של הבוט והאם הוא צריך לתפעל ברמה קשה.
@Override public Move botMove()	הפעולה דורסת את פעולה botMove ממחלקת BotType. הפעולה מחזירה מהלך פתיחה אם התור של המשחק הוא מתחת לחמש, אחרת מפעילה את הפעולה findBestGlobalMove ומחזירה את התוצאה.
public Move findBestGlobalMove()	הפעולה מבצעת Iterative AlphaBeta Deepening כלומר, כל עוד נשאר זמן אנו נזמן את פעולת minimax ובכל פעם נחפש בעומק גדול ב-1 מהעומק הקודם כי אולי נמצא מהלך טוב יותר.
public short minimax(byte depth, Boolean maximizingPlayer, short alpha, short beta)	הפעולה מבצעת את אלגוריתם alphabeta עד אשר הגענו לעומק הרצוי או עד שנגמר הזמן לפעולה.
private short evaluate()	הפעולה מחזירה את ניקוד המשחק הנוכחי.
public boolean isGivenDistanceBetter ThanBestDistance(char pColor, byte oldRow, byte newRow)	הפעולה מחזירה אמת אם המרחק בין הצעד המתקבל גדול יותר מצעד הטוב ביותר ב bestMove.
private short evaluateGreen(Player greenP, Player redP)	הפעולה מחשבת את ניקוד השחקן הירוק.
private short evaluateRed(Player redP, Player greenP)	הפעולה מחשבת את ניקוד השחקן האדום.
private boolean checkIfMarbleAlone(Marble m, Player p)	הפעולה בודקת האם הכלי שהתקבל לא נמצא ליד אף כלי אחר של אותו השחקן.
private short checkShapesForGivenMarble(Marble m, Player p, Player enemyP, byte checkingFromButtomToUpper) {	הפעולה בודקת צורות של כלי שהתקבל – צמד, משולש או צמד כפול אחד אחרי השני, ומחזירה את הניקוד עבור הצורות של אותו הכלי.
private short checkPairAndTriple(Marble m, Player p, byte checkingFromButtomToUpper)	הפעולה בודקת צמדיים אפשריים ומשולש עם הכי הנתון, ומחזירה ניקוד בהתאם לצורה שהוא מרכיב (אם הוא מרכיב בכלל).
private short checkQuad(Marble m, Player p, Player enemyP, byte checkingFromButtomToUpper)	הפעולה בודקת צמדים כפולים משני הצדדים עבור כלי נתון.
private short checkOneSideQuad(Marble m, Player p, Player enemyP, byte	הפעולה בודקת צד אחד של צמד כפול עבור כלי נתון.

	checkingFromButtomToUpper, byte sideDiff)
--	--

מחלקת MCTS:

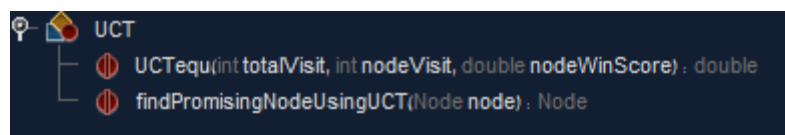
מכילה את הלוגיקה עבור MonteCarlo.



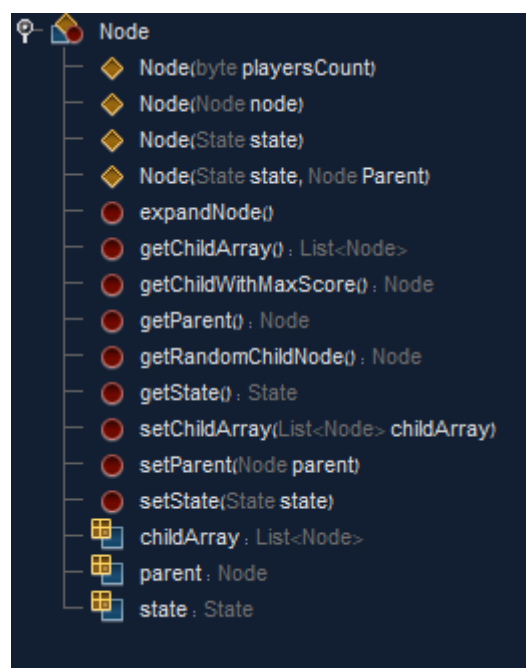
שם הפעולה	תיעוד הפעולה
public MonteCarlo(BotLobby botLob, byte randomOpening)	בנאי מחלקת AlphaBetaBot. הבנאי מקבל את לובי הבוט ממנו זומן ואינדקס של מהלך פתיחה רנדומלי
@Override public Move botMove()	הפעולה דורסת את פעולה botMove ממחלקת BotType. הפעולה מחזירה מהלך פתיחה אם התור של המשחק הוא מתחת לחמש, אחרת מפעילה את הפעולה findNextMove ומחזירה את התוצאה.
public Move findNextMove()	הפעולה מבצעת את אלגוריתם MonteCarlo בכמות הזמן המוקצבת למהלך של הבוט.
private Node selectPromisingNode(Node root)	הפעולה מקבלת את שורש העץ ומחזירה את העלה שהכי כדאי לחקור אותו.
private byte simulateRandomPlayout(Node node)	הפעולה מקבלת עלה ומריצה סימולצית משחק מהמצב שלו.
private void backPropagation(Node NodeToExplore, int result)	הפעולה עולה במעלי הnoden שהתקבל ומעדכנת בכל צומת את תוצאת המשחק של הnoden ומעלה בו את כמות הביקורים באחד.

מחלקת UCT:

מחלקה סטטית המכילה את הפעולות לחישוב נוסחת הUCT



שם הפעולה	תיעוד הפעולה
public static Node findPromisingNodeUsingUCT(Node node)	הפעולה עוברת על כל הילדים של node שהתקבל ומחזירה את הילדה עם תוצאת הUCT הגדולה ביותר
public static double UCTequ(int totalVisit , int nodeVisit , double nodeWinScore)	הפעולה מקבלת פרטים של node ומחזירה את תוצאת הUCT שלו

מחלקת Node:

שם הפעולה	תיעוד הפעולה
public Node(byte playersCount)	פעולה בונה המקבלת כמות שחקנים.
public Node(Node node)	פעולה בונה מעתיקה
public Node(State state)	פעולה בונה המקבלת מצב משחק נוכחי.
public Node(State state , Node Parent)	פעולה בונה המקבלת מצב משחק נוכחי, והורה לnode.
private byte simulateRandomPayout(Node node)	הפעולה מקבלת עלה ומריצה סימולציית משחק מהמצב שלו.
public Node getRandomChildNode()	הפעולה מחזירה node אקראי מתוך רשימת הילדים של הnode הנוכחי.
public Node getChildWithMaxScore()	הפעולה מחזירה את הילד עם התוצאה הכי גדולה.

public void expandNode()	הפעולה אמורה לשמש לעלים, ומטרת להוסיף לnode הנוכחי ילדים לפי המהלכים האפשריים מהמצב הנוכחי של node.
--------------------------	---

3.2 רשימת הנתונים

מחלקת WebSocketServer:

שם הנתון	פירוט הנתון
private static final Map<Session, Lobby> sessionsDict = new HashMap<>();	מילון המכיל עבור כל Session המחובר לסרבר את Lobby שלו.
private static final Map<String, OnlineLobby> onlineLobbiesList = new HashMap<>();	מילון הלובים שהם אונליין והמשחק בהם עדיין לא התחיל.
private boolean isConnectionValid = true;	דגל המורה האם התקשורת תקינה בין הלקוח לשרת. משתנה במידה ומתרחשת שגיאה.

מחלקת Lobby:

שם הנתון	פירוט הנתון
protected Board gameBoard;	לוח המשחק של הלובי.
protected byte currPlayerIndex;	האינדקס של השחקן הנוכחי בלובי.
protected short currGameTurn;	מספר התור הנוכחי של הלובי.

מחלקת Board:

שם הנתון	פירוט הנתון
private final static short CELLS_COUNT = 425-9;	קבוע המכיל את כמות הביטים בboardMask. נשים לב שיש 17 שורות ו25 עמודות בלוח הדמקה הסינית, ונשים לב שיש בסך הכל 416 ביטים בboardMask. הסיבה לכך היא שה-9 ביטים האחרונים בכל מקרה יהיו 0 ולכן לא שמרתי אותם, ולכן גם הCELLS_COUNT יהיה 425-9.
public final static byte ROWS = 17;	קבוע שערכו שווה למספר השורות בלוח הדמקה הסינית.

קבוע שערכו שווה למספר העמודות בלוח הדמקה הסינית.	public final static byte COLS = 25;
קבוע – מדובר במסיכה המייצגת את התאים שקיימים בלוח עבור כל ביט שדולק בה.	private final static byte[] boardMask
קבוע – מערך כיוונים שעוזר להתקדם בין מיקום נתון למיקום הבא.	public final static byte[][] dirArr
מערך השחקנים של הלוח הנוכחי.	protected Player[] playersArr;

מחלקת Player:

שם הנתון	פירוט הנתון
private final Marble[] currMarblesPos;	מערך הכלים של השחקן (מיקומם משתנה במהלך המשחק).
private final Location[] startingMarblesPos;	מערך המיקומים ההתחלתי של השחקן (נשאר זהה לאורך כל המשחק).
private final char color;	צבע השחקן.

מחלקת Marble:

שם הנתון	פירוט הנתון
private final Location loc;	המיקום של הכלי.
private boolean isOnStartingSpot;	אם הכלי נמצא בבית של השחקן שלו הדגל יהיה אמת אחרת יהיה שקר.
private boolean isOnWinningSpot;	אם הכלי נמצא בבית של השחקן שממולו הדגל יהיה אמת אחרת יהיה שקר.

מחלקת Location:

שם הנתון	פירוט הנתון
private byte row;	השורה של המיקום.
private byte col;	העמודה של המיקום.

מחלקת Move:

שם הנתון	פירוט הנתון
private Location startingLoc;	המיקום ההתחלתי של המהלך.
private Location endingLoc;	המיקום הסופי של המהלך.

מחלקת OnlineLobby:

שם הנתון	פירוט הנתון
private final Session[] sessions;	מערך המכיל את sessions המחוברים ללובי.
private final String lobName;	שם הלובי.
private final String playersColorsStr;	מחרוזת המכילה את הצבעים המשתתפים בלובי זה.
private byte currentPlayerCount;	כמות שחקנים נוכחית בלובי.

מחלקת BotLobby:

שם הנתון	פירוט הנתון
private final BotType[] botT;	מערך המכיל לוגיקות בוט שונות.

מחלקת BotType:

שם הנתון	פירוט הנתון
protected BotLobby botLob;	ה BotLobby שמכיל את BotType.
protected byte randomOpeningMove;	מספר המייצג פתיחה רנדומלית של הבוט.

מחלקת AlphaBetaBot:

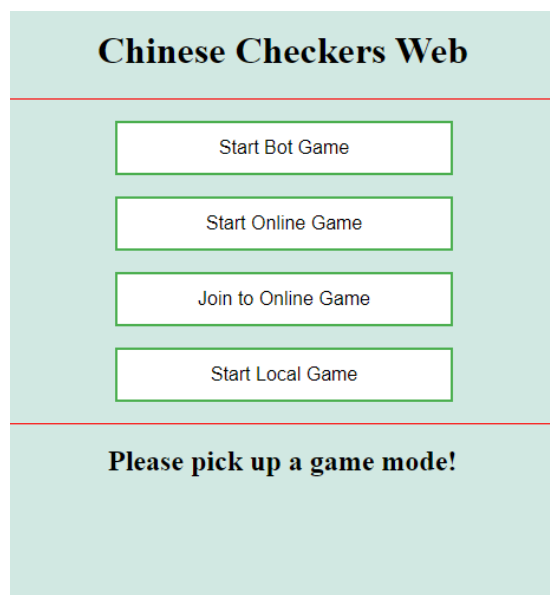
שם הנתון	פירוט הנתון
private final static short MAX = Short.MAX_VALUE;	קבוע המייצג ערך מקסימלי.
private final static short MIN = Short.MIN_VALUE;	קבוע המייצג ערך מינימלי.
private final static byte INITIAL_DEPTH = 3;	קבוע המייצג עומק חיפוש התחלתי.
private final static int TIMEOUT_MILLISECONDS = 1000;	זמן המותר לכל מהלך של הבוט.
private final Player botP;	מצביע לשחקן של AlphaBeta.

האינדקס של שחקן AlphaBeta.	private final byte botIndex;
מצביע לשחקן היריב.	private final Player enemyP;
האינדקס של השחקן היריב.	private final byte enemyIndex;
מכיל את המהלך הבטוח והטוב ביותר.	private Move globalBestMove;
מכיל את המהלך הזמני הטוב ביותר.	private final Move bestMove;
עומק נוכחי.	private byte currentDepth;
זמן ההתחלה של פונקציית findBestGlobalMove במילישניות.	private long start;
דגל המסמן האם נגמר הזמן או לא.	private boolean timeout;

4. מדריך למשתמש

כעת אתן הסבר קצר עבור המשתמש כיצד לתפעל את המערכת כראוי.

מסך הפתיחה:



Chinese Checkers Web

Start Bot Game

Start Online Game

Join to Online Game

Start Local Game

Please pick up a game mode!

כאשר המשתמש ילחץ על Start Bot Game, הוא יצטרך לבחור בסוג השחקנים עבור הצבע הירוק והצבע האדום:



Chinese Checkers Web

Choose player for green side:
Alpha Beta ▼

Choose player for red side:
Alpha Beta Hard ▼

Start Bot Game

Select wanted bots!

לחיצה נוספת על Start Bot Game, תתחיל את המשחק עם הבוטים.

כאשר המשתמש ילחץ על Start Online Game, הוא יצטרך לבחור בצבעים שהוא רוצה שישחקו, ולבחור שם ללובי שלו:

Chinese Checkers Web

Return to Menu

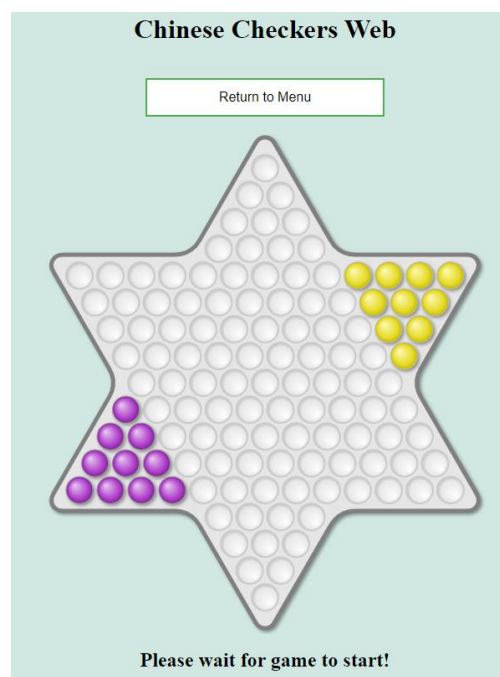
This Is My First Lobby!

- ☐ Green
- ☐ Red
- ☒ Yellow
- ☒ Purple
- ☐ Orange
- ☐ Blue

Start Online Game!

Select wanted colors!

לחיצה על Start Online Game! תפתח את הלובי המבוקש, ועל המשתמש יהיה להמתין לשחקנים נוספים שיצטרפו:

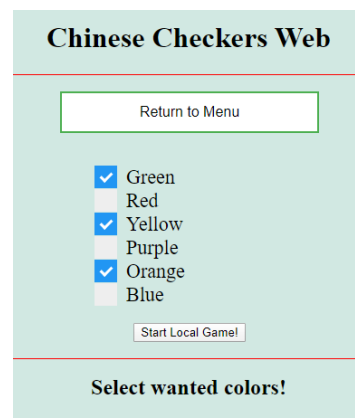


כאשר המשתמש ילחץ על Join To Online Game הוא יקבל רשימת משחקים המכילים לשחקנים נוספים:

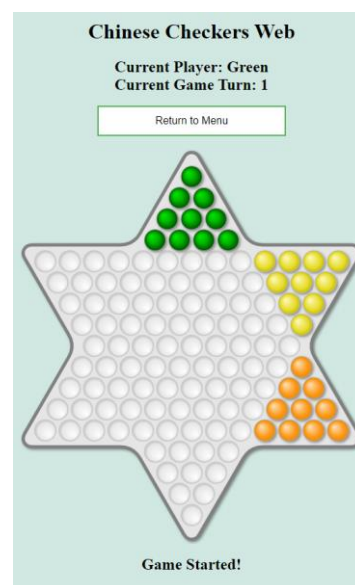


כדי להיכנס ללובי המשתמש צריך ללחוץ על Join Now! ליד הלובי שהוא מעוניין להצטרף.

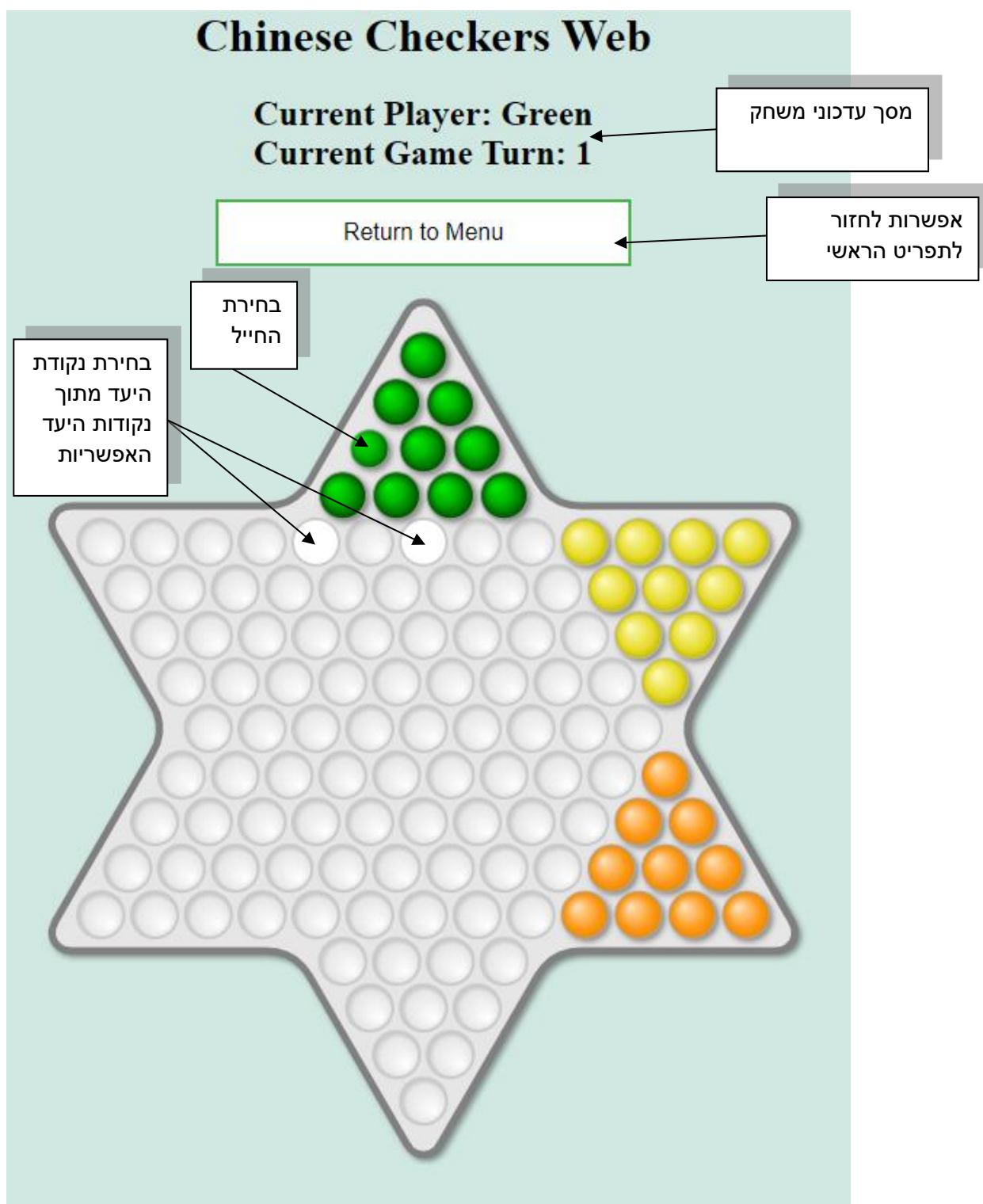
כאשר המשתמש ילחץ על Start Local Game הוא יצטרך לבחור את הצבעים שהוא רוצה שיהיו במשחק:



לחיצה על Start Local Game! תתחיל את המשחק והמשתמש יוכל לשחק:



במהלך המשחק:



5. ביבליוגרפיה

תודות:

מיכאל צ'רנובילסקי על מעקב וייעוץ אחרי הפרויקט, ולאלון חיימוביץ על הייעוץ בנוגע לWebSocket שימוש ב NetBeans ובכלל בשפת התכנות JAVA.

אתרי אינטרנט:

- API עבור WebSocket: <https://docs.oracle.com/javaee/7/tutorial/websocket.htm#GKJIQ5>
- הסבר על אלמנט ה-USE של SVG והוכחה שניתן לשים בו ID: <https://svgwg.org/svg2-draft/single-page.html#struct-UseElement>
- הסבר על JAVA DOCUMENTATION: https://www.tutorialspoint.com/java/java_documentation.htm
- StackOverflow <https://stackoverflow.com/> - נושאים רבים עזרו לי באתר זה, ביניהם:
 1. פוסט מפורט על מונטה קרלו - <https://stackoverflow.com/questions/9056571/monte-carlo-tree-searching-uct-implementation>
 2. מימוש של מונטה קרלו במשחק איקס עיגול - <https://stackoverflow.com/questions/23803186/monte-carlo-tree-search-implementation-for-tic-tac-toe>
- הסבר מפורט על AlphaBeta ומימושו בשחמט: <https://youtu.be/c7OdYGVZdPc>
- הסבר מפורט על Iterative Deepening AlphaBeta ומימושו בשחמט: <https://youtu.be/qiq64olusr0>
- הסבר מפורט על MinMax ועל AlphaBeta באופן תיאורטי: <https://youtu.be/l-hh51ncgDI>
- אתר geeksforgeeks עבור שני מאמרים בנושאי MinMax ו AlphaBeta:
 1. <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>
 2. <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/?ref=rp>
- מימוש (לא שלם) של Monte Carlo עבור איקס עיגול, בשפת JAVA: <https://www.baeldung.com/java-monte-carlo-tree-search>
- Chinese Checkers – In The Light Of AI: <https://slideplayer.com/slide/4625480/>

6. נספח – קוד התוכנית

מחלקת WebSocketServer:

```
package ChineseCheckersPackage;

import java.io.IOException;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;
import javax.json.Json;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.websocket.OnOpen;
import javax.websocket.OnMessage;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

/**
 * Chinese WebSocket server. Handle the connection and communication
 * between
 * every client and other clients or to the game bot.
 *
 * @author Maor Milakandov
 * @version 1.0
 * @since 2019-12-01
 */

@ServerEndpoint("/endpoint")
public class WebSocketServer {
    // Each Session is belonged to a lobby
    private static final Map<Session, Lobby> sessionsDict = new
    HashMap<>();

    // Save Online Lobbys that are waiting for more players
    private static final Map<String, OnlineLobby> onlineLobbiesList =
    new HashMap<>();

    private boolean isConnectionValid = true;

    // -----
    // Server Endpoint methods (onOpen, onMessage, onClose, onError):
    /**
     * This method handles all the new connections to the server by
     * adding the
     * connected session to the sessions dictionary with no signed
     * lobby.
     * @param s the connected session.
     */
    @OnOpen
    public void onOpen(Session s) {
        // Print log message that session joined to the server
        System.out.println(s.getId() + " Log into the server");

        // Add the session to the session dictionary with no lobby
    }
}
```

```

        sessionsDict.put(s, null);
    }

    /**
     * This method handles all the communication between the clients
     and the
     * server by receiving a JSON message and checking the type of the
     request
     * by the client - and then calling the right function to handle
     the request.
     * @param message receive JSON object as a string.
     * @param s the session that sent the message.
     */
    @OnMessage
    public void onMessage(String message,
                          Session s) {
        // Make the message a json object
        JsonReader reader = Json.createReader(new
StringReader(message));
        JsonObject jsonMessage = reader.readObject();

        // Get the type of the message
        String msgType = jsonMessage.getString("type");

        switch (msgType) {
            case "createLobby":
                handleCreateLobby(jsonMessage, s);
                break;
            case "getLobbiesList":
                handleGetLobbiesList(s);
                break;
            case "joinLobby":
                handleJoinLobby(jsonMessage, s);
                break;
            case "exitLobby":
                handleExitLobby(s);
                break;
            case "firstClick":
                handleFirstClick(jsonMessage, s);
                break;
            case "secondClick":
                handleSecondClick(jsonMessage, s);
                break;
            default:
                break;
        }
    }

    /**
     * This method get a session and removes it from the sessions
     dictionary
     * @param s the session of the disconnected client.
     */
    @OnClose
    public void onClose(Session s) {
        // If the session was part of online game, remove the session
        from the
        // online game lobby
        Lobby lob = sessionsDict.get(s);

        // Remove the session to the sessions dictionary
    }

```

```

        sessionsDict.remove(s);

        if (lob instanceof OnlineLobby) {
            removeSessionFromLobbyOnline(s, (OnlineLobby) lob);
        }

    }

    /**
     * This method handle errors that happen on the server.
     * @param s the session of the disconnected client.
     * @param e the throwable error.
     */
    @OnError
    public void onError(Session s, Throwable e) {
        System.out.println("Error Happened: " + e.toString());

        // get StackTraceElements
        // using getStackTrace()
        StackTraceElement[] stktrace
            = e.getStackTrace();

        // print element of stktrace
        for (int i = 0; i < stktrace.length; i++) {

            System.out.println("Index " + i
                               + " of stack trace"
                               + " array conatins = "
                               + stktrace[i].toString());
        }
        this.isConnectionValid = false;
    }

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

    // Handle methods for each type of communication between the
    // clients:

    /**
     * This method handle createLobby request by given client by
     * checking
     * the type of the lobby that the client wants to create, and
     * according
     * to the lobby type the method creates the lobby.
     * @param jsonMessage the given JSON message that contains the
     * gameType
     * that the client wish to create.
     * @param s the given session.
     */
    public void handleCreateLobby(JsonObject jsonMessage,
                                  Session s) {
        // Get the type of the lobby - bot lobby, lan lobby or online
        lobby
        String gameType = jsonMessage.getString("gameType");

        switch (gameType) {
            case "bot":

```

```

        createBotLobby(jsonMessage, s);
        break;
    case "online":
        createOnlineLobby(jsonMessage, s);
        break;
    case "local":
        createLocalLobby(jsonMessage, s);
        break;
    default:
        break;
    }
}

/**
 * This method handle getLobbiesList request by given client by
checking
 * sending the client all the online lobbies that need to be
filled with
 * more players to start the game
 * @param s          the given session.
 */
public void handleGetLobbiesList(Session s) {
    for (OnlineLobby tempLob : onlineLobbiesList.values()) {
        sendToSession(s,
            buildLobDetailsJsonObject(tempLob));
    }
}

/**
 * This method handle createLobby request by given client by
checking
 * the type of the lobby that the client wants to create, and
according
 * to the lobby type the method creates the lobby.
 * @param jsonMessage the given JSON message that contains the
gameType
 *
 * that the client wish to create.
 * @param s          the given session.
 */
public void handleJoinLobby(JsonObject jsonMessage,
    Session s) {
    String lobName = jsonMessage.getString("lobName");
    OnlineLobby lob =
(OnlineLobby)onlineLobbiesList.get(lobName);
    if(lob != null) {
        sessionsDict.replace(s, lob);
        sendToSession(s, buildJoinedOnlineGameJsonObject());
        if (lob.addPlayerToOnlineLobby(s)) {
            sendToSessionsArray(((OnlineLobby)lob).getSessions(),
buildOnlineGameStartedJsonObject());
            sendCurrGameDetails(s, lob);
            onlineLobbiesList.remove(lobName);
        }
    }
    else
        sendToSession(s, buildFailedJoinOnlineGameJsonObject());
}

/**

```

```

    * This method sends the details of the given OnlineLobby to all
    it's players
    * @param lob the given OnlineLobby.
    */
    public void sendOnlineLobDetails(OnlineLobby lob) {
        sendToSessionsArray(lob.getSessions(),
        buildLobDetailsJsonObject(lob));
    }

    /**
    * This method handle exit lobby request from given session.
    * @param s the given session.
    */
    public void handleExitLobby(Session s) {
        // Get the lobby of the session
        Lobby lob = sessionsDict.get(s);
        // Exit the lobby
        exitLobby(s, lob);
        // Send exitedLobby message successfully to the player
        sendToSession(s, buildExitedLobbyJsonObject());
    }

    /**
    * This method exit lobby of given session from given lobby.
    * @param s the given session.
    * @param lob the given lobby
    */
    public void exitLobby(Session s, Lobby lob) {
        // remove the lobby from the sessionDict
        sessionsDict.replace(s, null);

        // If the lobby belongs to online lobby, remove the session
        from the
        // session array of the online lobby
        if (lob instanceof OnlineLobby) {
            removeSessionFromLobbyOnline(s, (OnlineLobby)lob);
        }
    }

    /**
    * This method handle first click of user by checking if the
    clicked marble
    * is of the current player - If so, the method will check what
    moves are
    * available and will send the coordinates to the player,
    otherwise will send
    * invalidMarble to the player.
    * @param jsonMessage the given JSON message that contains first
    click
    *
    * details.
    * @param s the given session.
    */
    public void handleFirstClick(JsonObject jsonMessage,
                                Session s) {
        // Get the first clicked location
        String startLoc = jsonMessage.getString("startLoc");

        // Get the player lobby
        Lobby lob = sessionsDict.get(s);

        // Find all the empty cells that can be reached

```

```

        // from this location
        Location loc = new Location(startLoc);

        if(lob instanceof OnlineLobby &&
            !((OnlineLobby)lob).isSessionTurn(s, loc)) {
            sendToSession(s, buildInvalidMarbleJsonObject());
        }
        else {
            // Get all the possible positions
            String allPossMoves =
lob.getAllPossibleMovesOfClickedMarble(loc);
            if (allPossMoves.equals("invalidMarble")) {
                // Tell the client this is not his marble!
                sendToSession(s, buildInvalidMarbleJsonObject());
            }
            else {
                // Return all the positions to the session
                sendToSession(s,
buildPossibleMovesJsonObject(allPossMoves));
            }
        }
    }

    /**
     * This method handle second click of user by updating the game
board
     * and sending the information to the session(s) in the lobby - if
     * the session is on BotLobby, this method will order the bot to
make a move
     * and inform the player about it.
     * @param jsonMessage the given JSON message that contains second
click
     *
     * details.
     * @param s           the given session.
     */
    public void handleSecondClick(JsonObject jsonMessage,
                                Session s) {
        // Get the clicked locations
        String startLocStr = jsonMessage.getString("startLoc");
        String endLocStr = jsonMessage.getString("endLoc");

        Location startLoc = new Location(startLocStr);
        Location endLoc = new Location(endLocStr);

        // Get the player lobby
        Lobby lob = sessionsDict.get(s);

        makeMoveWithPlayer(s, lob, startLoc, endLoc, jsonMessage);
    }

    /**
     * This method make a move with a player and sends the info about
the move
     * to the clients.
     * @param s           the given session (client).
     * @param lob         the given bot lobby.
     * @param startLoc    the starting location.
     * @param endLoc      the ending location
     * @param jsonMessage the given jsonMessage.

```



```

    */
    public void makeMoveWithPlayer(Session s,
                                   Lobby lob,
                                   Location startLoc,
                                   Location endLoc,
                                   JsonObject jsonMessage) {

        // Update the move of the player
        lob.updateMoveOfPlayer(startLoc,
                               endLoc);

        // Check if the game turn is above 25 and if the player have
a piece
        // at his home
        /*if (lob.getCurrGameTurn() > 25 &&
lob.didPlayerStayInHome()) {
            ;
        }*/

        // Send second click to player(s) in lobby
        sendToSessionsInLobby(s, lob, jsonMessage);

        // If the player didn't won
        if(!checkIfPlayerWon(s, lob)) {
            // Update the curr player index
            lob.updateGameDetailsOfLobby();
            // send the curr game details
            sendCurrGameDetails(s, lob);

            // Do move with bot if lobby is BotLobby
            if (lob instanceof BotLobby) {
                makeMoveWithBot(s, (BotLobby)lob);
            }
        }
    }

    /**
     * This method returns true if the current player in the lobby won
and inform
     * all the sessions in the lobby about it if he did.
     * @param s the given session (player).
     * @param lob the given lobby.
     * @return This method returns true if given player won and inform
all the
     * sessions in the lobby about it if he did.
     */
    public boolean checkIfPlayerWon(Session s,
                                   Lobby lob) {

        // Check if player win
        if (lob.didPlayerWin()) {
            // Send game won message to player(s) in lobby
            sendToSessionsInLobby(s, lob,

buildGameWonJsonObject(lob.getCurrColor()));
            // Exit the lobby
            exitLobby(s, lob);
            return true;
        }
        return false;
    }
}

```

```

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

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

// Auxiliary methods:

/**
 * This method handle new bot lobby request from given session.
 * @param jsonMessage the given request.
 * @param s           the given session.
 */
public void createBotLobby(JsonObject jsonMessage, Session s) {
    String greenPlayerStr = jsonMessage.getString("greenP");
    String redPlayerStr = jsonMessage.getString("redP");
    BotLobby newLobby = new BotLobby(greenPlayerStr,
redPlayerStr);
    joinLobbyToSessionsDictAndInformTheClient(s, newLobby);

    if(!redPlayerStr.equals("human") &&
!greenPlayerStr.equals("human")) {
        simulateTwoBotsGame(s, newLobby);
    }
    else if (!greenPlayerStr.equals("human"))
        makeMoveWithBot(s, newLobby);
}

/**
 * This method simulate game between 2 bots and inform the client
about every
 * move.
 * @param s    the given session (client).
 * @param lob the given bot lobby.
 */
public void simulateTwoBotsGame(Session s,
                                BotLobby lob) {
    boolean didPlayerTwoWin = false;

    // Make moves with bots as long as either won
    while(isConnectionValid && !didPlayerTwoWin &&
!makeMoveWithBot(s, lob))
        if(makeMoveWithBot(s, lob))
            didPlayerTwoWin = true;
}

/**
 * This method handle new local lobby request from given session.
 * @param jsonMessage the given request.
 * @param s           the given session.
 */
public void createLocalLobby(JsonObject jsonMessage,
                             Session s) {
    String playersColors = jsonMessage.getString("pColors");
    Lobby newLobby = new LocalLobby(playersColors);
    joinLobbyToSessionsDictAndInformTheClient(s, newLobby);
}

/**
 * This method handle new online lobby request from given session.
 * @param jsonMessage the given request.

```

```

    * @param s          the given session.
    */
    public void createOnlineLobby(JsonObject jsonMessage,
                                Session s) {
        String pColors = jsonMessage.getString("pColors");
        byte pCount = (byte) (pColors.length());
        String lobName = jsonMessage.getString("lobName");

        if (onlineLobbiesList.containsKey(lobName))
            sendToSession(s, buildNameTakenJsonObject());
        else
        {
            OnlineLobby newLobby = new OnlineLobby(pColors, lobName);
            onlineLobbiesList.put(lobName, newLobby);
            newLobby.addPlayerToOnlineLobby(s);
            // Put the lobby to the session on the sessionsDict
            sessionsDict.replace(s, newLobby);
            // Inform the client he joined the lobby
            sendToSession(s, buildJoinedOnlineGameJsonObject());
        }
    }

    /**
     * This method make a move with a bot and sends the info about the
     move
     * to the client.
     * @param s    the given session (client).
     * @param lob  the given bot lobby.
     * @return the method returns true if the bot won otherwise
     returns false.
     */
    public boolean makeMoveWithBot(Session s,
                                   BotLobby lob) {
        // Make move with bot
        String moveStrID = lob.botMove();
        // Send the move to the client
        sendToSession(s, buildSecondClickJsonObject(moveStrID));

        // If the bot didn't win
        if (!checkIfPlayerWon(s, lob)) {
            // Update the curr player index
            lob.updateGameDetailsOfLobby();
            // Send the curr game details
            sendCurrGameDetails(s, lob);

            // Return false if the bot didn't win
            return false;
        }

        // Return true if the bot win
        return true;
    }

    /**
     * This method removes given session from his online lobby.
     * @param s    the given session (client).
     * @param lob  the given online lobby.
     */
    public void removeSessionFromLobbyOnline(Session s,
                                              OnlineLobby lob) {
        if (lob.removeSessionFromSessionsArr(s)) {

```

```

        Session[] sessionsArr = lob.getSessions();
        for (Session session : sessionsArr) {
            if (session != null && session != s) {
                sessionsDict.replace(session, null);
                sendToSession(session,
buildGameCanceledJsonObject());
            }
        }
        onlineLobbiesList.remove(lob.getLobName());
    }
}

/**
 * This method gets a new made up lobby and:
 * - Puts the lobby to the session on the sessionsDict
 * - Sends the client a message that he joined the lobby
successfully
 * - Sends the client the game details
 * @param s The given session.
 * @param newLobby The new made up lobby.
 */
public void joinLobbyToSessionsDictAndInformTheClient(Session s,
Lobby
newLobby) {
    // Put the lobby to the session on the sessionsDict
    sessionsDict.replace(s, newLobby);
    // Send the client a message that he joined the lobby
successfully
    sendStartedGame(s);
    // Send the client the game details
    sendCurrGameDetails(s, newLobby);
}

/**
 * This method gets a session and sends the startedGame JSON
message to
 * the client.
 * @param s The given session.
 */
public void sendStartedGame(Session s) {
    sendToSession(s, buildStartedGameJsonObject());
}

/**
 * This method gets session and lobby and sends the current game
details
 * to the player(s) in the lobby.
 * @param s The given session.
 * @param lob The given lobby.
 */
public void sendCurrGameDetails(Session s,
Lobby lob) {
    /*if (lob instanceof OnlineLobby && lob.getCurrGameTurn() ==
-1) {

        }
        else*/
        sendToSessionsInLobby(s, lob,
buildCurrGameDetailsJsonObject(lob));
    }
}

```

```

// -----
// -----
// Methods for building new JSON objects:

/**
 * This method returns new startedGame typed JSON object.
 * @return JsonObject The new JSON of 'startedGame' type.
 */
public JsonObject buildStartedGameJsonObject() {
    return Json.createObjectBuilder()
        .add("type", "startedGame")
        .build();
}

/**
 * This method gets a lobby and builds a new JSON Object that
contains the
 * game details (the color of the current player, and the current
game turn).
 * @param lob The given lobby.
 * @return JsonObject The new JSON object that contains the game
details.
 */
public JsonObject buildCurrGameDetailsJsonObject(Lobby lob) {
    return Json.createObjectBuilder().add("type",
"currGameDetails")
        .add("currColor",
lob.getCurrColor())
        .add("currTurn",
lob.getCurrGameTurn())
        .build();
}

/**
 * This method returns new invalidMarble typed JSON object.
 * @return JsonObject The new JSON of 'invalidMarble' type.
 */
public JsonObject buildInvalidMarbleJsonObject() {
    return Json.createObjectBuilder().add("type",
"invalidMarble").build();
}

/**
 * This method returns new possibleMoves typed JSON object.
 * @param allPossMoves The given all possible moves String.
 * @return JsonObject The new JSON of 'possibleMoves' type.
 */
public JsonObject buildPossibleMovesJsonObject(String
allPossMoves) {
    return Json.createObjectBuilder().add("type",
"possibleMoves")
        .add("possibleMoves",
allPossMoves)
        .build();
}

/**
 * This method returns new exitedLobby typed JSON object.

```

```

    * @return JsonObject The new JSON of 'exitedLobby' type.
    */
    public JsonObject buildExitedLobbyJsonObject() {
        return Json.createObjectBuilder().add("type",
"exitedLobby").build();
    }

    /**
    * This method returns new secondClick typed JSON object.
    * @param moveStrID The given moveID which contains the old
position
    * and the new position.
    * @return JsonObject The new JSON of 'secondClick' type.
    */
    private JsonObject buildSecondClickJsonObject(String moveStrID) {
        return Json.createObjectBuilder()
            .add("type", "secondClick")
            .add("startLoc", moveStrID.substring(0,4))
            .add("endLoc", moveStrID.substring(4,8))
            .build();
    }

    /**
    * This method returns new gameWon typed JSON object.
    * @param color The char of the player that won
    * @return JsonObject The new JSON of 'gameWon' type.
    */
    private JsonObject buildGameWonJsonObject(char color) {
        return Json.createObjectBuilder()
            .add("type", "gameWon")
            .add("winningColor", color)
            .build();
    }

    /**
    * This method returns new gameCanceled typed JSON object.
    * @return JsonObject The new JSON of 'gameCanceled' type.
    */
    private JsonObject buildGameCanceledJsonObject() {
        return Json.createObjectBuilder()
            .add("type", "gameCanceled")
            .build();
    }

    /**
    * This method returns new nameTaken typed JSON object.
    * @return JsonObject The new JSON of 'nameTaken' type.
    */
    private JsonObject buildNameTakenJsonObject() {
        return Json.createObjectBuilder()
            .add("type", "nameTaken")
            .build();
    }

    /**
    * This method returns new onlineGameStarted typed JSON object.
    * @return JsonObject The new JSON of 'onlineGameStarted' type.
    */
    private JsonObject buildOnlineGameStartedJsonObject() {
        return Json.createObjectBuilder()
            .add("type", "onlineGameStarted")

```

```

        .build();
    }

    /**
     * This method returns new lobDetails typed JSON object.
     * @return JsonObject The new JSON of 'lobDetails' type.
     */
    private JsonObject buildLobDetailsJsonObject(OnlineLobby lob) {
        return Json.createObjectBuilder()
            .add("type", "lobDetails")
            .add("lobName", lob.getLobName())
            .add("lobCount", lob.getCurrentPlayersCount())
            .add("lobMaxPlayers", lob.getSessions().length)
            .add("colorsStr", lob.getPlayersColors())
            .build();
    }

    /**
     * This method returns new joinedOnlineGame typed JSON object.
     * @return JsonObject The new JSON of 'joinedOnlineGame' type.
     */
    public JsonObject buildJoinedOnlineGameJsonObject() {
        return Json.createObjectBuilder()
            .add("type", "joinedOnlineGame")
            .build();
    }

    /**
     * This method returns new failedJoinOnlineGame typed JSON object.
     * @return JsonObject The new JSON of 'failedJoinOnlineGame' type.
     */
    public JsonObject buildFailedJoinOnlineGameJsonObject() {
        return Json.createObjectBuilder()
            .add("type", "failedJoinOnlineGame")
            .build();
    }

    // -----

    // -----

    // Methods for sending JSON message to the clients:

    /**
     * This method get a lobby and sends a JSON object as string to
the sessions
     * in the lobby.
     * @param s          The session that belong to the lobby.
     * @param lob        The given lobby.
     * @param jsonMessage The given JSON message.
     */
    public void sendToSessionsInLobby(Session s,
                                     Lobby lob,
                                     JsonObject jsonMessage) {
        if (lob instanceof OnlineLobby) {
            sendToSessionsArray(((OnlineLobby) lob).getSessions(),
                               jsonMessage);
        }
    }

```

```

        else {
            sendToSession(s, jsonMessage);
        }
    }

    /**
     * This method sends a JSON object as string to given sessions
     array.
     * @param sessions The given sessions array.
     * @param message The given JSON message.
     */
    private void sendToSessionsArray(Session[] sessions,
                                     JsonObject message) {
        for (Session session : sessions) {
            if (session != null)
                sendToSession(session, message);
        }
    }

    /**
     * This method sends a JSON object as String to given session.
     * @param s The given session.
     * @param message The given JSON message.
     */
    private void sendToSession(Session s,
                               JsonObject message) {
        try {
            s.getBasicRemote().sendText(message.toString());
        } catch (IOException ex) {
            onError(s, ex);
        }
    }

    // -----
    -----
}

```


:Lobby מחלקת

```

package ChineseCheckersPackage;

public abstract class Lobby {

    protected Board gameBoard;
    protected byte currPlayerIndex;
    protected short currGameTurn;

    /**
     * The method builds new Lobby.
     * @param countOfPlayers the count of the players in the lobby.
     */
    public Lobby (byte countOfPlayers) {
        this.gameBoard = new Board(countOfPlayers);
        this.currPlayerIndex = 0;
        this.currGameTurn = -1;
    }

    /**
     * The method returns the gameBoard of the lobby.
     * @return The game board of the lobby.
     */
    public Board getGameBoard() {
        return this.gameBoard;
    }

    /**
     * The method returns the getCurrPlayerIndex of the lobby.
     * @return The getCurrPlayerIndex.
     */
    public byte getCurrPlayerIndex() {
        return this.currPlayerIndex;
    }

    /**
     * The method returns the getCurrGameTurn of the lobby.
     * @return The getCurrGameTurn.
     */
    public short getCurrGameTurn() {
        return this.currGameTurn;
    }

    /**
     * The method adds new Player to the gameBoard and updates the
lobby
     * game details.
     * @param p the given player.
     */
    public void addPlayer(Player p) {
        this.gameBoard.setPlayer(p, this.currPlayerIndex);
        updateGameDetailsOfLobby();
    }

    /**
     * The method adds new Players to the gameBoard based on given
playersColors
     * String
     * @param playersColors the given players colors String.
     */
    protected void addColorsToGame(String playersColors) {
        if (playersColors.indexOf('g') != -1)

```

```

        addPlayer(createGreenPlayer());
    if (playersColors.indexOf('y') != -1)
        addPlayer(createYellowPlayer());
    if (playersColors.indexOf('o') != -1)
        addPlayer(createOrangePlayer());
    if (playersColors.indexOf('r') != -1)
        addPlayer(createRedPlayer());
    if (playersColors.indexOf('p') != -1)
        addPlayer(createPurplePlayer());
    if (playersColors.indexOf('b') != -1)
        addPlayer(createBluePlayer());
}

/**
 * This method returns new green Player.
 * @return Player The new green Player.
 */
protected Player createGreenPlayer() {
    return new Player(new byte[]{ 0, 12, 1, 13, 1, 11, 2, 14, 2,
12, 2, 10,
                                3, 15, 3, 13, 3, 11, 3, 9 },
                        'g');
}

/**
 * This method returns new red Player.
 * @return Player The new red Player.
 */
protected Player createRedPlayer() {
    return new Player(new byte[]{ 16, 12, 15, 11, 15, 13, 14, 10,
14, 12,
                                14, 14, 13, 9, 13, 11, 13, 13,
13, 15 },
                        'r');
}

/**
 * This method returns new yellow Player.
 * @return Player The new yellow Player.
 */
protected Player createYellowPlayer() {
    return new Player(new byte[]{ 4, 24, 5, 23, 4, 22, 6, 22, 5,
21, 4, 20,
                                7, 21, 6, 20, 5, 19, 4, 18 },
                        'y');
}

/**
 * This method returns new purple Player.
 * @return Player The new purple Player.
 */
protected Player createPurplePlayer() {
    return new Player(new byte[]{ 12, 0, 11, 1, 12, 2, 10, 2, 11,
3, 12, 4,
                                9, 3, 10, 4, 11, 5, 12, 6},
                        'p');
}

/**
 * This method returns new orange Player.
 * @return Player The new orange Player.

```

```

    */
    protected Player createOrangePlayer() {
        return new Player(new byte[]{ 12, 24, 12, 22, 11, 23, 12, 20,
11, 21,
                                     10, 22, 12, 18, 11, 19, 10, 20,
9, 21},
                                'o');
    }

    /**
     * This method returns new blue Player.
     * @return Player The new blue Player.
     */
    protected Player createBluePlayer() {
        return new Player(new byte[]{ 4, 0, 4, 2, 5, 1, 4, 4, 5, 3,
6, 2, 4, 6,
                                     5, 5, 6, 4, 7, 3},
                                'b');
    }

    /**
     * This method updates the current player index and the current
game turn
     * of this lobby.
     */
    public void updateGameDetailsOfLobby() {
        this.currPlayerIndex++;
        if (this.currPlayerIndex == this.gameBoard.getPlayersCount())
        {
            this.currPlayerIndex = 0;
            this.currGameTurn++;
        }
    }

    /**
     * This function returns the current Player of the lobby.
     */
    public Player getCurrPlayer() {
        return this.gameBoard.getPlayer(this.currPlayerIndex);
    }

    /**
     * This function returns the color of the current Player of the
lobby.
     * @return the color of the player
     */
    public char getCurrColor(){
        return getCurrPlayer().getColor();
    }

    /**
     * This function update move of given coordinates.
     * @param oldLoc the old location of the marble
     * @param newLoc the new location of the marble
     */
    public void updateMoveOfPlayer(Location oldLoc,
                                    Location newLoc) {

this.gameBoard.getPlayer(this.currPlayerIndex).updateMove(oldLoc,

```

```

newLoc);
    }

    /**
     * This function returns if the current player of the lobby won
     the game.
     * @return true if the player won, otherwise returns false.
     */
    public boolean didPlayerWin() {
        return
this.gameBoard.getPlayer(this.currPlayerIndex).didPlayerWin();
    }

    public boolean didPlayerStayAtHome() {
        return
this.gameBoard.getPlayer(this.currPlayerIndex).didPlayerStayAtHome();
    }

    /**
     * This function returns all possible moves of clicked marble as
     StringID
     * @param loc the given location of the marble
     * @return string that contains all the possible next moves
     */
    public String getAllPossibleMovesOfClickedMarble(Location loc) {
        String allPossibleMoves;

if(!this.gameBoard.getPlayer(this.currPlayerIndex).isMarbleBelongsToP
layer(loc)) {
    allPossibleMoves = "invalidMarble";
}
else
    allPossibleMoves = this.gameBoard.allNextMovesStr(loc);
return allPossibleMoves;
}
}

```

:Board מחלקת

```

package ChineseCheckersPackage;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Board {
    //
    =====
    // Consts Board details:
    //
    =====
    // The CELLS_COUNT represents the ammount of bits in the
    boardMask.
    // The last 9 bits are 0 so I didn't wrote them to the board
    mask, so we
    // have to sub them from the CELLS_COUNT.
    private final static short CELLS_COUNT = 425-9;
    // There are 17 rows in the board
    public final static byte ROWS = 17;
    // There are 25 cols in the board
    public final static byte COLS = 25;

    // The boardMash the represents by bits the real cells in the
    Board.
    private final static byte[] boardMask = {
        0x00, 0x08, 0x00, 0x00,
        0x0A, 0x00, 0x00, 0x0A,
        (byte)0x80, 0x00, 0x0A, (byte) 0xA0,
        0x0A, (byte)0xAA, (byte)0xAA, (byte)0xAA,
        (byte)0xAA, (byte)0xAA, (byte)0xA8, (byte)0xAA,
        (byte)0xAA, (byte)0xA8, 0x2A, (byte)0xAA,
        (byte)0xA8, 0x0A, (byte)0xAA, (byte)0xA8,
        0x0A, (byte)0xAA, (byte)0xAA, 0x0A,
        (byte)0xAA, (byte)0xAA, (byte)0x8A, (byte)0xAA,
        (byte)0xAA, (byte)0xAA, (byte)0xAA, (byte)0xAA,
        (byte)0xA8, 0x02, (byte)0xA8, 0x00,
        0x00, (byte)0xA8, 0x00, 0x00,
        0x28, 0x00, 0x00, 0x08};

    // The diraction array.
    public final static byte[][] dirArr = {
        {-1, -1}, // Top-Left
        {-1, 1}, // Top-Right
        {0, 2}, // Right
        {1, 1}, // Buttom-Right
        {1, -1}, // Buttom-Left
        {0, -2} // Left
    };

    // The players array
    private final Player[] playersArr;

    public Board(byte countOfPlayers) {
        this.playersArr = new Player[countOfPlayers];
    }

```

```

    public Board(Board other) {
        this.playersArr = new Player[other.getPlayersArr().length];

        for (byte i = 0; i < this.playersArr.length; i++) {
            this.playersArr[i] = new
Player(other.getPlayersArr()[i]);
        }
    }

    public Player getPlayer(byte playerIndex) {
        return this.playersArr[playerIndex];
    }
    public void setPlayer(Player p, byte playerIndex) {
        this.playersArr[playerIndex] = p;
    }
    public byte getPlayersCount() {
        return (byte) this.playersArr.length;
    }

    public Player[] getPlayersArr() {
        return playersArr;
    }

    public void performMove(byte playerIndex, Move move) {
        this.playersArr[playerIndex].updateMove(move);
    }

    /**
     * This function returns if the any player of the lobby won the
game.
     * @return the index of the player that one. If no one won, return
     * IN_PROGRESS ( = -2)
     */
    public byte checkStatus() {
        for(byte i = 0; i < this.playersArr.length; i++) {
            if(didPlayerWin(i))
                return i;
        }
        return MCTS.IN_PROGRESS;
    }

    /**
     * This function returns if the current player of the lobby won
the game.
     * @return true if the player won, otherwise returns false.
     */
    private boolean didPlayerWin(byte index) {
        return this.playersArr[index].didPlayerWin();
    }

    public List<Move> getAllPossibleMoves(byte playerIndex) {

        List<Move> allPossibleMoves = new ArrayList<>();
        Marble[] marblesArr =
this.playersArr[playerIndex].getCurrMarblesPos();

        for (byte i = 0; i < marblesArr.length; i++) {
            String possibleMoves =
allNextMovesStr(marblesArr[i].getLoc());

```

```

        for (byte j = 0; j < possibleMoves.length() / 4; j++) {
            Location newLoc = new
Location(possibleMoves.substring(j*4, j*4 + 4));
            allPossibleMoves.add(new Move(new
Location(marblesArr[i].getLoc()),
                                newLoc));
        }
    }
    return allPossibleMoves;
}

public String allNextMovesStr(Location loc) {
    Set<String> possiblePosLst = new HashSet<>();
    String allPossibleMoves = "";
    String currMarbleStrID = loc.getLocStrID();

    // Add the given cell so we later remove it
    possiblePosLst.add(currMarbleStrID);

    // Go all over the possibilities of the given cell
    goAllOverPossibilities(loc, possiblePosLst, true);
    // Remove the given cell from the list
    possiblePosLst.remove(currMarbleStrID);

    // Go all over the returned possible positions
    for(String tempStr : possiblePosLst) {
        allPossibleMoves += tempStr;
    }

    return allPossibleMoves;
}

private void goAllOverPossibilities(Location loc,
                                    Set<String> possiblePosLst,
                                    boolean checkForComplexMove)
{
    for (int i = 0; i < Board.dirArr.length; i++) {
        goOverOneSidePossibilities(loc,
                                    Board.dirArr[i][0],
                                    Board.dirArr[i][1],
                                    possiblePosLst,
                                    checkForComplexMove);
    }
}

private void goOverOneSidePossibilities(Location loc,
                                        byte rowDif, byte colDif,
possiblePosLst,
                                        boolean
checkForComplexMove) {

    Location currLoc = new Location((byte) (loc.getRow() +
rowDif),
                                    (byte) (loc.getCol() +
colDif));
    Location nextLoc = new Location((byte) (loc.getRow() +
(rowDif*2)),

```

```

        (byte)(loc.getCol() +
(colDif*2)));

        String currStr = currLoc.getLocStrID();
        String nextStr = nextLoc.getLocStrID();

        // Check if the cell exists on the board and wasn't already
checked
        if (Board.isCellExists(currLoc)
            && !possiblePosLst.contains(currStr)) {
            // Check if there is a marble on the cell
            if (isThereMarble(currLoc)) {
                // Check if the next cell is in the board borders,
                // AND there isn't a marble on the next cell
                // AND it wasn't already checked
                if (Board.isCellExists(nextLoc) &&
                    !isThereMarble(nextLoc) &&
                    !possiblePosLst.contains(nextStr)) {

                    // If the second top-left is clear, do a
recurrision with
                    // the cell due to complex move
                    possiblePosLst.add(nextStr);
                    goAllOverPossibilities(nextLoc,
                                            possiblePosLst,
                                            false);
                }
            }
            // Otherwise move the empty space to the moves options
            else if (checkForComplexMove){
                possiblePosLst.add(currStr);
            }
        }
    }

    /**
     * The function returns true if given location exists in the game
board,
     * otherwise returns false.
     * @param loc the given location
     * @return true if given location exists in the game board,
     *         otherwise returns false.
     */
    public static boolean isCellExists(Location loc) {
        byte row = loc.getRow();
        byte col = loc.getCol();

        if (row < 0 || col < 0)
            return false;
        if (col >= COLS || row >= ROWS)
            return false;
        if (row*COLS + col > CELLS_COUNT)
            return false;
        return (((boardMask[(row*COLS + col) / 8]) << (row*COLS +
col) % 8) & 0x80) != 0);
    }

    /**
     * This method gets marble that is represented by row and col

```



```
    * and checks if the marble belongs to one of the players in the
    lobby.
    * @param loc the location of the marble
    * @return the method returns true if the marble belongs to one of
    the
    *         players in the lobby, otherwise returns false.
    */
    public boolean isThereMarble(Location loc) {
        // Flag to continue the method as long as we didn't find a
        marble
        boolean isThereMarble = false;
        // Go all over the players
        for (byte i = 0; !isThereMarble && i <
this.playersArr.length; i++) {
            // Check if the marble belong to the player
            isThereMarble =
this.playersArr[i].isMarbleBelongsToPlayer(loc);
        }

        return isThereMarble;
    }
}
```

:Player מחלקת

```

package ChineseCheckersPackage;

public class Player {
    // currMarblesPos have the marbles of the player
    private final Marble[] currMarblesPos;
    // startingMarblesPos is const and contains the starting
    locations of the
    // player's marbles
    private final Location[] startingMarblesPos;
    // The color of the player
    private final char color;

    /**
     * The method builds new Player object
     * @param startingLoc starting coordinates of marbles
     * @param color the color of the player
     */
    public Player(byte[] startingLoc, char color) {
        this.currMarblesPos = new Marble[10];
        this.startingMarblesPos = new Location[10];
        this.color = color;

        for(byte i = 0; i < startingLoc.length; i+=2)
        {
            this.currMarblesPos[i/2] = new Marble(startingLoc[i],
startingLoc[i+1]);
            this.startingMarblesPos[i/2] = new
Location(startingLoc[i], startingLoc[i+1]);
        }
    }

    /**
     * Copy-Constructor.
     * @param other the other player.
     */
    public Player(Player other) {
        this.currMarblesPos = new Marble[10];
        this.startingMarblesPos = new Location[10];
        this.color = other.getColor();

        for (byte i = 0; i < this.currMarblesPos.length; i++) {
            this.currMarblesPos[i] = new
Marble(other.getCurrMarblesPos()[i]);
            this.startingMarblesPos[i] = new
Location(other.getStartingMarblesPos()[i]);
        }
    }

    public Marble[] getCurrMarblesPos() {
        return this.currMarblesPos;
    }

    public Location[] getStartingMarblesPos() {
        return this.startingMarblesPos;
    }

    public char getColor() {
        return this.color;
    }
}

```

```

    }

    public boolean didPlayerStayAtHome() {
        boolean didPlayerStayAtHome = false;
        for (byte i = 0; !didPlayerStayAtHome && i <
currMarblesPos.length; i++) {
            didPlayerStayAtHome =
this.currMarblesPos[i].getIsOnWinningSpot();
        }
        return didPlayerStayAtHome;
    }

    public boolean didPlayerWin() {
        boolean didPlayerWinFlag = true;
        for (byte i = 0; didPlayerWinFlag && i <
currMarblesPos.length; i++) {
            didPlayerWinFlag =
this.currMarblesPos[i].getIsOnWinningSpot();
        }
        return didPlayerWinFlag;
    }

    public void updateMove(Move move) {
        updateMove(move.getStartingLoc(), move.getEndingLoc());
    }

    public void updateMove(Location oldLoc,
                            Location newLoc) {
        boolean isUpdated = false;
        for (byte i = 0; !isUpdated && i <
this.currMarblesPos.length; i++) {
            if (this.currMarblesPos[i].getRow() == oldLoc.getRow() &&
this.currMarblesPos[i].getCol() == oldLoc.getCol()) {
                this.currMarblesPos[i].setMarbleLocation(newLoc);
                updateMarbleInfoPos(this.currMarblesPos[i]);
                isUpdated = true;
            }
        }
    }

    public void updateMarbleInfoPos(Marble m) {
        boolean isUpdated = false;

        m.setIsOnStartingSpot(false);
        for (byte i = 0; !isUpdated && i <
this.startingMarblesPos.length; i++) {
            Location tempLoc = this.startingMarblesPos[i];
            if (m.getRow() == tempLoc.getRow() && m.getCol() ==
tempLoc.getCol()) {
                m.setIsOnStartingSpot(true);
                isUpdated = true;
            }
        }

        if(!isUpdated) {
            m.setIsOnWinningSpot(false);
            for (byte i = 0; !isUpdated && i <
this.startingMarblesPos.length; i++) {
                Location tempLoc = this.startingMarblesPos[i];

```

```
        if ( (m.getRow() == (Board.ROWS - tempLoc.getRow() -
1)) &&
            (m.getCol() == (Board.COLS - tempLoc.getCol() -
1)) ) {
            m.setIsOnWinningSpot(true);
            isUpdated = true;
        }
    }
}

public boolean isMarbleBelongsToPlayer(Location loc) {
    boolean isBelong = false;
    for (byte i = 0; !isBelong && i < this.currMarblesPos.length;
i++) {
        if (this.currMarblesPos[i].getRow() == loc.getRow() &&
            this.currMarblesPos[i].getCol() == loc.getCol()) {
            isBelong = true;
        }
    }
    return isBelong;
}
```

:Marble מחלקת

```

package ChineseCheckersPackage;

public class Marble {
    // The location of the Marble (contains row and col)
    private final Location loc;
    // Flag to indicate if this marble is on the starting spot of the
    player
    // it belongs to
    private boolean isOnStartingSpot;
    // Flag to indicate if this marble is on the winning spot of the
    player
    // it belongs to
    private boolean isOnWinningSpot;

    public Marble(byte row, byte col) {
        this.loc = new Location(row, col);
        this.isOnStartingSpot = true;
        this.isOnWinningSpot = false;
    }

    public Marble(Marble other) {
        this.loc = new Location(other.getLoc());
    }

    public Location getLoc() {
        return this.loc;
    }
    public boolean getIsOnStartingSpot() {
        return this.isOnStartingSpot;
    }
    public boolean getIsOnWinningSpot() {
        return this.isOnWinningSpot;
    }

    public byte getRow() {
        return this.loc.getRow();
    }
    public byte getCol() {
        return this.loc.getCol();
    }
    public void setRow(byte row) {
        this.loc.setRow(row);
    }
    public void setCol(byte col) {
        this.loc.setCol(col);
    }
    public void setMarbleLocation(byte row, byte col) {
        this.setRow(row);
        this.setCol(col);
    }
    public void setMarbleLocation(Location other) {
        this.loc.setRow(other.getRow());
        this.loc.setCol(other.getCol());
    }

    public void setIsOnStartingSpot(boolean isOnStartingSpot) {
        this.isOnStartingSpot = isOnStartingSpot;
    }
}

```

```

    public void setIsOnWinningSpot(boolean isOnWinningSpot) {
        this.isOnWinningSpot = isOnWinningSpot;
    }
}

```

:Location מחלקת

```

package ChineseCheckersPackage;

public class Location {
    private byte row;
    private byte col;

    public Location(String locationID) {
        this.row = (byte)((locationID.charAt(0) - '0') * 10 +
            locationID.charAt(1) - '0');
        this.col = (byte)((locationID.charAt(2) - '0') * 10 +
            locationID.charAt(3) - '0');
    }

    public Location(byte row, byte col) {
        this.row = row;
        this.col = col;
    }

    public Location(Location other) {
        this.row = other.getRow();
        this.col = other.getCol();
    }

    public byte getRow() {
        return this.row;
    }
    public byte getCol() {
        return this.col;
    }
    public void setRow(byte row) {
        this.row = row;
    }
    public void setCol(byte col) {
        this.col = col;
    }

    /**
     * This method gets marble that is represented by row and col
     * and convert the coordinates to string ID of the marble.
     * @param row the row of the marble
     * @param col the col of the marble
     * @return string ID of the marble.
     */
    public String getLocStrID() {
        String locId;

        if (this.row < 10)
            locId = '0' + Byte.toString(this.row);
        else
            locId = Byte.toString(this.row);

        if (this.col < 10)
            locId += '0' + Byte.toString(this.col);
        else

```

```

        locId += Byte.toString(this.col);

        return locId;
    }
}

```

:LocalLobby מחלקת

```

package ChineseCheckersPackage;

public class LocalLobby extends Lobby {

    public LocalLobby(String playersColors) {
        super((byte) (playersColors.length()));
        super.addColorsToGame(playersColors);
    }
}

```

:OnlineLobby מחלקת

```

package ChineseCheckersPackage;

import javax.websocket.Session;

public class OnlineLobby extends Lobby {
    private final Session[] sessions;
    private final String lobName;
    private final String playersColorsStr;
    private byte currentPlayersCount;

    /**
     * The method is constructor that builds new OnlineLobby object
     * @param playersColors String that each char represents
different color
     * @param lobName      The name of the lobby
     */
    public OnlineLobby(String playersColors,
                        String lobName) {
        super((byte) playersColors.length());
        super.addColorsToGame(playersColors);
        this.lobName = lobName;
        this.playersColorsStr = playersColors;
        this.sessions = new Session[playersColors.length()];
        this.currentPlayersCount = 0;
        this.currGameTurn = -1;
    }

    /**
     * The method checks if given player clicked on marble on his
turn
     * @param s    The given session
     * @param loc The name of the lobby
     * @return true if it the player turn otherwise returns false
     */
    public boolean isSessionTurn(Session s, Location loc) {
        byte savedIndex = 0;
        for (byte i = 0; i < this.sessions.length; i++) {
            if (sessions[i] == s)
                savedIndex = i;
        }
    }
}

```

```

        return
    this.gameBoard.getPlayer(savedIndex).isMarbleBelongsToPlayer(loc);
    }

    public Session[] getSessions() {
        return this.sessions;
    }
    public String getLobName() {
        return this.lobName;
    }
    public byte getCurrentPlayersCount() {
        return this.currentPlayersCount;
    }
    public String getPlayersColors() {
        return this.playersColorsStr;
    }
    }

    /**
     * The method adds a given player to the game and checks if the
     game can
     * start after he was added
     * @param s    The given session
     * @return     returns true if after adding the player the game
     can begin,
     *             otherwise returns false
     */
    public boolean addPlayerToOnlineLobby(Session s) {
        boolean isAdded = false;
        for (byte i = 0; !isAdded && i < this.sessions.length; i++) {
            if (this.sessions[i] == null) {
                this.sessions[i] = s;
                isAdded = true;
                this.currentPlayersCount++;
            }
        }
        super.updateGameDetailsOfLobby();

        if (this.currentPlayersCount == this.sessions.length) {
            return true;
        }
        return false;
    }

    /**
     * The method removes given player from the lobby and checks if
     the game
     * need to be canceled because he left the lobby
     * start after he was added
     * @param s    The given session
     * @return     returns true if the game should close due to the
     player
     *             leaving the game, otherwise returns false
     */
    public boolean removeSessionFromSessionsArr(Session s) {
        if (this.currGameTurn == -1) {
            // Decrease the ammount of players in lobby by 1
            this.currentPlayersCount--;
            this.currPlayerIndex--;

            // Check if there are players in the room
            if (this.currentPlayersCount == 0) {

```



```

        return true;
    }
    else {
        removeSessionSafely(s);
        return false;
    }
}
else
    return true;
}

/**
 * The method removes given player from the lobby.
 * @param s    The given session
 */
private void removeSessionSafely(Session s) {
    boolean isRemoved = false;
    for (byte i = 0; !isRemoved && i < this.sessions.length; i++)
    {
        if (this.sessions[i] == s) {
            this.sessions[i] = null;
            isRemoved = true;
        }
    }
}
}

```

:BotLobby מחלקת

```

package ChineseCheckersPackage;

import java.util.Random;

public class BotLobby extends Lobby {
    // BotType array to represents different bots with different
    logics
    private final BotType[] botT;

    public BotLobby(String greenPlayerStr, String redPlayerStr) {
        super((byte)2);
        super.addPlayer(super.createGreenPlayer());
        super.addPlayer(super.createRedPlayer());
        this.botT = new BotType[2];

        addBot(greenPlayerStr, (byte)0);
        addBot(redPlayerStr, (byte)1);
    }

    private void addBot(String playerStr, byte index) {
        byte randomOpeningMove = (byte)((new Random()).nextInt(2));
        switch (playerStr) {
            case "alphaBeta":
                this.botT[index] = new AlphaBetaBot(this,
randomOpeningMove,
                                                                    index,
                                                                    false);
                break;
            case "alphaBetaHard":
                this.botT[index] = new AlphaBetaBot(this,

```

```

randomOpeningMove,

                                                                    index,
                                                                    true);
        break;
    case "monteCarlo":
        this.botT[index] = new MCTS(this,
                                    randomOpeningMove);
        break;
    default:
        this.botT[index] = null;
        break;
    }
}

public String botMove() {
    Move botNextMove = this.botT[this.currPlayerIndex].botMove();
    updateMoveOfPlayer(botNextMove);
    return botNextMove.getMoveID();
}

public boolean didPlayerWin(byte playerIndex) {
    return this.gameBoard.getPlayer(playerIndex).didPlayerWin();
}

public void updateMoveOfPlayer(Move move) {
    this.gameBoard.getPlayer(this.currPlayerIndex).updateMove(move);
}

public void updateMoveOfPlayer(Location oldLoc,
                                Location newLoc,
                                byte playerIndex) {
    this.gameBoard.getPlayer(playerIndex).updateMove(oldLoc,
                                                        newLoc);
}
}

```

מחלקת BotType

```

package ChineseCheckersPackage;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public abstract class BotType {
    public final static int TIMEOUT_MILLISECONDS = 1000;
    protected BotLobby botLob;
    protected byte randomOpeningMove;

    public abstract Move botMove();

    private boolean isNewLocationMakesSense(byte oldRow,
                                             byte newRow, byte newCol,
                                             char playerColor,
                                             boolean isEndingGame) {
        boolean isLocationOnMiddleAndAdvance = false;

        if (playerColor == 'g') {
            if (newRow > oldRow || (isEndingGame && newRow == oldRow
&& newRow >= 12)) {
                if (checkIfMarbleOnMiddleBoard(newRow, newCol) ||
                    checkIfMarbleOnStartOrTarget(newRow))
                    isLocationOnMiddleAndAdvance = true;
            }
        }
        else if (playerColor == 'r') {
            if (newRow < oldRow || (isEndingGame && newRow == oldRow
&& newRow <= 4)) {
                if (checkIfMarbleOnMiddleBoard(newRow, newCol) ||
                    checkIfMarbleOnStartOrTarget(newRow))
                    isLocationOnMiddleAndAdvance = true;
            }
        }
        return isLocationOnMiddleAndAdvance;
    }

    public byte getAllPossibleMoves(Player p,
                                    Map<Marble, List<Location>>
allPossibleMoves,
                                    boolean
shouldLookForLogicalMoves,
                                    boolean isEndGame) {
        Marble[] marblesArr = p.getCurrMarblesPos();

        byte movesCount = 0;
        for (byte i = 0; i < marblesArr.length; i++) {
            List<Location> tempPosList = new ArrayList<>();
            allPossibleMoves.put(marblesArr[i], tempPosList);
            String possibleMoves =
botLob.getGameBoard().allNextMovesStr(marblesArr[i].getLoc());

            for (byte j = 0; j < possibleMoves.length() / 4; j++) {
                Location tempLoc = new
Location(possibleMoves.substring(j*4, j*4 + 4));
                if (shouldLookForLogicalMoves) {
                    if
(isNewLocationMakesSense(marblesArr[i].getRow(),

```

```

tempLoc.getCol(),
tempLoc.getRow(),
p.getColor(),
isEndGame)) {
    tempPosList.add(tempLoc);
    movesCount++;
}
}
else {
    tempPosList.add(tempLoc);
    movesCount++;
}
}
return movesCount;
}

private boolean checkIfMarbleOnMiddleBoard(byte newRow, byte
newCol) {
    boolean isOnMiddleBoard = false;

    byte [][] matLocs = {
        {4, 12, 17, 7},
        {5, 11, 18, 6},
        {6, 10, 19, 5},
        {7, 9, 20, 4}
    };

    if (newRow == 8)
        isOnMiddleBoard = true;

    for (byte i = 0; !isOnMiddleBoard && i < 4; i++) {
        if ((newRow == matLocs[i][0] || newRow == matLocs[i][1])
&&
            (newCol < matLocs[i][2] && newCol > matLocs[i][3]))
            isOnMiddleBoard = true;
    }

    return isOnMiddleBoard;
}

private boolean checkIfMarbleOnGreenHouse(byte newRow) {
    return newRow == 0 || newRow == 1 || newRow == 2 || newRow ==
3;
}
private boolean checkIfMarbleOnRedHouse(byte newRow) {
    return newRow == 13 || newRow == 14 || newRow == 15 || newRow
== 16;
}

private boolean checkIfMarbleOnStartOrTarget(byte newRow) {
    return checkIfMarbleOnGreenHouse(newRow) ||
        checkIfMarbleOnRedHouse(newRow);
}

protected Move makeGreenKnownOpeningMove() {
    Move [][] knownOpeningMoves = {
        {
            new Move(new byte[]{3, 9, 4, 10}),
            new Move(new byte[]{3, 15, 4, 14}),

```

```

        new Move(new byte[] {1, 11, 5, 11}),
        new Move(new byte[] {1, 13, 5, 13}),
        new Move(new byte[] {0, 12, 1, 11})
    },
    {
        new Move(new byte[] {3, 15, 4, 14}),
        new Move(new byte[] {3, 9, 4, 10}),
        new Move(new byte[] {1, 13, 5, 13}),
        new Move(new byte[] {1, 11, 5, 11}),
        new Move(new byte[] {0, 12, 1, 13})
    }
};

return
knownOpeningMoves[this.randomOpeningMove][botLob.currGameTurn];
}

protected Move makeRedKnownOpenningMove() {
    Move[][] knownOpeningMoves = {
        {
            new Move(new byte[] {13, 9, 12, 10}),
            new Move(new byte[] {13, 15, 12, 14}),
            new Move(new byte[] {15, 11, 11, 11}),
            new Move(new byte[] {15, 13, 11, 13}),
            new Move(new byte[] {16, 12, 15, 11})
        },
        {
            new Move(new byte[] {13, 15, 12, 14}),
            new Move(new byte[] {13, 9, 12, 10}),
            new Move(new byte[] {15, 13, 11, 13}),
            new Move(new byte[] {15, 11, 11, 11}),
            new Move(new byte[] {16, 12, 15, 13})
        }
    };

    return
    knownOpeningMoves[this.randomOpeningMove][botLob.currGameTurn];
}
}

```

:AlphaBetaBot מחלקת

```

package ChineseCheckersPackage;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class AlphaBetaBot extends BotType {
    //
    =====
    // Consts:
    //
    =====
    private final static short MAX = Short.MAX_VALUE;
    private final static short MIN = Short.MIN_VALUE;
    private final static byte INITIAL_DEPTH = 3;

    //
    =====
    // References and indexers to the players:
    //
    =====
    private final Player botP;
    private final byte botIndex;
    private final Player enemyP;
    private final byte enemyIndex;

    //
    =====
    // Best move saved information:
    //
    =====
    private Move globalBestMove;
    private final Move bestMove;
    private byte currentDepth;
    private long start;
    private boolean timeout;

    //
    =====
    // Is on hard mode
    //
    =====
    private final boolean isHardMode;

    public AlphaBetaBot(BotLobby botLob,
                        byte randomOpening,
                        byte botIndex,
                        boolean isHardMode) {
        this.botLob = botLob;
        this.randomOpeningMove = randomOpening;

        this.botIndex = botIndex;
        this.enemyIndex = (byte)(1 - botIndex);

        this.botP =
this.botLob.getGameBoard().getPlayer(this.botIndex);

```

```

        this.enemyP =
this.botLob.getGameBoard().getPlayer(this.enemyIndex);

        this.bestMove = new Move();

        this.isHardMode = isHardMode;
    }

    @Override
    public Move botMove() {
        if (this.botLob.currGameTurn < 5) {
            if (this.botP.getColor() == 'g')
                return makeGreenKnownOpeningMove();
            else
                return makeRedKnownOpeningMove();
        }
        else {
            return findBestGlobalMove();
        }
    }

    // Returns optimal value for current player
    public short minimax(byte depth,
                        Boolean maximizingPlayer,
                        short alpha, short beta) {
        if (this.globalBestMove != null &&
            System.currentTimeMillis() - this.start >
TIMEOUT_MILLISECONDS)
        {
            this.timeout = true;
            return MIN;
        }

        if(this.botLob.didPlayerWin(this.botIndex))
            return MAX;

        // Terminating condition
        if (depth == 0)
            return evaluate();

        if (maximizingPlayer) {

            // Get all possible moves that make sense
            Map<Marble, List<Location>> allPossMoves = new
HashMap<>();
            if(getAllPossibleMoves(botP, allPossMoves, true, false)
== 0)
                if(getAllPossibleMoves(botP, allPossMoves, true,
true) == 0)
                    getAllPossibleMoves(botP, allPossMoves, false,
false);

            Marble[] botMarbles = botP.getCurrMarblesPos();

            // Traverse all cells, evaluate minimax function for all
empty cells
            // and return the cell with optimal value.
            for (byte i = 0; i < botMarbles.length; i++) {
                // Saved original location

```

```

        Location oldLoc = new
Location(botMarbles[i].getLoc());

        // New locations list
        List<Location> newMarbleLocs =
allPossMoves.get(botMarbles[i]);
        // Go all over the new locations
        for (Location newLoc : newMarbleLocs) {

            // Make new move
            this.botLob.updateMoveOfPlayer(oldLoc,
                                           newLoc,
                                           this.botIndex);

            // compute evaluation function for this move.
            short moveVal = minimax((byte)(depth - 1),
                                   !maximizingPlayer,
                                   alpha, beta);

            // Undo the move
            this.botLob.updateMoveOfPlayer(newLoc,
                                           oldLoc,
                                           this.botIndex);

            if (moveVal > alpha) {
                alpha = moveVal;

                if (depth == this.currentDepth) {
                    bestMove.setMove(oldLoc, newLoc);
                }
            }
            else if (depth == this.currentDepth && moveVal ==
alpha) {
                if
(isGivenDistanceBetterThanBestDistance(this.botP.getColor(),
oldLoc.getRow(),
newLoc.getRow())) {
                    bestMove.setMove(oldLoc, newLoc);
                }
            }

            // Alpha Beta Pruning
            if (beta <= alpha)
                return alpha;
        }
        return alpha;
    }
    else {
        // Get all possible moves that make sense
        Map<Marble, List<Location>> allPossMoves = new
HashMap<>();
        if(getAllPossibleMoves(enemyP, allPossMoves, true, false)
== 0)
            if(getAllPossibleMoves(enemyP, allPossMoves, true,
true) == 0)
                getAllPossibleMoves(enemyP, allPossMoves, false,
false);
    }

```



```

        Marble[] enemyMarbles = enemyP.getCurrMarblesPos();

        // Traverse all cells, evaluate minimax function for all
empty cells
        // and return the cell with optimal value.
        for (byte i = 0; i < enemyMarbles.length; i++) {
            // Saved original location
            Location oldLoc = new
Location(enemyMarbles[i].getLoc());

            // New locations list
            List<Location> newMarbleLocs =
allPossMoves.get(enemyMarbles[i]);
            // Go all over the new locations
            for (Location newLoc : newMarbleLocs) {
                // Make new move
                this.botLob.updateMoveOfPlayer(oldLoc,
                                                newLoc,
                                                this.enemyIndex);

                // compute evaluation function for this move.
                short moveVal = minimax((byte)(depth - 1),
                                        !maximizingPlayer,
                                        alpha, beta);

                // Undo the move
                this.botLob.updateMoveOfPlayer(newLoc,
                                                oldLoc,
                                                this.enemyIndex);

                // Alpha Beta Pruning
                if (moveVal <= beta) {
                    beta = moveVal;
                }
                if (alpha >= beta) {
                    return beta;
                }
            }
        }
        return beta;
    }
}

// This will return the best global possible move for the player
public Move findBestGlobalMove() {
    this.timeout = false;
    this.start = System.currentTimeMillis();

    for (byte d = 0;; d++)
    {
        if (d == 0) {
            this.globalBestMove = null;
        }
        else {
            //this.globalBestMove = new Move(this.bestMove);
            this.globalBestMove = this.bestMove;
            System.out.println("Completed search with depth " +
this.currentDepth);
            System.out.println("time: " +
(System.currentTimeMillis() - this.start));

```

```

    }
    this.currentDepth = (byte) (INITIAL_DEPTH + d);
    if (minimax(this.currentDepth, true, MIN, MAX) == MAX)
        return this.bestMove;
    if (this.timeout)
    {
        System.out.println("timeout!");
        System.out.println("bestMove: " +
this.bestMove.getMoveID());
        System.out.println("globalBestMove: " +
this.globalBestMove.getMoveID());
        return globalBestMove;
    }
}

private short evaluate() {
    if (this.botP.getColor() == 'g') {
        return (short) (evaluateGreen(botP, enemyP) -
evaluateRed(enemyP, botP));
    }
    else {
        return (short) (evaluateRed(botP, enemyP) -
evaluateGreen(enemyP, botP));
    }
}

public boolean isGivenDistanceBetterThanBestDistance(char pColor,
byte oldRow,
byte
newRow) {
    if (this.botP.getColor() == 'g') {
        if (newRow - oldRow >
            bestMove.getEndingLoc().getRow() -
bestMove.getStartingLoc().getRow() &&
            oldRow < bestMove.getStartingLoc().getRow())
            return true;
    }
    else if (this.botP.getColor() == 'r') {
        if (oldRow - newRow >
            bestMove.getStartingLoc().getRow() -
bestMove.getEndingLoc().getRow() &&
            oldRow > bestMove.getStartingLoc().getRow())
            return true;
    }
    return false;
}

private short evaluateGreen(Player greenP, Player redP) {
    short score = 0;
    Marble[] marblesArr = greenP.getCurrMarblesPos();

    for (byte i = 0; i < marblesArr.length; i++) {
        score += marblesArr[i].getRow();
        if (checkIfMarbleAlone(marblesArr[i], greenP))
            score -= 10;
        if (marblesArr[i].getIsOnStartingSpot())
            score -= 30;

        if (this.botLob.currGameTurn < 40 && this.isHardMode) {

```

```

        score += checkShapesForGivenMarble(marblesArr[i],
                                           greenP, redP,
                                           (byte)1);
    }
}

return score;
}

private short evaluateRed(Player redP, Player greenP) {
    short score = 0;
    Marble[] marblesArr = redP.getCurrMarblesPos();

    for (byte i = 0; i < marblesArr.length; i++) {
        score += Board.ROWS - marblesArr[i].getRow() - 1;
        if (checkIfMarbleAlone(marblesArr[i], redP))
            score -= 10;
        if (marblesArr[i].getIsOnStartingSpot())
            score -= 30;

        if (this.botLob.currGameTurn < 40 && this.isHardMode) {
            score += checkShapesForGivenMarble(marblesArr[i],
                                                redP, greenP,
                                                (byte)-1);
        }
    }

    return score;
}

private boolean checkIfMarbleAlone(Marble m, Player p) {
    boolean isMarbleAlone = true;

    // Check if the marble is alone ONLY on the side color side
    of the
    // board
    /*if (p.getColor() == 'g' && m.getRow() > 8)
        isMarbleAlone = false;
    else if (p.getColor() == 'r' && m.getRow() < 8)
        isMarbleAlone = false;
    */
    for (int i = 0; isMarbleAlone && i < Board.dirArr.length;
i++) {
        Location newLoc = new Location((byte) (m.getRow() +
Board.dirArr[i][0]),
                                     (byte) (m.getCol() +
Board.dirArr[i][1]));
        if (p.isMarbleBelongsToPlayer(newLoc))
            isMarbleAlone = false;
    }
    return isMarbleAlone;
}

private short checkShapesForGivenMarble(Marble m,
                                         Player p, Player enemyP,
                                         byte
checkingFromBottomToUpper) {
    return (short) (checkPairAndTriple(m, p,
checkingFromBottomToUpper) +

```

```

        checkQuad(m, p, enemyP,
checkingFromButtomToUpper));
    }

    // Check the following shapes:
    // --*----
    // ---*---
    // OR:
    // -----*--
    // ---*---
    // Also add bonus if both exists (meaning):
    // --*-*---
    // ---*---
    private short checkPairAndTriple(Marble m, Player p,
        byte checkingFromButtomToUpper)
    {
        short score = 0;

        Location tempLocation;
        boolean isRightPair = false;

        tempLocation = new Location((byte) (m.getRow() + (1 *
checkingFromButtomToUpper)),
        (byte) (m.getCol() + 1));

        if (Board.isCellExists(tempLocation) &&
            p.isMarbleBelongsToPlayer(tempLocation)) {
            tempLocation.setRow((byte) (m.getRow() + (2 *
checkingFromButtomToUpper)));
            tempLocation.setCol((byte) (m.getCol() + 2));

            if (Board.isCellExists(tempLocation) &&
!this.botLob.getGameBoard().isThereMarble(tempLocation)) {
                score++;
                isRightPair = true;
            }
        }

        // Check left
        tempLocation.setRow((byte) (m.getRow() + (1 *
checkingFromButtomToUpper)));
        tempLocation.setCol((byte) (m.getCol() - 1));

        if (Board.isCellExists(tempLocation) &&
            p.isMarbleBelongsToPlayer(tempLocation)) {
            tempLocation.setRow((byte) (m.getRow() + (1 *
checkingFromButtomToUpper)));
            tempLocation.setCol((byte) (m.getCol() - 2));

            if (Board.isCellExists(tempLocation) &&
!this.botLob.getGameBoard().isThereMarble(tempLocation)) {
                score++;

                // If both pairs exists, add bonus to the score
                // because it is triangle shape
                if (isRightPair)
                    score += 10;
            }
        }
    }
}

```

```

        return score;
    }

    // Check the following shapes:
    // --*---
    // ---*---
    // --*---
    // ---*---
    // OR:
    // ----*--
    // ---*---
    // ----*--
    // ---*---
    private short checkQuad(Marble m, Player p, Player enemyP,
        byte checkingFromBottomToUpper) {
        short score = 0;

        score += checkOneSideQuad(m, p, enemyP,
            checkingFromBottomToUpper,
            (byte)1);
        score += checkOneSideQuad(m, p, enemyP,
            checkingFromBottomToUpper,
            (byte)-1);

        return score;
    }

    private short checkOneSideQuad(Marble m, Player p, Player enemyP,
        byte checkingFromBottomToUpper,
        byte sideDiff) {
        short score = 0;

        Location tempLocation;
        // Check first progression
        tempLocation = new Location((byte)(m.getRow() + (1 *
checkingFromBottomToUpper)),
            (byte)(m.getCol() + (1 *
sideDiff)));

        if (Board.isCellExists(tempLocation) &&
            p.isMarbleBelongsToPlayer(tempLocation)) {
            tempLocation.setRow((byte)(m.getRow() + (2 *
checkingFromBottomToUpper)));
            tempLocation.setCol((byte)(m.getCol()));

            if (Board.isCellExists(tempLocation) &&
                p.isMarbleBelongsToPlayer(tempLocation)) {
                // Check second progression
                tempLocation.setRow((byte)(m.getRow() + (3 *
checkingFromBottomToUpper)));
                tempLocation.setCol((byte)(m.getCol() + (1 *
sideDiff)));

                if (Board.isCellExists(tempLocation) &&
                    p.isMarbleBelongsToPlayer(tempLocation)) {
                    tempLocation.setRow((byte)(m.getRow() + (4 *
checkingFromBottomToUpper)));
                    tempLocation.setCol((byte)(m.getCol()));

                    if (Board.isCellExists(tempLocation) &&

```


:MCTS מחלקת

```

package ChineseCheckersPackage;

public class MCTS extends BotType {
    // IN_PROGRESS defention (for Monte-Carlo)
    public static final byte IN_PROGRESS = -2;

    /**
     * Builds new MCTS Object
     * @param botLob      The given botLobby
     * @param randomOpening The given random opening number it will
execute
    */
    public MCTS(BotLobby botLob,
                byte randomOpening) {
        this.botLob = botLob;
        this.randomOpeningMove = randomOpening;
    }

    // Overrides the bot move for Monte Carlo logic
    @Override
    public Move botMove() {
        if (this.botLob.currGameTurn < 5) {
            if (this.botLob.getCurrColor() == 'g')
                return makeGreenKnownOpenningMove();
            else
                return makeRedKnownOpenningMove();
        }
        else {
            return findNextMove();
        }
    }

    /** The main monte carlo function that plays the game for set
ammount
    * of time and returns the best move it found
    */
    public Move findNextMove()
    {
        byte playOutResult;

        Board board = this.botLob.getGameBoard();
        byte playerNo = this.botLob.getCurrPlayerIndex();

        State boardState = new State(board);
        Node rootNode = new Node(boardState);
        byte opponent = (byte) (1-playerNo);
        //set the player to the opponent
        rootNode.getState().setPlayerNo(opponent);
        // current time
        long startingTime = System.currentTimeMillis();

        while(System.currentTimeMillis() - startingTime < 2000) {
            //stage 1 - selecting
            // promising Node is the leaf node we are going to
explore
            Node promisingNode = selectPromisingNode(rootNode);

            //stage 2 - expand

```

```

        if(promisingNode.getState().getBoard().checkStatus() ==
MCTS.IN_PROGRESS &&
            promisingNode.getChildArray().isEmpty()) {
            promisingNode.expandNode();
        }

        //stage 3 - simulation
        Node NodeToExplore = promisingNode;
        if(!NodeToExplore.getChildArray().isEmpty()) {
            // Node to Explore holds one of the just expanded
nodes
            NodeToExplore = promisingNode.getRandomChildNode();

            playOutResult = simulateRandomPayout(NodeToExplore);
        }
        else {
            playOutResult =
NodeToExplore.getState().getBoard().checkStatus();
        }

        // stage 4 - backpropagation
        // back propage and update all the visited nodes
        backPropagation(NodeToExplore, playOutResult);
    }

    // the winner node is the max value child (the
backpropagation)
    Node winnerNode = rootNode.getChildWithMaxScore();
    return winnerNode.getState().getMove();
}

/**
 * The method gets a root node of a tree and find the best leaf to
explore
 * @param root root node of a tree
 * @return best leaf to explore
 */
private Node selectPromisingNode(Node root)
{
    // this function
    Node node = root;

    while(!node.getChildArray().isEmpty())
    {
        node = UCT.findPromisingNodeUsingUCT(node);
    }
    return node;
}

/**
 * The method gets a leaf node and run a random game from his
state
 * @param node given node
 * @return the board status at the end of the simulation
 */
private byte simulateRandomPayout(Node node) {
    Node tempNode = new Node(node);
    State tempState = tempNode.getState();
    byte boardStatus = tempState.getBoard().checkStatus();
    long startingTime = System.currentTimeMillis();

```



```

        while (System.currentTimeMillis() - startingTime < 100 &&
               boardStatus == MCTS.IN_PROGRESS)
        {
            // move the turn to the other player
            tempState.togglePlayer();
            // play a random move from the current
            tempState.randomPlay();
            // check if the game is over, if it is return the one who
won
            boardStatus = tempState.getBoard().checkStatus();
        }
        return boardStatus;
    }

    /**
     * The method updates the current move sequence with the
simulation result.
     * @param NodeToExplore the given leaf node
     * @param result the result of the game
     */
    private void backPropagation(Node NodeToExplore , int result) {
        Node tempNode = NodeToExplore;
        while(tempNode != null)
        {
            tempNode.getState().incrementVisit();
            tempNode.getState().setWinScore(result *
tempNode.getState().getPlayerNo());
            tempNode = tempNode.getParent();
        }
    }
}

```

מחלקת UCT:

```
package ChineseCheckersPackage;

import java.util.Collections;
import java.util.Comparator;

public class UCT {

    /**
     * The method going over all the children of given node and
     * find out the one with the highest UCT value
     * @param node given node
     * @return The method returns the node with the highest UCT value
     */
    public static Node findPromisingNodeUsingUCT(Node node)

    {
        // the T for the equation
        int parentVisit = node.getState().getVisitCount();

        // going over all the nodes in the child array and find which has
        // the highest UCT value
        return Collections.max(
            node.getChildArray(),
            Comparator.comparing(c -> UCTequ(parentVisit, c.getState().getVisitCount(),
            c.getState().getWinScore())));
    }
}
```

```
public static double UCTequ(int totalVisit , int nodeVisit , double nodeWinScore)
{
    /*wi = number of wins after the i-th move
    ni = number of simulations after the i-th move
    c = exploration parameter (theoretically equal to  $\sqrt{2}$ )
    t = total number of simulations for the parent node
    */
    return (nodeWinScore / (double) nodeVisit) + (1.41 * Math.sqrt(Math.log(totalVisit) /
(double) nodeVisit));
}
}
```

:מחלקת Node

```

package ChineseCheckersPackage;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Node {
    State state;
    Node parent;
    List<Node> childArray;

    public Node(byte playersCount)
    {
        state = new State(playersCount);
        parent = null;
        childArray = new ArrayList<>();
    }

    public Node(Node node)
    {
        this.state = new State(node.state);
        parent = node.parent;
        childArray = node.childArray;
    }

    public Node(State state)
    {
        this.state = new State(state);
        parent = new Node(state.getBoard().getPlayersCount());
        childArray = new ArrayList<>();
    }

    public Node(State state , Node Parent)
    {
        this.state = new State(state);
        this.parent = Parent;
        childArray = new ArrayList<>();
    }

    public State getState() {
        return state;
    }
    public void setState(State state) {
        this.state = new State(state);
    }
    public Node getParent() {
        return parent;
    }
    public void setParent(Node parent) {
        this.parent = parent;
    }
    public List<Node> getChildArray() {
        return childArray;
    }
    public void setChildArray(List<Node> childArray) {
        this.childArray = childArray;
    }
}

```

```

/**
 * @return The method returns a random node from the children
 */
public Node getRandomChildNode()
{
    int selectRandom = (int) (Math.random() *
this.childArray.size());
    return this.childArray.get(selectRandom);
}

/**
 * The method finds the child with max value
 * @return The method returns the child with max value
 */
public Node getChildWithMaxScore() {
    return Collections.max(this.childArray,
Comparator.comparing(c -> {
        return c.getState().getVisitCount();
    }));
}

/**
 * The method is used to a leaf node and add to his children array
all
 * the possible states from this node state
 * available states holds all the possible states from that
 * certain state
 */
public void expandNode()
{
    List<State> availableStates =
this.state.getAllPossibleStates();
    for(int i =0 ; i < availableStates.size(); i++)
    {
        Node newNode = new Node(availableStates.get(i),this);
        this.childArray.add(newNode);
    }
}
}

```

:State מחלקת

```
package ChineseCheckersPackage;

import java.util.ArrayList;
import java.util.List;

public class State {
    private Board board;
    private Move move;
    private byte playerNo;
    private int visitCount;
    private int winScore;

    public State(byte playersCount) {
        this.board = new Board(playersCount);
    }

    public State(State state) {
        this.board = new Board(state.getBoard());
        //this.move = new Move(state.getMove());
        this.playerNo = state.getPlayerNo();
        this.visitCount = state.getVisitCount();
        this.winScore = state.getWinScore();
    }

    public State(Board board) {
        this.board = new Board(board);
        this.winScore = 0;
    }

    public Move getMove() {
        return move;
    }

    public void setMove(Move move) {
        this.move = move;
    }

    public Board getBoard() {
        return board;
    }

    public void setBoard(Board board) {
        this.board = board;
    }

    public byte getPlayerNo() {
        return playerNo;
    }

    public void setPlayerNo(byte playerNo) {
        this.playerNo = playerNo;
    }

    public int getVisitCount() {
        return visitCount;
    }

    public void setVisitCount(int visitCount) {
        this.visitCount = visitCount;
    }
}
```

```

    }

    public int getWinScore() {
        return winScore;
    }

    public void setWinScore(int winScore) {
        this.winScore += winScore;
    }

    /**
     * The method constructs a list of all possible states from
     current state
     * @return list with all the possible states
     */
    public List<State> getAllPossibleStates()
    {
        List<State> possibleStates = new ArrayList<>();
        List<Move> availablePositions =
this.board.getAllPossibleMoves(this.playerNo);
        availablePositions.forEach(p -> {
            State newState = new State(this.board);
            newState.setPlayerNo((byte) (1 - this.playerNo));
            newState.getBoard().performMove(newState.getPlayerNo(),
p);
            newState.setMove(p);
            possibleStates.add(newState);
        });
        return possibleStates;
    }

    /**
     * The method makes a list of all possible positions on the board
     and
     * plays a random move
     */
    public void randomPlay()
    {
        List<Move> availablePositions =
this.board.getAllPossibleMoves(this.playerNo);
        int totalPossibilities = availablePositions.size();
        int selectRandom = (int) (Math.random() *
totalPossibilities);
        this.board.performMove(this.playerNo,
availablePositions.get(selectRandom));
    }

    void incrementVisit() {
        this.visitCount++;
    }

    void togglePlayer() {
        this.playerNo = (byte) (1 - this.playerNo);
    }

    int getOpponent() {
        return 1 - playerNo;
    }
}

```