Onitama

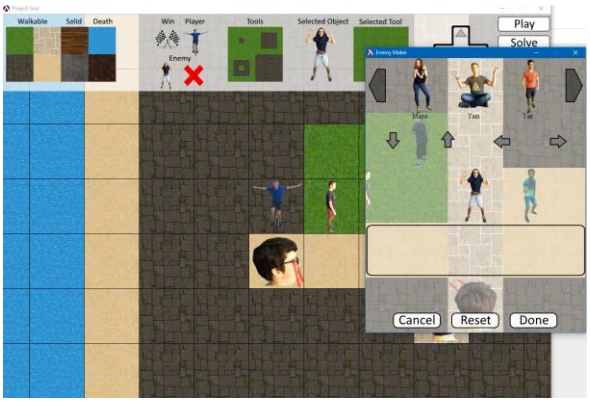
עבודה על ידי שי קליימן

ת"ז - 211641162
בית ספר - ליאו באק
מקצוע - טכם
מורה - אהובה שפרלינג

תוכן עניינים

מבוא

היו לי מספר רעיונות שונים לאורך השנה לגבי איזה סוג פרוייקט אני מעוניין לעשות. תחילה ניסיתי להכין גרסה פשוטה של משחק הקלפים הדיגיטלי, Hearthstone. עבדתי על משחק זה בערך כשבועיים בתחילת השנה, עוד לפני שלמדנו אלגוריתמים של בינה מלאכותית רציניים. עזבתי את רעיון זה במהרה כי הבנתי שמשחק עם אלמנטים אקראיים יעשה מלא בעיות בבינה מלאכותית והחלטתי לעזוב את זה.

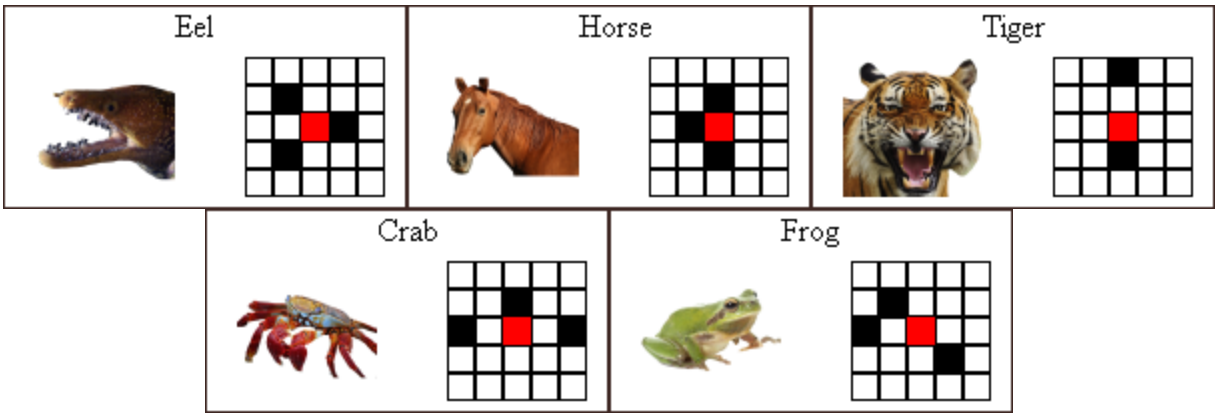


הרעיון השני שניסיתי לפתח השנה היה משהו שקראתי לו תחילה PROJECT GRID.. לא היה לי רעיון ספציפי למשחק אבל רציתי להכין משחק עם תזוזה על רשת של משבצות. הרעיון לזה הגיע מהאלגוריתמים שלמדנו באותה העת. רעיתי את הפוטנצאל של A* במשחק סוג זה וחשבתי שזה יהיה נושא טוב לפרוייקט. עבדתי על פרוייקט זה בערך חודש ובגרסתו הוספית היה 9 אויבים שונים, אלגוריתם A* לפתירת שלבים ואף עורך שלבים ויזואלי. כל הפרוייקט יחד היה מעל 500 שורות וזה היה פרוייקט התכנות הכי גדול שיצרתי עד לנקודה ההיא. לבסוף גם על פרוייקט זה בחרתי לוותר בגלל שלא מצאתי דרך להשתמש באללגוריתמים היותר מסובכים שלמדנו כמו מינימאקס למשחק שיצרתי.

עברו מספר חודשים ונתבקשנו בכיתה לבחור נושא לפרוייקט. למרות שהשקעתי כבר זמן רב, לא היה לי שום רעיון מתאים. לבסוף החלטתי ליצור גרסת ראקט למשחק הלוח אוניטמה.
משחק לוח שכבר תיכננתי בעבר בזמני החופשי כאפליקציית אנדרויד.
פרוייקט זה לא היה בחירתי הראשונה או אף השנייה. אני מרגיש שלו הייתי מתחיל את השנה מחדש כנראה הייתי מספיק לעבוד על משהו מעניין יותר שמתאים לקריטריונים.
אני מקווה שבעבודתי על הפרוייקט אני אצליח ליצור משהו חדש ולא סתם אעשה port לגרסת האנדרויד שלי לשפת תכנות אחרת.

אוניטמה

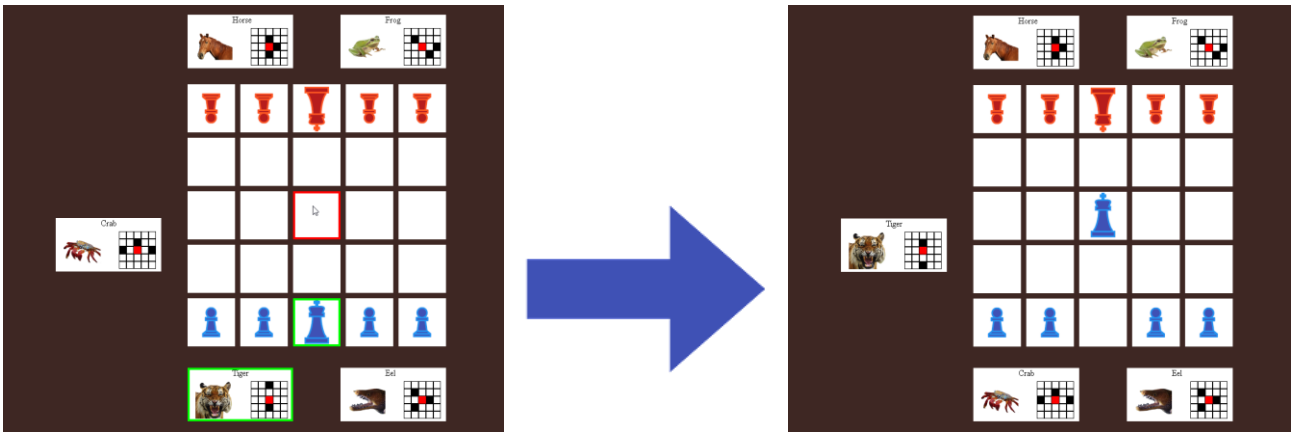
אוניטמה הוא משחק לוח אסטרטגי לשני שחקנים. המשחק דומה במקצת למשחק השחמט אך עם מספר הבדלים משמעותיים. לכל שחקן 5 כלים ודרך התזוזה של הכלים מבוססת על הקלפים שמחזיק השחקן.
כל משחק נבחרים סט של חמישה קלפים. (במשחק המקורי קלפים אלה נבחרים באקראי מערימה של 16 קלפים שונים אך כדי שאלגוריתם הלמידה שלי יעבוד במהירות סבירה השתמשתי בחמישה קלפים קבועים). חמשת הקלפים האלו הם הקלפים היחידים שבהם משתמשים במהלך המשחק. כל שחקן מתחיל עם שני קלפים והקלף החמישי מונח ביניהם.



על כל קלף מסומנים 2-4 מהלכים שונים שהקלף מאפשר לשקחן לעשות. לדוגמה עם קלף הסוס ניתן לזוז צעד אחד שמאלה, קדימה או אחורה. תזוזות אלה הם באופן יחסי לשחקן. משמע כשהקלף ביד של השחקן השני התזוזה מתהפכת.

כל שחקן מתחיל עם 4 חיילים ומלך אחד. אין הבדל בין דרך התזוזה של החיילים לדרך התזוזה של המלך.

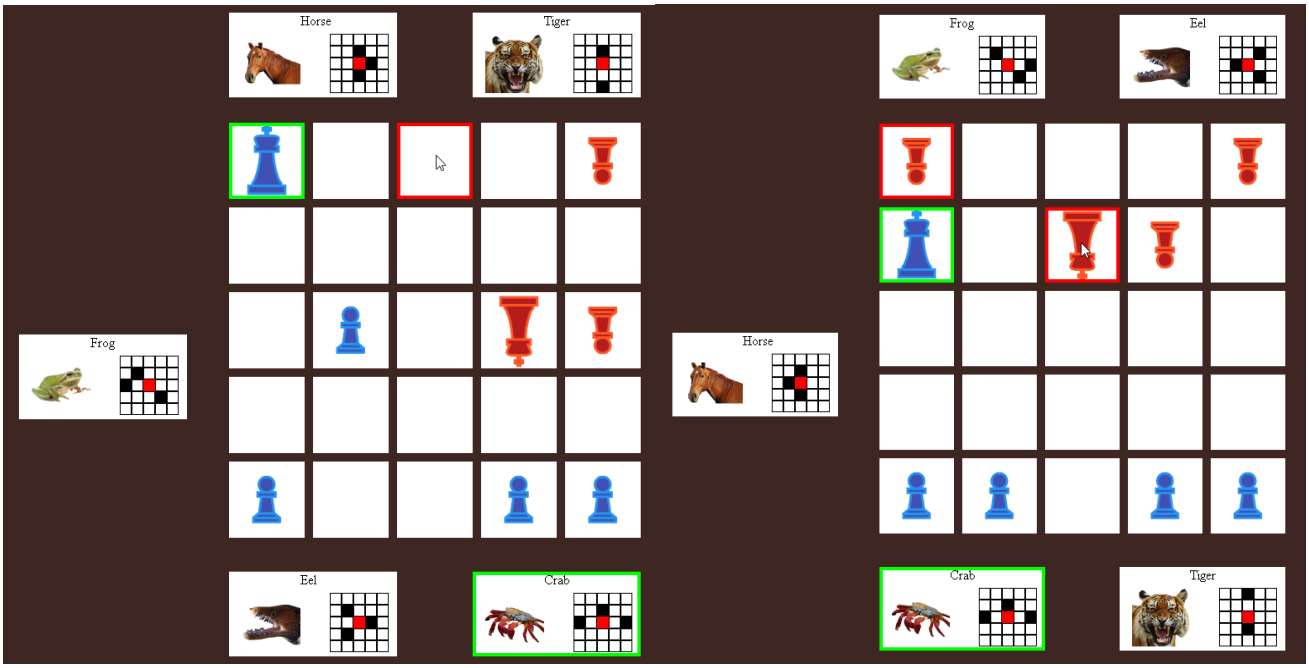
כל תור שחקן בוחר כלי, קלף ומהלך מבין המהלכים שבקלף. בדומה למשחק השחמט, ניתן לזוז למשבצות בהן אין כלים או שיש כלי של היריב אשר אותם "אוכלים". בסוף תור הקלף אשר שומש על ידי השחק מסתובב 180 מעלות ומוחלף עם הקלף החמישי.



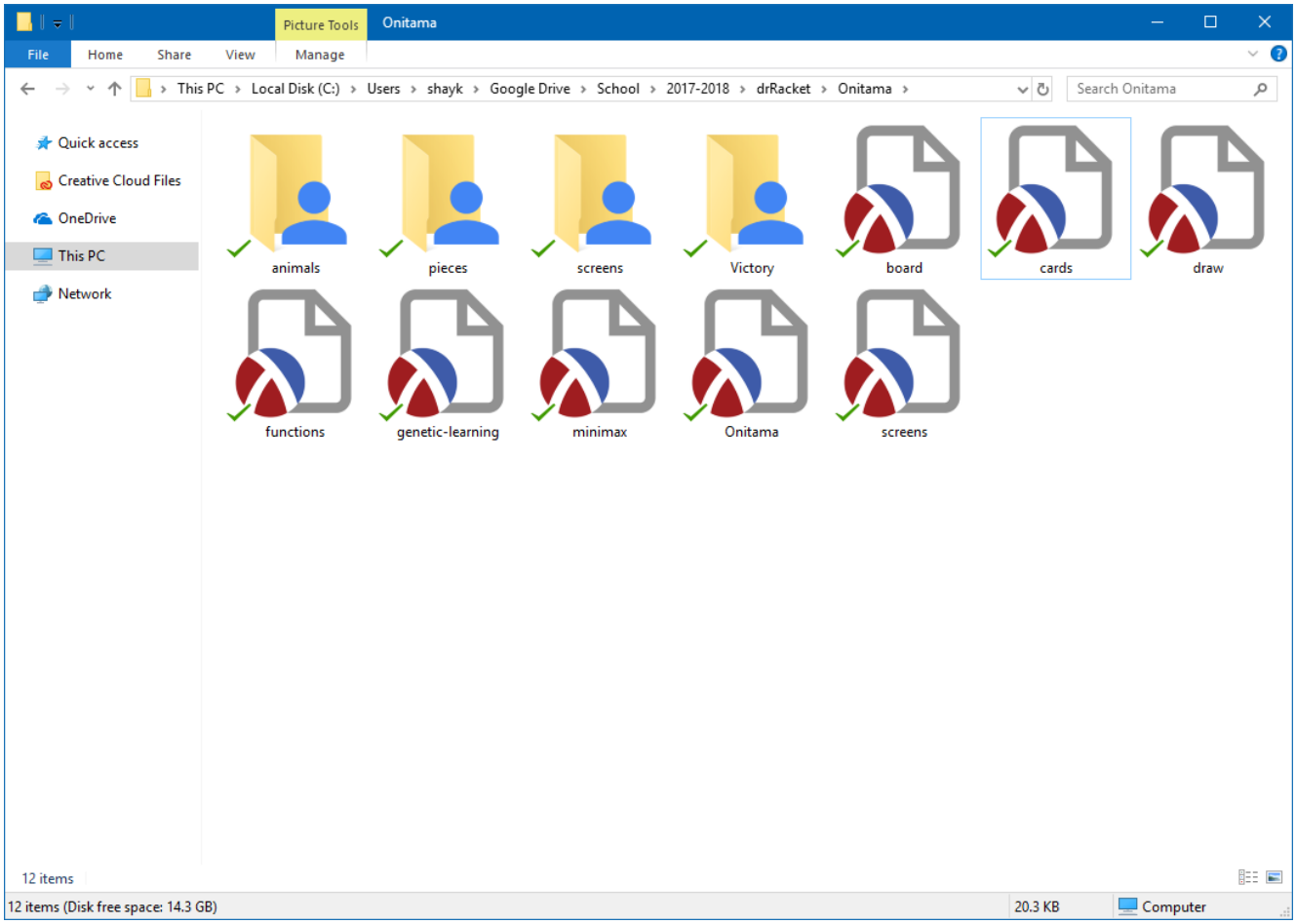
יש שני דרכים לנצח במשחק של אוניטמה

לאכול את המלך של היריב

להביא את המלך לנקודת ההתחלה של המלך של היריב



מבנה הפרוייקט



הפרוייקט מחולק לשמונה קבצים.

- Onitama - הקובץ שמריץ את המשחק ומספר משתנים כללים
- Game - לולאת המשחק העיקרית
- Board - פונקציות שעוסקות בלוח המשחק וכלי המשחק
- Cards - הגדרה של structure הקלף והגדרתם של כל הקלפים
- Functions - פונקציות עזר כלליות
- Draw - פונקציות של ציור המשחק
- Screens - פונקציות של המסכים האחרים (תפריט, ניצחון, חוקים)
- Minimax - אלגוריתם מינימאקס, עם ובלי גיזום
- Genetic - אלגוריתם הלמידה הגנטית

בתיקיות ממוקמות התמונות.

הפעלת הפרוייקט

את המשחק מפעילים בעזרת הקובץ Onitama. כדי להפעיל את המשחק יש להריץ את הקובץ בזמן שהוא נמצא בתיקייה ביחד עם כל שאר קבצי הראקט ותיקיות של התמונות.

את ההגדרות של המשחק משנים בעזרת סדרה של קבועים בראש קובץ Onitama.

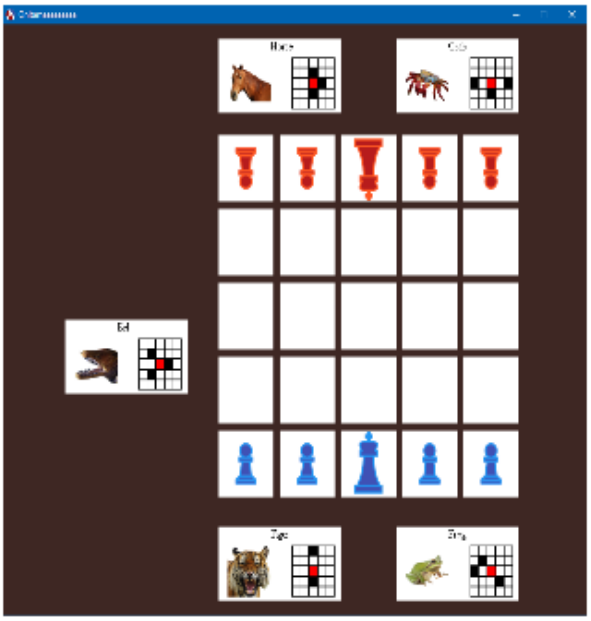
```
;CONSTANTS
(define red-ai? #f) ;Determins if red is ai or player controlled
(define blue-ai? #t) ;Determins if blue is ai or player controlled
(define ai-depth 3) ;Determins the depth of the minimax algorithm
(define alpha-beta #t) ;Determins if the algorithm used is minimax with or without pruning

(define gen-size 10) ;Amount of chromosomes in a generation
(define gene-range 5) ;Range of values per gene (5 - 0-4)
(define mut-rate 5) ;Percentage of the time a chromosome mutates
```

לדוגמה אם רוצים לשחק שחקן נגד שחקן יש להגדיר את red-ai ואת blue-ai כ#f.

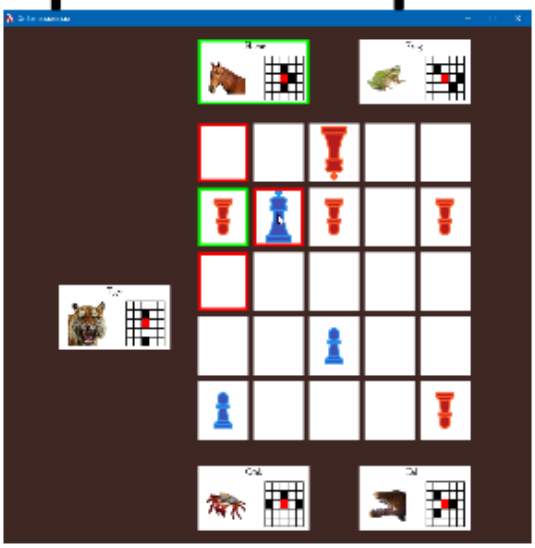
דוגמת הרצה

תפריט

התחלת משחק



סוף משחק



ניצחון

מבנה נתונים

התוכנה שומרת את כל האינפורמציה של המשחק בstructure בשם state המחולק לשלושה איברים:
(define-struct state (board cards side))

לוח - Board

איבר הלוח שומר את מיקום כל הכלים על הלוח. איבר הלוח עושה זאת בעזרת רשימה חד מימדית באורך 25.
איבר במיקום i ברשימה מייצג את הכלי הממוקם בשורה 5/i ובטור 5%i.
כל כלי הוא בעצמו structure שמחלוק לשני איברים: (define-struct piece (type side))

Type - סוג הכלי. אם הכלי הוא מלך אז הסוג שווה ל2. אם הכלי הוא חיל פשוט אז הסוג שווה ל1.
Side - הצד אליו שייך הכלי. אם הכלי הוא כחול אז הצד שווה ל 1-. אם הכלי הוא אדום אז הצד שווה ל1.

אם משבצת על הלוח היא ריקה ואין בה כל כלי אז גם הסוג וגם הצד שווים ל0.

קלפים - Cards

איבר הקלפים שומר את כל הקלפים במשחק מסויים.הרשימה באורך 5 והסדר של האיברים ברשימה קובע אצל מי הקלפים נמצאים. שני הקלפים הראשונים ברשימה שייכים לשחקן הנוכחי. הקלף השלישי הוא הקלף הקלף שאינו שייך לאף אחד מהשחקנים. שני הקלפים האחרונים שייכים לשחקן השני.

כל קלף הוא structure שמחולק לשלושה איברים: (define-struct card (name moves pic))

Name - השם של הקלף מיוצג בעזרת string.
Pic - התמונה של קלף מיוצגת בעזרת כתובת לתמונה בתיקיית המשחק
Moves - המהלכים של הקלף. מיוצג בעזרת רשימה של איברים מהposn structure. כל איבר מכיל ערך x וy המייצגים כמה משבצות בציר הx ובציר המהלך המסויים הזה מזיז את הכלי.

צד - Side

איבר הצד שומר את הצד של השחקן הנוכחי. בדומה לאיך שכלים שומרים צד, כחול זה 1- ואדום זה 1.

מינימקס

אלגוריתם המינימאקס הוא אלגוריתם לפתירה של משחקי שני שחקנים סכום אפס. האלגוריתם מסתכל מספר מסויים של תורות קדימה ומנסה למצוא מה הוא המהלך הכי טוב אם האויב משחק באופן אידאלי.
האלגוריתם מפתח nodes באופן רקורסיבי בדרך דומה לאלגוריתם depth first search. כאשר האלגוריתם מגיע לעומק הנבחר מראש (4-5) מבוצעת פונקציה היוריסטיקה על המצב ומוחזרת התוצאה. בכל שלב בסימולציה

הרקורסיבית שבו משחק המחשב, נבחרת האופציה מקסימלית. לעומת זאת לכל האופציות של השחקן היריב תמיד נבחרת האופציה המינימלית. באופן זה האלגוריתם מדמה מצב בו גם היריב מנסה לעשות maximize למצבו.

אלפא בטא

אלפא בטא היא גרסה משופרת של אלגוריתם המינימאקס. באלגוריתם זה מגזמים "עלים" של עץ הרקורסיה שניתן לחשב מראש שהם לא יהיו יעלים. באופן זה ניתן להקטין את זמן החישב באופן משמעותי בלי לפגוע ביעילות של האלגוריתם עצמו.
אם אחד הערכים במקסימזציה גדול יותר מאחד הערכים במינימזציה בnode קודם אז מגזמים את הnode. אם אחד הערכים במינימזציה נמוך יותר מערך במקסימיזציה בnode קודם אז גם מגזמים את ה.node

עומק

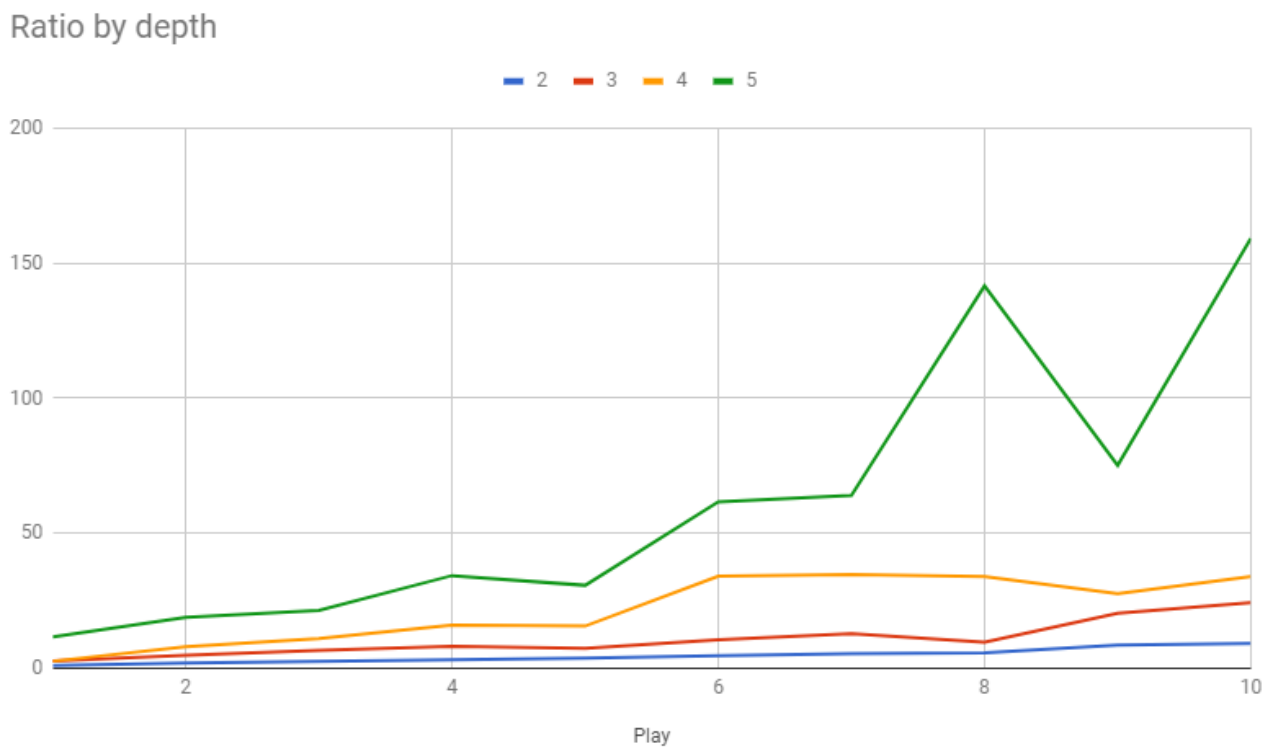
שינוי אחד שעשיתי לאלגוריתם זה שבמקום שניצחון יתן קבוע אינסוף נקודות הוא יתן מספר נקודות שתלוי בעומק.
((win? state) (* (- -1000 (* 100 depth)) max?))
הסיבה לשינוי זה היא שלעיתים האלגוריתם היה נכנס ללולאה אינסופית בה הוא עשה מהלך שהוביל לניצחון עוד 2+ תורות ולא המהלך שמביא לניצחון מידי. שינוי זה גורם לכל שהאלגוריתם תמיד יעדיף את הניצחון הקצר ביותר.

השוואה בין אלפא ובטה

בכדי למדוד את ההשפעה של גיזום על מספר הצמתים, יצרתי פונקציה שפותרת כל מצב בשנים האופנים ומדפיסה את מספר הצמתים בכל אחד מהם.

Play	2			3			4			5		
	Minimax	Alpha-Beta	Ratio	Minimax	Alpha-Beta	Ratio	Minimax	Alpha-Beta	Ratio	Minimax	Alpha-Beta	Ratio
1	99	99	1	990	368	2.690217391	13003	4969	2.616824311	154332	13251	11.64681911
2	209	110	1.9	2096	432	4.851851852	27659	3478	7.952558942	314490	16650	18.88828829
3	339	130	2.607692308	3491	531	6.574387947	45325	4117	11.00923002	521268	24349	21.40818925
4	495	156	3.173076923	4917	607	8.100494234	68335	4290	15.92890443	744910	21708	34.31499908
5	675	180	3.75	6633	893	7.427771557	90453	5785	15.6357822	1077139	35063	30.72010381
6	857	182	4.708791209	8476	809	10.47713226	105500	3099	34.04323975	1365233	22148	61.64136717
7	1049	192	5.463541667	11127	864	12.87847222	118617	3412	34.76465416	1746706	27269	64.0546408
8	1274	225	5.662222222	14454	1494	9.674698795	140857	4155	33.90060168	1974442	13946	141.5776567
9	1441	167	8.628742515	17450	855	20.40935673	165827	6018	27.55516783	2209695	29414	75.12392058
10	1589	172	9.238372093	19242	792	24.29545455	195287	5742	34.01027517	2436149	15303	159.1942103

השוותי את עשרת התורים הראשונים בעומקים שונים כדי לראות מה ההשפעה של הגיזום בכל עומק.



בגרף ניתן לראות שככל שהעומק רב יותר היחס של מספר הצמתים בין מינימקס בלי גיזום למינימקס עם גיזום עולה. בנוסף יוצא שהיחס גם גדל ככל שהמשחק מתקדם. הסיבה לתופעה זו היא כנראה שככל שהמשחק מתקדם ההבדל בין ערכים יוריסטים של תורים שונים גדל ולכן מספר הצמתים שנגזמים גדל.

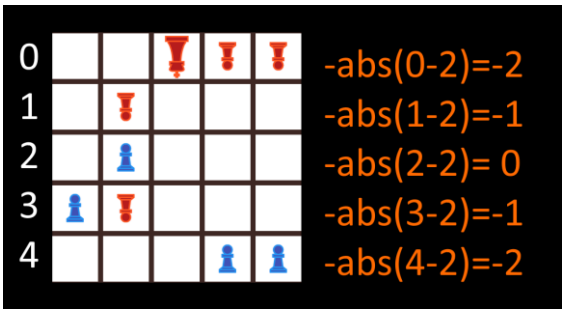
פונקצית הערכה

```
(define (heur state)
  (define allies (get-pieces state (state-side state)))
  (define enemies (get-pieces state (- (state-side state))))
  (define difference (- (length allies) (length enemies)))
  (- (+ (heur-board allies (- 10 difference))
           (heur-cards (take (state-cards state) 1)))
        (+ (heur-board enemies (+ 10 difference))
           (heur-cards (list-tail (state-cards state) 2)))))


(define (heur-board pieces val)
  (cond
    ((empty? pieces) 0)
    (else (+ (- val (abs (- 2 (posn-y (first pieces)))))
             (heur-board (rest pieces) val)))))

(define (heur-cards cards)
  (cond
    ((empty? cards) 0)
    (else (+ (get-gene chrom (card-name (first cards))) (heur-cards (rest cards))))))
```

פונקצת ההערכה מורכבת משני חלקים עיקריים: הערכת לוח והערכת קלפים. פונקצית הערכה מקיימת את אותה ההערכה גם לצד השחקן הנוכחי וגם לצד האויב ואז מחסרת את האחד מהשני. לאחר מכן פונקציית מינימקס כופלת את הערך שמתקבל מפונקצית ההערכה ב1 אם כרגע תור של מקסימיזציה וב1- אם להפך.

הערכת לוח

הניקוד הבסיסי של כל חיל הוא 10 נקודות אך הוא מושפע על ידי שני גורמים.



(abs (- 2 (posn-y (first pieces))))
ככל שהחיל רחוק יותר ממרכז הלוח ערכו קטן. זה בגלל שבמשחק אוניטמה, בדומה לשחמט יש ערך בשליטה על אמצע הלוח.

(define difference (- (length allies) (length enemies)))
ככל שיש לשחקן יש יותר כלים מליריב, הערך האינדיבידואלי של כל כלי קטן. הסיבה לזה היא שכאשר שחקן מוביל במספר כלים החלפות של כלים נעשים ליותר טובים כיוון שמצב של 2 כלים נגד 1 יותר טוב מה 3 כלים נגד 2.

הערכת קלפים

הערכת קלפים היא הערכה המבוססת על הקלפים הנמצאים בידיים של שני השחקנים. מטרת ההערכה היא לעודד שימור של קלפים איכותיים יותר ושימוש בהם רק עם הם נותנים יתרון משמעותי. לדוגמה, טיגריס הוא הקלף הכי טוב במשחק. אם לא היה קיים הערכת הקלפים השחקן שהתחיל עם טיגריס היה סתם משתמש בו בתור הראשון במטרה להגיע לאמצע הלוח. זוהי טעות אבל כי הכח של הטיגריס הא מלהחזיק אותו ביד ולאיים על כל כלי שנמצא שתי משצבות מקדימה לכלים שלך.

הניקוד של כל קלף מבוסס על הקבוע chrom. קבוע זה מבוסס על תוצאות של אלגוריתם גנטי.

גנטי

אלגוריתם גנטי הוא אלגוריתם למידה שמטרתו לעשות אופטימיזציה לפונקצית ההערכה של אלגוריתם המינמקס.
בפרוייקט שלי האגוריתם הגנטי מנסה לעשות הערכה של הניקוד שניתן לכל קלף.
כל פיתרון באלגוריתם נקרא כרומוזום. כל כרומוזום מכיל חמישה ספרות מ 0 עד ל gene-range - 1. כל ספרה כזו היא גן, מספר המייצג את הערך של קלף מסויים. הקלפים שהמשחק שלי משתמש בהם הם tiger crab horse eel frog.
הסדר של גנים בכרומוזום זהה לסדרה של קלפים ברשימה basic-set. לדוגמא: בכרומוזום (1 0 3 3 4)' לטיגריס יש את הערך הכי גבוה ולצלופח הערך הכי נמוך.
אוסף של כרומוזומים נקרא אוכלוסייה. אוכלוסייה היא רשימה של רשימות המכילה gen-size כרומוזומים. דוגמא:
((0 2 4 1 0) (1 1 0 3 0) (0 0 1 0 1) (3 3 0 0 3) (4 3 1 0 3) (1 4 0 4 0) (2 0 3 3 3) (2 1 2 0 3) (1 2 4 0 4))

1. האלגוריתם יוצר אוכלוסייה אקראית - זה הוא הדור התחילתי. פיתרונות אלה הם לרוב גרועים יותר מאשר סתם לא לתת ערך לקלפים.
2. מבחן כושר - האלגוריתם בוחן
3. סימולציה - כל כרומוזום באוכלוסייה משחק נגד כל כרומוזום אחר פעמיי. על כל ניצחון, כרומוזום מקבל נקודה, על כל הפסד הוא מפסיד נקודה. האלגוריתם אוסף את כל הנקודות ברשימה בשם fitness דוגמא: (5 3 10 7- 0 2 4- 1 12- 6 10- 5 1)'
4. **Equalize** - האלגוריתם מחסר המספר הכי נמוך ברשימה (12-) מכל הספרות ברשימה כדי שהמספר מינימלי יהיה 0.
   דוגמא: (17 15 22 5 12 14 8 13 0 18 2 17 13)
5. **Evaluate**- האלגוריתם לוקח את הכרומוזום הכי מוצלח בתחרות ונותן לו דירוג אובייקטיבי. כדי ששיטת הדירוג תיהיה אחידה האלגוריתם מנסה כל קומבינאציה של קלפית תחילתית וכל אחת עם שחקן אחר שמתחיל. כל זה יחד יוצא 240 = !5 * 2 משחקים. האלגוריתם מדפיס את מספר הנצחונות ואת אחוז הנצחונות.
6. יצירת דור חדש - האלגוריתם יוצר gen-size כרומוזומים חדשים על ידי ערבוב של כרומוזומים קודמים.
   a. בחירת 2 כרומוזומים לזווג - האלגוריתם משתמש בשיטת "roulette". האלגוריתם מגריל מספר מ1 עד לסכום כל הספרות ברשימה fitness. האלגוריתם עובר על הרשימה ומחסר מהמספר המוגרל את המספרים ברשימה עד שהמספר קטן או שווה ל0. בנקודה הזאתי הרלגוריתם בוחר בכרומוזום שהוא הגיע אליו. האלגוריתם חוזר על אותו תהליך בלי המנצח של הroulette הראשון. על פי שיטה זו לכלכרומוזום באוכלוסיה יש סיכוי להיבחר באופן פרופורציונלי לכמה נקודות הוא קיבל בתחרות.
   b. הגרלת נקודות הצלבה - האלגוריתם יוצר רשימה באורך כרומוזום בא כל ספרה מוגרלת להיות 0 או 1 דוגמא: (0 1 0 0 1)'. המטרה של רשימה הזאתי היא לבחור מה הן נקודות ההצלבה בין שני ה כרומוזומים, איזה גנים עוברים לאיזה אחד מהילדים
   c. זיווג - האלגוריתם יוצר שני ילדים לכרומוזומים הנבחרים. כל ילד בוחר איזה גן לקחת על פיק רשימת נקודות ההצלבה. דוגמא: (3 4 0 1 2)+(2 3 4 1 0) ← (3 3 1 0 0)+(2 4 4 1 2)
   d. מוטאציה - לאחר יצירת הכרומוזום יש 1 ל mut-rate סיכוי שהכרומוזום יעבור מוטאציה. לכרומוזום כזה גן אקראי מתחלף עם ערך אקראי בין 0 ל gene-range.
   e. האלגוריתם חוזר על השלב הזה עד שיש אוכלוסייה בגודל gen-size.
7. חזור לשלב 2.

היפתרות מכרומוזומים זהים

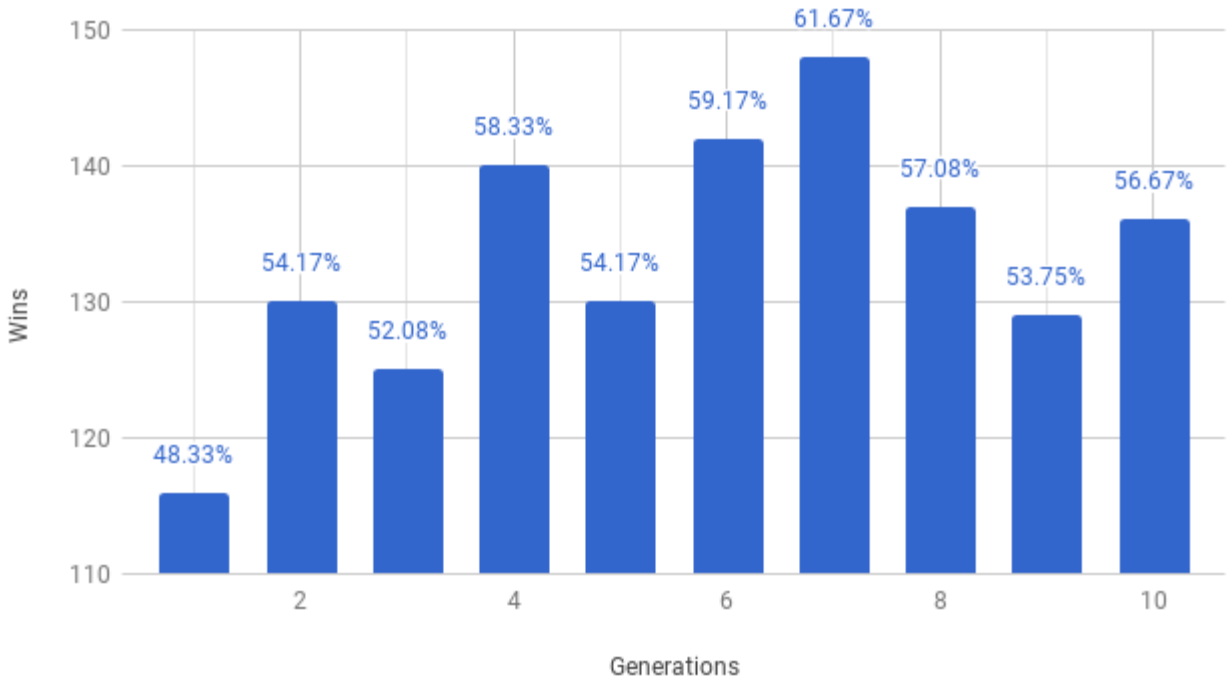
שינוי אחד שעשיתי לאלגוריתם הגנטי הוא שהאלגוריתם שיוצר דור חדש נפתר מכרומוזומים זהים. דבר זה נועד כדי לשמור על מגוון גנטי.

2 קרבות בין כל זוג כרומוזומים

במקום שיהיה קרב אחד בין כל זוג כרומוזומים באלגוריתם למידה שלי יש שני קרבות. דבר זה נועד כדי שכל צד יתחיל ראשון פעם אחת. במשחק אוניטמה יש יתרון ללהתחיל ראשון ושינוי זה נועד לתקן את האי שיוויון שיוצא מזה.

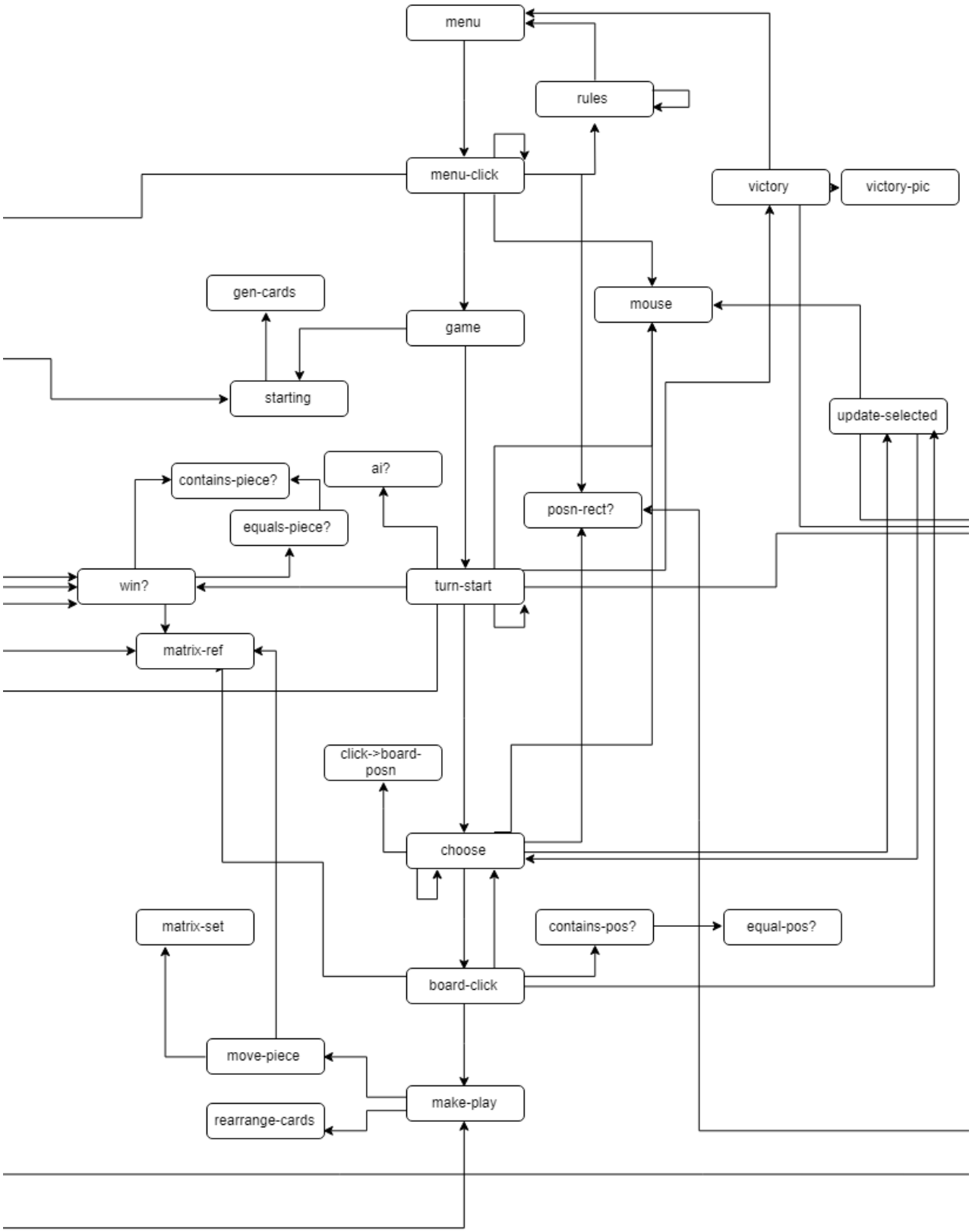
מחקר למידה

גרף זה מציג את את מספר ואחוז הנצחות של הסד הכי טוב בכל דור. האלגוריתם מתחיל עם סט אקראי שמפסיד יותר מאשר הוא מנצח. ניתן לראות שלט חל שיפור באיכות הכרומוזום בכל דור אך עד לדור שביעי יש מגמה כלפי מעלה. לאחר הנקודה הזאת האלגוריתם אינו עולה 60% ניצחון ולכן ניתן להסיק שדור שביעי זה אופטימאלי.

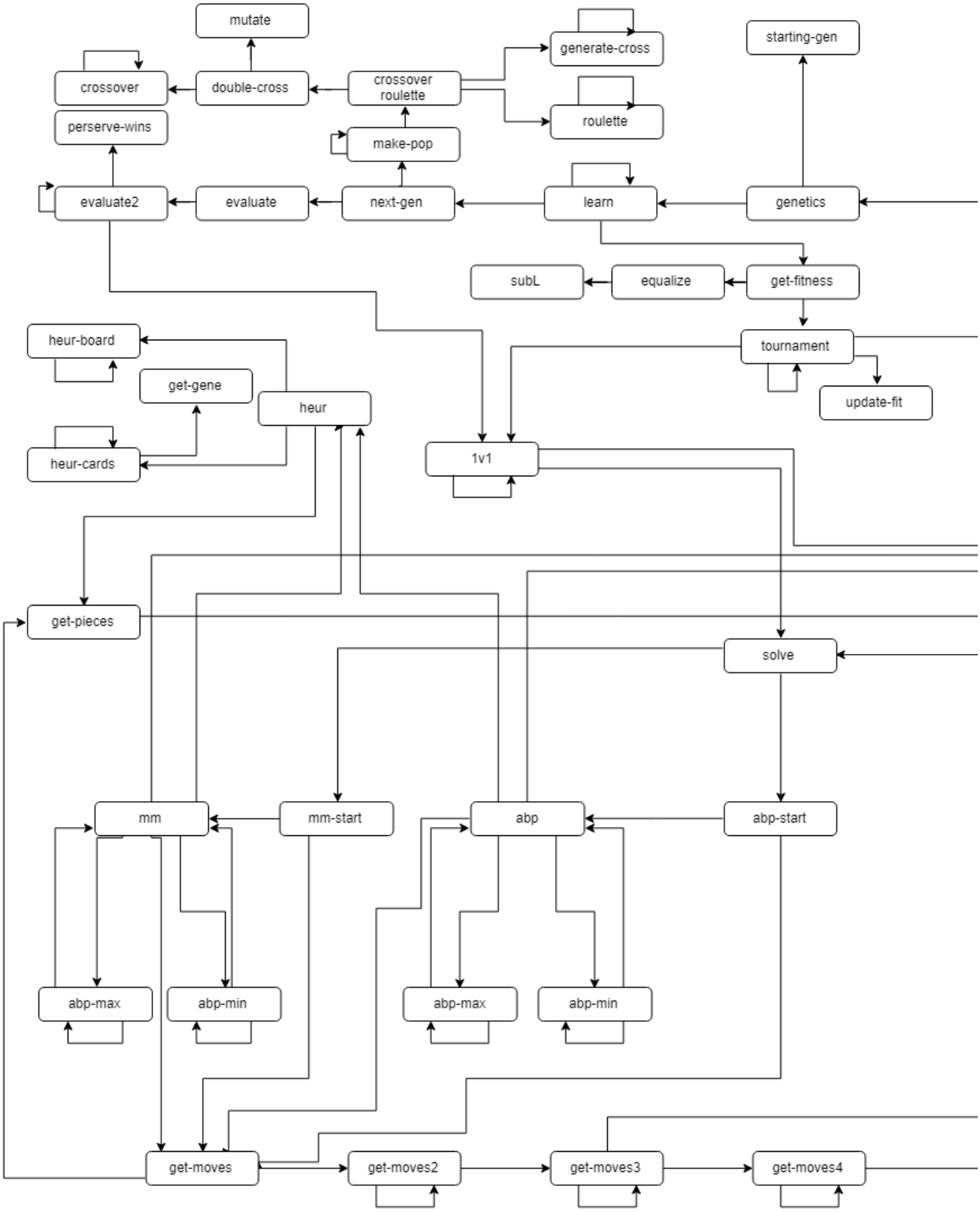


הפרמטרים שהשתמשתי בהם בשביל המחקר היו:
gen-size - 14
gene-range - 7
mut-rate - 20
ai-depth - 3

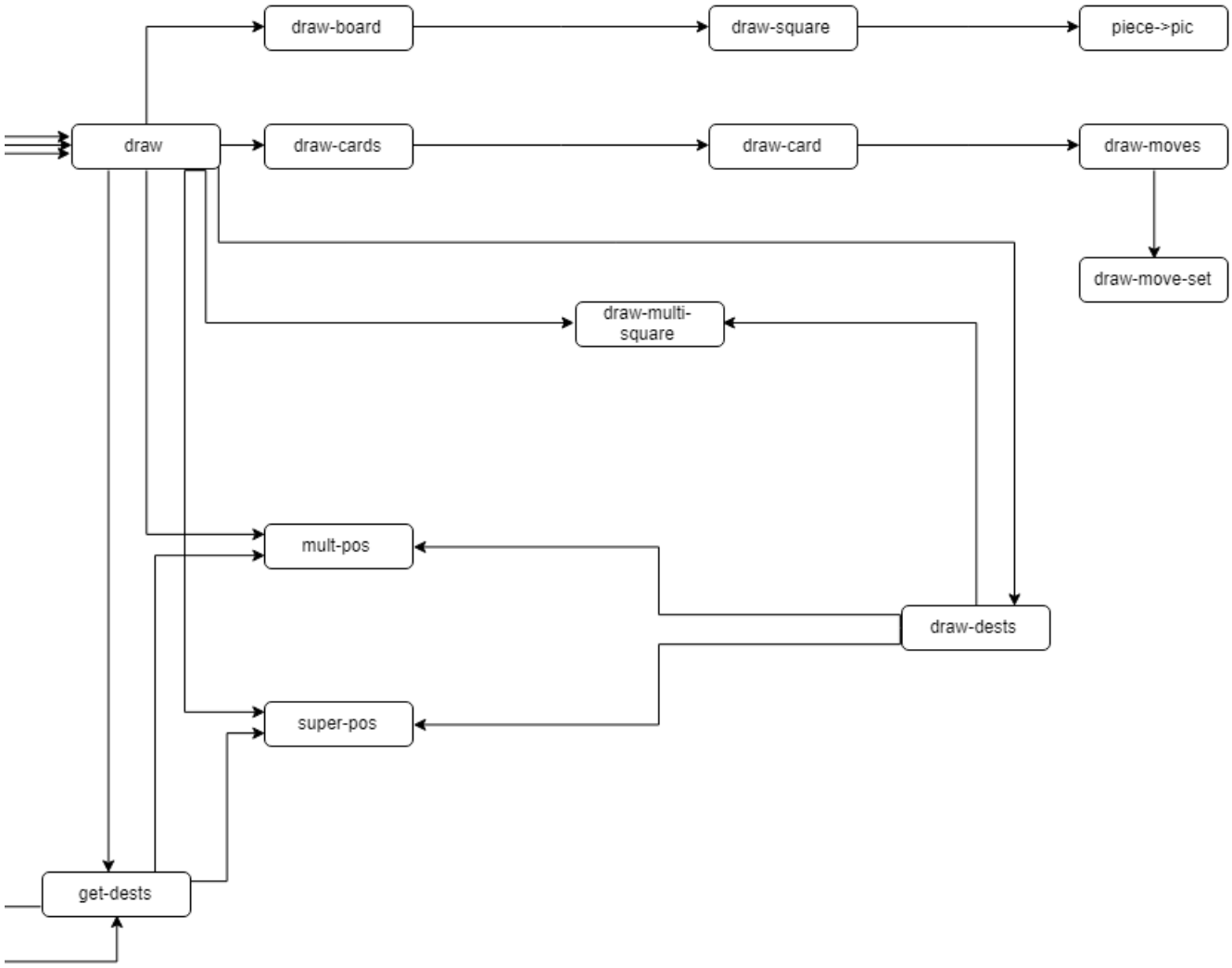
עץ קריאות מרכז



עץ קריאות שמאל - מינימקס + גנטי



עץ קריאות ימין - ציור



סיכום אישי

אני מרגיש מאוד מסופק עם עצמי שסיימתי את הפרוייקט. נתקעלתי בדרך בהרבה מכשולים בתחום האלגוריתמים. הייתי תקוע בבעיות שלא ידעתי איך לפתור גם במינמקס וגם בגנטי אך בסופו של דבר התמודדתי איתם.

יצא לי פרוייקט שאני באמת גאה בו. הוא אולי לא הכי מרשים גרפית או הכי מושך את העין אבל אני באמת מרגיש שלמדתי הרבה לאורך הדרך והידע שצברתי בנושא האלגוריתם הגנטי יעזור לי עוד הרבה בבינה מלאכותית בעתיד.

אם הייתי משנה משהו בתהליך עבודה לא הייתי דוחה חלק משמעותי מהעבודה לסוף. התחלתי את הפרוייקט יחסית מוקדם אבל בנקודה מסויימת איבדתי עניין והתקשתי להתרכז בעבודה. דבר זה יצר לחץ מיותר עלי שרק פגע בי בתהליך העבודה.

Code

Onitama.rkt

```
(require racket/include)
(require racket/list)
(require graphics/graphics)
(open-graphics)


;CONSTANTS
(define red-ai? #f) ;Determins if red is ai or player controlled
(define blue-ai? #t) ;Determins if blue is ai or player controlled
(define ai-depth 3) ;Determins the depth of the minimax algorithm
(define alpha-beta #t) ;Determins if the algorithm used is minimax with or without pruning

(define gen-size 14) ;Amount of chromosomes in a generation
(define gene-range 7) ;Range of values per gene (5 - 0-4)
(define mut-rate 20) ;1 out of how many chromosomes mutates
```

```
;Side variable are equal to either red or blue.
;Determines which side a piece belongs to.
(define blue -1)
(define red 1)

;Receives a side, returns the equivilant ai boolean.
(define (ai? side)
  (cond
    ((= side red) red-ai?)
    (else blue-ai?)))

;Structure for storing all the information of a given game state.
(define-struct state (board cards side))

;Returns a starting state.
;This is a function and not a constant so that every time it would generate a new card set.
(define (starting) (make-state board (gen-cards) blue))

;Returns true if game is won by the previous play.
(define (win? state)
  (or (not (contains-piece? (state-board state) (make-piece king (state-side state))))
      (equal-piece? (matrix-ref (state-board state) (make-posn 2 (+ 2 (* -2 (state-side state))))) (make-piece king
(- (state-side state)) ))))


(include "Board.rkt")
(include "Draw.rkt")
(include "Screens.rkt")
(include "Cards.rkt")
(include "Functions.rkt")
(include "Minimax.rkt")
(include "Genetic.rkt")
(include "Game.rkt")


;Starts program at menu
(menu)
```

Game.rkt

```
;Starts the game screen and calls the turn start function with a starting state
(define (game)
  (set! view (open-viewport "Onitamaaaaaaaaa"  950 800))
  (turn-start (starting)))

;Draws state
;If game won returns victory screen.
;If current player is ai calls turn-start recursively with ai solution of current turn.
;if currrent player isn't ai calls turn-start recursively with player choice of which turn to make.
(define (turn-start state)
  (draw state -1 -1)
  (cond
    ((win? state) (victory state))
    ((ai? (state-side state)) (turn-start (solve state)))
    (else (turn-start (choose state 1 -1 (mouse))))))

;Draws the choice made by the player and waits for new click.
(define (update-selected state card-index piece-pos)
  (draw state card-index piece-pos)
  (choose state card-index piece-pos (mouse)))

;Processes player click
;If player clicks on a card, draws the board with the card chosen and awaits new click
;If player clicks on the board calls board-click
;If player clicks on nothing waits for new click recursively
```

```
(define (choose state card-index piece-pos click)
  (cond
    ((posn-rect? click board-x (* 2 square-size) (side-y (state-side state)) square-size)
     (update-selected state 0 piece-pos))
    ((posn-rect? click card2-x (* 2 square-size) (side-y (state-side state)) square-size)
     (update-selected state 1 piece-pos))
    ((posn-rect? click board-x (* square-size 5) board-y (* square-size 5))
     (board-click state card-index piece-pos (click->board-posn click)))
    (else
     (choose state card-index piece-pos (mouse)))))

;Processes board clicks
;If player clicks on a friendly piece, redraws the board with it selected
(define (board-click state card-index piece-pos click-pos)
  (cond
    ((= (piece-side (matrix-ref (state-board state) click-pos)) (state-side state))
     (update-selected state card-index click-pos))
    ((and (posn? piece-pos) (> card-index -1) (contains-pos? (get-dests state card-index piece-pos) click-pos))
      (make-play state card-index piece-pos click-pos))
    (else (choose state card-index piece-pos (mouse)))))

;Receives a state a card a piece and a move destination and returns a state after that play is made
(define (make-play state card-index piece-pos dest-pos)
  (make-state (move-piece (state-board state) piece-pos dest-pos)
              (rearrange-cards (state-cards state) card-index)
              (- (state-side state))))

;Receives a list of cards and the index of the used card and returns the list of cards after a play is made using
the card at the index.
(define (rearrange-cards  cards card-index)
  (list (third cards) (fourth cards) (fifth cards) (list-ref cards (- 1 card-index)) (list-ref cards card-index)))

;Moves a piece from one location to another.
;If a piece is already at the destination replaces it
(define (move-piece board piece-pos dest-pos)
  (matrix-set (matrix-set board dest-pos (matrix-ref board piece-pos)) piece-pos (make-piece empty-tile empty-
tile)))
```

Minimax.rkt

```
;Chromosome used for card heuristic function
(define chrom '(4 4 1 2 3))

;Solves state using the minimax algorithm (with or withouting abp depending on settings)
(define (solve state)
  (cond
    (alpha-beta (abp-start state))
    (else (mm-start state))))


;Inital alpha beta minimax function
;Returns the child of state with the highest score
(define (abp-start state)
  (argmax (lambda (move) (abp move (sub1 ai-depth) -inf.0 +inf.0 -1)) (get-moves state)))

;Main minimax abp loop
(define (abp state depth a b max?)
  (cond
    ((win? state) (* (- -1000 depth) max?))
    ((= depth 0)  (* (heur state) max?))
    ((= max? +1) (abp-max (get-moves state) (sub1 depth) a b))
    ((= max? -1) (abp-min (get-moves state) (sub1 depth) a b))))

(define (abp-max moves depth a b)
  (cond
    ((or (empty? moves) (>= a b)) a)
    (else (abp-max (rest moves) depth (max a (abp (first moves) depth a b -1)) b))))

(define (abp-min moves depth a b)
  (cond
    ((or (empty? moves) (>= a b)) b)
```

```
    (else (abp-min (rest moves) depth a (min b (abp (first moves) depth a b 1))))))

;Inital minimax function
;Returns the child of state with the highest score
(define (mm-start state)
  (argmax (lambda (move) (mm move (sub1 ai-depth) -1)) (get-moves state)))

;Main minimax abp loop
(define (mm state depth max?)
  (cond
    ((win? state) (* (- -1000 (* 100 depth)) max?))
    ((= depth 0)  (* (heur state) max?))
    ((= max? +1) (mm-max (get-moves state) (sub1 depth)))
    ((= max? -1) (mm-min (get-moves state) (sub1 depth)))))

(define (mm-max moves depth)
  (cond
    ((empty? moves) -inf.0)
    (else (max (mm (first moves) depth -1) (mm-max (rest moves) depth)))))

(define (mm-min moves depth)
  (cond
    ((empty? moves) +inf.0)
    (else (min (mm (first moves) depth 1) (mm-min (rest moves) depth)))))


;Returns the posn in the grid of every piece of side side
(define (get-pieces state side)
  (define list (build-list 25 (lambda (n) (make-posn (remainder n 5) (quotient n 5)))))
  (define proc  (lambda (pos) (= (piece-side (matrix-ref (state-board state) pos)) side)))
  (filter proc list))

;Returns all child nodes of state
(define (get-moves state)
  (get-moves2 state 0 (get-pieces state (state-side state))))

(define (get-moves2 state card-index pieces)
  (cond
    ((= card-index 2) '())
    (else (append (get-moves3 state card-index pieces) (get-moves2 state (add1 card-index) pieces)))))

(define (get-moves3 state card-index pieces)
  (cond
    ((empty? pieces) '())
    (else (append (get-moves4 state card-index (first pieces) (get-dests state card-index (first pieces))) (get-
moves3 state card-index (rest pieces))))))

(define (get-moves4 state card-index piece dests)
  (cond
    ((empty? dests) '())
```

```
    (else (cons (make-play state card-index piece (first dests)) (get-moves4 state card-index piece (rest
dests))))))


;Returns the heuristics of state from the prespective of (state-side state)
(define (heur state)
  (define allies (get-pieces state (state-side state)))
  (define enemies (get-pieces state (- (state-side state))))
  (define difference (- (length allies) (length enemies)))
  (- (+ (heur-board allies (- 10 difference))
          (heur-cards (take (state-cards state) 1)))
        (+ (heur-board enemies (+ 10 difference))
          (heur-cards (list-tail (state-cards state) 2)))))


(define (heur-board pieces val)
  (cond
    ((empty? pieces) 0)
    (else (+ (- val (abs (- 2 (posn-y (first pieces)))))
             (heur-board (rest pieces) val)))))

(define (heur-cards cards)
  (cond
    ((empty? cards) 0)
    (else (+ (get-gene chrom (card-name (first cards))) (heur-cards (rest cards))))))

;Receives a chromosome and the name of a gene.
;Returns the gene within the chromosome with the index of the card with the name
(define (get-gene chrom name)
  (list-ref chrom (index-of (map card-name basic-set) name)))
```

Genetic.rkt

```
;Returns a randomly generated population
(define (starting-gen)
  (generate-pop gen-size))

;Receives the amount of chromosomes in a population
;Returns a randomly generated population
(define (generate-pop gen-size)
  (cond
    ((zero? gen-size) '())
    (else (cons (generate-chrom (length basic-set)) (generate-pop (sub1 gen-size))))))

;Receives the amount of genes in a chromosome
;Returns a randomly generated chromosome
(define (generate-chrom chrom-size)
  (cond
    ((zero? chrom-size) '())
    (else (cons (random gene-range) (generate-chrom (sub1 chrom-size))))))



;Returns fitness score for a population
(define (get-fitness pop)
  (tournament pop (make-list (length pop) 0) 0 1))

;Returns fitness score for a population empty list of length pop and 0 1
(define (tournament pop fitness n1 n2)
  (cond
    ((= n1 (length pop)) (equalize fitness))
    ((= n2 (length pop)) (tournament pop fitness (add1 n1) 0))
    ((= n1 n2) (tournament pop fitness n1 (add1 n2)))
    (else (tournament pop (update-fit n1 n2 fitness (1v1 (starting) (list-ref pop n1) (list-ref pop n2) 0)) n1 (add1
n2)))))

;Recieves 2 chromosomes and the number 0
;Returns 1 if c1 wins in a simulation, -1 if c2 wins and 0 if it's a tie
(define (1v1 state c1 c2 turn)
  (set! chrom c1)
  (cond
    ((win? state) (state-side state))
```

```
    ((= turn 100) 0)
    (else (1v1 (solve state) c2 c1 (add1 turn)))))

;Recieves a fitness list, 2 indexes and the result of a 1v1
;Returns the fitness list with the result added to index n1 and subtracted from index n2
(define (update-fit n1 n2 fitness result)
  (list-set (list-set fitness n1 (+ (list-ref fitness n1) result)) n2 (- (list-ref fitness n2) result)))



;Receives a population, fitness score, a sum of fitness and an empty list
;Returns a new population based on the existing population using the roulette method
(define (make-pop pop fitness sum new-pop)
  (cond
    ((<= gen-size (length new-pop)) new-pop)
    (else (make-pop pop fitness sum (remove-duplicates (append (crossover-roulette pop fitness sum) new-
pop))))))

;Receives a population, fitness and sum of fitness
;Returns 2 chromsomes made as a crossover between 2 chromosomes in the population using the roulette
method
(define (crossover-roulette pop fitness sum)
  (define win1 (roulette fitness (add1 (random sum))))
  (define win2 (roulette (list-set fitness win1 0) (add1 (random (- sum (list-ref fitness win1))))))
  (double-cross (list-ref pop win1) (list-ref pop win2) (generate-cross (length (first pop)))))

;Receives a list and a random number between 0 and the sum of fitness
;Subtracts (first fitness) from num recursively untill it reaches 0 then returns the index of (first fitness)
;Example (roulette '(5 7 0 2 4 8) 13) -> 3
(define (roulette fitness num)
  (cond
    ((<= num (first fitness)) 0)
    (else (add1 (roulette (rest fitness) (- num (first fitness)))))))

;Receives 2 chromosomes and a list of (length c1) with 0 and 1 randomly
;Returns a new chromosome that includes genes from c1 with a coresponding 0 in cross and genes from c2
with coresponding 1 in cross
;Example (crossover '(1 2 3) (4 5 6) (0 0 1)) -> '(1 2 6)
(define (crossover c1 c2 cross)
  (cond
    ((empty? c1) '())
    ((zero? (first cross)) (cons (first c1) (crossover (rest c1) (rest c2) (rest cross))))
    (else (cons (first c2) (crossover (rest c1) (rest c2) (rest cross))))))

;Receives 2 chromosomes and a list of (length c1) with 0 and 1 randomly
;Returns both possible crossovers with the chromosomes and cross
(define (double-cross c1 c2 cross)
  (cons (mutate (crossover c1 c2 cross)) (cons (mutate (crossover c2 c1 cross)) '())))

;Returns a list of (random 2) of length n
```

```
(define (generate-cross n)
  (cond
    ((zero? n) '())
    (else (cons (random 2) (generate-cross (sub1 n))))))



(define  standard '(0 0 0 0 0))

;Receives a chromosome
;Returns a string containing the number and percentage of wins by c over standard chrom out of all 240 starting
combinations for 5 cards
;The simulation is always done in depth 3 to save time
(define (evaluate c)
  (define cur-depth ai-depth)
  (set! ai-depth 3)
  (define wins (evaluate2 c (permutations basic-set)))
  (set! ai-depth cur-depth)
  (string-append (number->string wins) "/240, " (number->string (round (/ wins 2.4))) "%"))

;Recieves 2 chromosomes and a list of cards list combinations
;Returns number of wins of c1 over c2
(define (evaluate2 c combs)
  (cond
    ((empty? combs) 0)
    (else (+ (perserve-wins (1v1 (make-state board (first combs) blue) c standard 0))
             (perserve-wins (- (1v1 (make-state board (first combs) blue) standard c 0)))
             (evaluate2 c (rest combs))))))

;Returns 1 if v is 1 and 0 otherwise
(define (perserve-wins v)
  (cond
    ((= v 1) 1)
    (else 0)))



;1 out of mut-rate times returns chrom with a random gene swapped out with a random gene within gene-range
(define (mutate chrom)
  (cond
    ((zero? (random mut-rate)) (list-set chrom (random (length basic-set))  (random gene-range)))
    (else chrom)))



;Init functions for genetics algorithm
;Displays some stuff and starts the learning loop
(define (genetics)
  (display (string-append "gen-size - " (number->string gen-size) "\ngene-range - " (number->string gene-range)
"\nmut-rate - " (number->string mut-rate) "\nai-depth - " (number->string ai-depth) "\n\n"))
```

```
  (learn (starting-gen) 0))

;Genetic algorithm learning loop
(define (learn pop gen)
  (display (string-append "Generation #" (number->string gen) ": "))
  (display pop)
  (learn (next-gen pop (get-fitness pop)) (add1 gen)))

;Receives a population and the population's fitness score
;Prints out the fitness score and prints out the best chrom's wins and percentage
;Returns next generation pop
(define (next-gen pop fitness)
  (define best-chrom (list-ref pop (maxL-index fitness 0 0)))
  (display "\nFitness: ")
  (display fitness)
  (display "\n")
  (display best-chrom)
  (display " - ")
  (display (evaluate best-chrom))
  (newline)
  (newline)
  (make-pop pop fitness (apply + fitness) '()))
```

Board.rkt

```
(define board-x 350) ;Left edge of board on viewport
(define board-y 150) ;Top edge of board on viewport
(define board-start (make-posn board-x board-y))
(define square-size 100) ;Width and Height of a square on the board grid
(define square-part (* square-size 0.9)) ;The part of a square that isn't empty space

;Receives a click posn and returns a posn on the board grid
(define (click->board-posn click)
  (make-posn (quotient (- (posn-x click) board-x) square-size) (quotient (- (posn-y click) board-y) square-size)))

;Receives a board and a posn on the board grid
;returns the coresponding piece from the board
(define (matrix-ref board posn)
  (list-ref board (+ (* (posn-y posn) 5) (posn-x posn))))

;Receives a board, a posn on the board grid and a piece
;returns the board with the coresponding piece set to v
(define (matrix-set L pos v)
  (list-set L (+ (* 5 (posn-y pos)) (posn-x pos)) v))


(define-struct piece (type side))


(define pawn 1)
(define king 2)
(define empty-tile 0)


;Returns true if the side and type of both pieces is equal
(define (equal-piece? piece1 piece2)
  (and (= (piece-side piece1) (piece-side piece2)) (= (piece-type piece1) (piece-type piece2))))

;Returns true if the board contains piece1
(define (contains-piece? board piece1)
  (memf (lambda (piece2) (equal-piece? piece1 piece2)) board))

(define piece-types (list pawn pawn king pawn pawn))
(define red-side (make-list 5 red))
(define blue-side (make-list 5 blue))
(define empty-tiles (make-list 5 empty-tile))

(define board
  (append (map make-piece piece-types red-side)
          (map make-piece empty-tiles empty-tiles)
          (map make-piece empty-tiles empty-tiles)
          (map make-piece empty-tiles empty-tiles)
          (map make-piece piece-types blue-side)))
```

```
;Returns all possible destinations in the state using card-index and piece-pos
(define (get-dests state card-index piece-pos)
  (define move->dest (lambda (move) (super-pos piece-pos (mult-pos move (make-posn (- (state-side state))
(state-side state))))))
  (define legal? (lambda (dest) (and (posn-rect? dest 0 4 0 4) (not (= (piece-side (matrix-ref (state-board state)
dest)) (state-side state))))))
  (filter legal? (map move->dest (card-moves (list-ref (state-cards state) card-index)))))
```

Cards.rkt

```
(define-struct card (name moves pic))

;Moves
;N - North, S - South, W - West, E - East
(define N  (make-posn  0  1))
(define NE (make-posn  1  1))
(define NEE (make-posn  2  1))
```

```
(define NW (make-posn -1  1))
(define NWW (make-posn -2  1))
(define E  (make-posn  1  0))
(define W  (make-posn -1  0))
(define S  (make-posn  0 -1))
(define SW (make-posn -1 -1))
(define SE (make-posn  1 -1))
(define NN (make-posn  0  2))
(define EE (make-posn  2  0))
(define WW (make-posn -2  0))

;card constants
(define monkey (make-card "Monkey" (list SE SW NW NE) "Animals/monkey.png"))
(define boar (make-card "Boar" (list W E N) "Animals/boar.png"))
(define elephant (make-card  "Elephant" (list W E NE NW) "Animals/elephant.png"))
(define crab (make-card "Crab" (list EE WW N) "Animals/crab.png"))
(define tiger (make-card "Tiger" (list NN S) "Animals/tiger.png"))
(define goose (make-card "Goose" (list NW W E SE) "Animals/goose.png"))
(define dragon (make-card "Dragon" (list SW SE NWW NEE) "Animals/dragon.png"))
(define frog (make-card "Frog" (list WW NW SE) "Animals/frog.png"))
(define rabbit (make-card "Rabbit" (list SW NE EE) "Animals/rabbit.png"))
(define rooster (make-card "Rooster" (list W SW E NE) "Animals/rooster.png"))
(define mantis (make-card "Mantis" (list NE NW S) "Animals/mantis.png"))
(define horse (make-card "Horse" (list W N S) "Animals/horse.png"))
(define ox (make-card "Ox" (list E N S) "Animals/ox.png"))
(define crane (make-card "Crane" (list N SW SE) "Animals/crane.png"))
(define eel (make-card "Eel" (list NW SW E) "Animals/eel.png"))
(define cobra (make-card "Cobra" (list SE NE W) "Animals/cobra.png"))

(define basic-set (list tiger crab horse eel frog))

(define (gen-cards)
  (shuffle basic-set))
```

Draw.rkt

```
(define view '()) ;Screen
(define pix  (open-pixmap   "Onitamaaaaaaaaa" 950 800)) ;Virtual screen to prevent screen tearing

(define space 30) ; y space between the edge of the board and the cards
(define p2-y (- board-y square-size space)) ; y cord of the p2 cards
(define p1-y (+ board-y (* square-size 5) space)) ; y cords of the p1 cards
(define card2-x (+ board-x (* square-size 2) square-part)) ; x cord of both the right cards

;Returns the y cord of the equivilant card
(define (side-y side)
  (cond ((= side blue) p1-y)
        (else p2-y)))
```

```
;Draws a square with a thickness of n pixels
(define (draw-multi-square pos width height color n)
  (build-list n (lambda (n) ((draw-rectangle pix) (make-posn (+ n (posn-x pos)) (+ n (posn-y pos))) (- width (* 2 n))
(- height (* 2 n)) color))))

;Draws the board
(define (draw-board board)
  (for-each draw-square board (build-list 25 (lambda (n) (make-posn (+ board-x (* square-size (remainder n 5)))
(+ board-y (* square-size (quotient n 5))))))))

;Draws a square and a piece at pos
(define (draw-square piece pos)
  ((draw-solid-rectangle pix) pos square-part square-part "white")
  (cond
    ((not (= empty-tile (piece-type piece))) ((draw-pixmap pix) (piece->pic (piece-type piece) (piece-side piece))
pos))))

;Returns corresponding pic
(define (piece->pic type side)
  (cond
    ((and (= type pawn) (= side red)) "Pieces/red_pawn.png")
    ((and (= type pawn) (= side blue)) "Pieces/blue_pawn.png")
    ((and (= type king) (= side red)) "Pieces/red_king.png")
    ((and (= type king) (= side blue)) "Pieces/blue_king.png")))

;Draws cards at cords
(define (draw-card card x y side)
  ((draw-solid-rectangle pix) (make-posn x y) (* square-size 2) square-size "white")
  ((draw-string pix) (make-posn (+ x 85) (+ y 15)) (card-name card) "black")
  (((draw-pixmap-posn (card-pic card) ) pix) (make-posn (+ x 15) (+ y 25)))
  (draw-moves (card-moves card) (+ x 120) (+ y 25) 14 side))

;Draws a set of cards
(define (draw-cards cards side)
  (draw-card (list-ref cards 0) board-x (side-y side) side)
  (draw-card (list-ref cards 1) card2-x (side-y side) side)
  (draw-card (list-ref cards 2) board-x (side-y (- side)) (- side))
  (draw-card (list-ref cards 3) card2-x (side-y (- side)) (- side))
  (draw-card (list-ref cards 4) 100 400 side))

;Draws all moves for a card
(define (draw-move-set moves x y size side)
  (for-each (lambda (move) ((draw-solid-rectangle pix) (make-posn (+ x (* (posn-x move) side -1 size))  (- y (*
(posn-y move) side -1 size))) size size "black")) moves))

;Draws a move grid, a red square and a move set
(define (draw-moves moves x y size side)
  (define pos (build-list 25 (lambda (n) (make-posn (+ x (* size (quotient n 5))) (+ y (* size (remainder n 5)))))))
  (for-each (lambda (pos) ((draw-rectangle pix) pos size size "black")) pos)
```

```
  ((draw-solid-rectangle pix) (make-posn (+ x (* 2 size)) (+ y (* 2 size))) size size "red")
  (draw-move-set moves (+ x (* 2 size)) (+ y (* 2 size)) size side))

;Draws a set of destinations
(define (draw-dests dests)
  (for-each (lambda (dest) (draw-multi-square (super-pos (mult-pos dest (make-posn square-size square-size))
board-start) square-part square-part "red" 4)) dests))

;Draws everything
(define (draw state card-index piece-pos)
  ((draw-viewport pix) (make-rgb 0.243 0.153 0.137))
  (draw-board (state-board state))
  (draw-cards (state-cards state) (state-side state))

  (cond ((posn? piece-pos)
        (draw-multi-square (super-pos (mult-pos piece-pos (make-posn square-size square-size)) board-start)
square-part square-part "green" 4)))
  (cond ((> card-index -1)
        (draw-multi-square (make-posn (+ board-x (* card-index (- card2-x board-x))) (side-y (state-side state))) (*
2 square-size) square-size "green" 4)))
  (cond ((and (posn? piece-pos) (> card-index -1))
        (draw-dests (get-dests state card-index piece-pos))))
  (copy-viewport pix view))
```

Screens.rkt

```
;Draws menu and returns menu-click
(define (menu)
  (set! view (open-viewport "Main Menu" 400 600))
  ((draw-pixmap view) "Screens/menu.png" (make-posn 0 0))
  (menu-click (make-posn 0 0)))

;Returns player's choice on the main menu screen
(define (menu-click click)
  (cond
    ((posn-rect? click 100 200 150 80) (close-viewport view) (game))
    ((posn-rect? click 100 200 250 80) (close-viewport view) (genetics))
    ((posn-rect? click 100 200 350 80) (close-viewport view) (set! view (open-viewport "rules" 700 500)) (rules 0))
    ((posn-rect? click 100 200 450 80) (exit))
    (else (menu-click (mouse)))))
```

```
;Shows last play, waits, shows victory screens, waits for click, goes to menu
(define (victory state)
  (draw state -1 -1)
  (sleep 2)
  ((draw-pixmap view) (victory-pic (- (state-side state))) (make-posn 0 0))
  (get-mouse-click view)
  (close-viewport view)
  (menu))

;Receives a side and returns corresponding victory picture
(define (victory-pic side)
  (cond
    ((= side blue) "Screens/blue.png")
    ((= side red) "Screens/red.png")))



(define (rules page)
  (cond
    ((= page 3)
     (close-viewport view)
     (menu))
    (else
     ((draw-pixmap view) (string-append "Screens/Rules" (number->string page) ".png") (make-posn 0 0))
     (get-mouse-click view)
     (rules (add1 page)))))
```

Functions.rkt

```
;Returns the posn of the next mouse-click
(define (mouse)
  (mouse-click-posn (get-mouse-click view)))

;Returns the highest number in a list
(define (maxL L)
  (cond
    ((empty? (rest L)) (first L))
    (else (maxL (cons (max (first L) (second L)) (rest (rest L)))))))

;Returns the index of the highest number in a list
(define (maxL-index L index best-index)
  (cond
    ((= index (length L)) best-index)
    ((> (list-ref L index) (list-ref L best-index)) (maxL-index L (add1 index) index))
    (else (maxL-index L (add1 index) best-index))))

;Returns the lowest number in a list
```

```
(define (minL L)
  (cond
    ((empty? (rest L)) (first L))
    (else (minL (cons (min (first L) (second L)) (rest (rest L)))))))

;Subtracts the minimum number in L from all values in L
(define (equalize L)
  (subL L (minL L)))

;Subtracts v from all values in L
(define (subL L v)
  (cond
    ((empty? L) '())
    (else (cons (- (first L) v) (subL (rest L) v)))))

;Returns true if posn is within the bounds of a rectangle
(define (posn-rect? posn x width y height)
  (and (>= (posn-x posn) x) (<= (posn-x posn) (+ x width)) (>= (posn-y posn) y) (<= (posn-y posn) (+ y height))))

;Adds up the coordinates of 2 posn
(define (super-pos pos1 pos2)
  (make-posn (+ (posn-x pos1) (posn-x pos2)) (+ (posn-y pos1) (posn-y pos2))))

;Multiplies the coordinates of 2 posn
(define (mult-pos pos1 pos2)
  (make-posn (* (posn-x pos1) (posn-x pos2)) (* (posn-y pos1) (posn-y pos2))))

;Returns true if both posn have equal coordinates
(define (equal-pos? pos1 pos2)
  (and (= (posn-x pos1) (posn-x pos2)) (= (posn-y pos1) (posn-y pos2))))

;Returns true if L contains pos1
(define (contains-pos? L pos1)
  (not (zero? (count (lambda (pos2) (equal-pos? pos1 pos2)) L))))
```