**בינה מלאכותית לשח רגלים**

­­

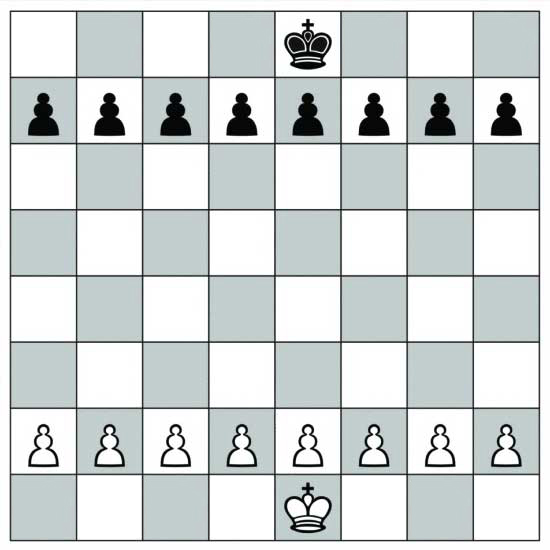
מגישים:

אילן גודיק – 316315332

יובל אלפסי – 318401015

מנחה:

מר שי בושינסקי

תאריך הגשה:

13 ליוני 2016

**תוכן עניינים**

מבוא – עמוד \_

אלגוריתם – עמוד \_

ארכיטקטורה – עמוד \_

סטטיסטיקה ובקרת איכות – עמוד \_

סיכום – עמוד \_

**מבוא**

בתחילת חודש יוני השתתפנו בהאקתון CRI, בו בנינו בינה מלאכותית שמשחקת Peshka – שח רגלים.

המנוע שלנו ל-Peshka נבנה בהשראת AlphaGo – הבינה המלאכותית שניצחה את אלוף העולם ב- Go. כמוהם, התבססנו על עץ חיפוש מונטה-קרלו בעל חיפושים חכמים במקום רנדומיים.

טרם תחילת ההאקתון, קראנו את המאמר על AlphaGo ביסודיות, והבנו לעומק כיצד המהנדסים שבחברת הבת של Google פיתחו את הבינה המלאכותית שהביסה את אלוף העולם. לאחר מכן, חשבנו שכדאי להתכוונן על מימוש משחק השחמט מהתחלה בשפת תכנות High-Level שנוחה לנו, ובנוסף, להבין לעומק את פרטיו של פרוטוקול ה-UCI. עם זאת, במהרה נוכחנו להבין שזוהי אופציה לא ריאלית – לפתח הכל מכלום בעשרים וארבע שעות. על כן, שינינו כיוון, והחלטנו שעדיף לפתח את המנוע על בסיס מנוע קיים – הורדנו את מנוע השחמט של Stockfish, בו היינו צריכים להתמודד עם אלפי שורות קוד. למדנו ביסודיות כיצד המנוע פועל, היכן הוא מתבצע קלט-פלט, כיצד הוא מפעיל את החיפוש שלו והיכן נמצאת פונקציית היוריסטיקה שלו. להאקתון הגענו עם תכנית מפורטת היכן להתחיל ומה לעשות.

במשך עשרים וארבע השעות של ההאקתון שינינו את הקרביים של המנוע של Stockfish ובנינו בינה מלאכותית שמשחקת שח רגלים באופן טוב למדי. השתמשנו באותם העקרונות שהשתמשו ב-AlphaGo והטמענו אותם באלגוריתם החיפוש שלנו. הבינה המלאכותית שלנו הודרכה על ידי עץ חיפוש מונטה-קרלו שהונחה על ידי ההיוריסטיקה של Stockfish – והתוצאות מדברות בעד עצמן.

**אלגוריתם**

הבינה המלאכותית שלנו מונחית על ידי עץ חיפוש מונטה-קרלו שמודרך באמצעות היוריסטיקה.

מדובר בעץ חיפוש בו הצמתים הם לוחות שחמט והקשתות הן מהלכי משחק אפשריים. מדובר בעץ חיפוש מונטה-קרלו, שבניגוד לעץ חיפוש Minimax שמתפתח באופן סימטרי, עץ חיפוש מונה-קרלו מתפתח באופן אסימטרי ומתמקד באזורים 'מעניינים' של המשחק יותר מאשר באזורים הפחות 'מעניינים' שפחות סביר שנגיע אליהם.

העץ מתפתח לעומק ולרוחב באופן איטרטיבי:

תחילה, בוחרים צומת משחק 'מעניין' בתחתית העץ בהתאם לסטטיסטיקה שנצברה – Selection

לאחר מכן פותחים בן 'מעניין' שלו ומוסיפים אותו לעץ – Expansion

עושים סימולציית משחק חכמה עד שמגיעים לניצחון, תיקו או הפסד – Simulation

מעדכנים לאחור את סטטיסטיקת המשחק שנצברה – Back Propagation

נסביר תהליך זה בפירוט:

Selection:

עלינו לבחור צומת 'עלה' מעניין – צומת שלא הוספנו עוד את כל בניו לעץ. לצורך כך, עלינו לבחור קבועExploration -Exploitation – כמה diversity נרצה: כמה נרצה להתעניין בצמתים שחקרנו מעט לעומת צמתים שנראים מעניינים וחקרנו הרבה, בכדי לנסות לשלול אותם. לצורך כך, התבססנו בנוסחת ה-UCT לעצי Monte-Carlo (Upper Confidence Bound). מתחילים משורש העץ, ומשם מתגלגלים למטה תוך בחירת הצומת בעל ערך ה-UCT המקסימאלי.

בהתאם לסטטיסטיקה שנצברה, ערך ה-UCT של צומת יהיה:

כאשר:

– מספר הניצחונות בתת העץ.

– סכום הevaluations בתת העץ.

– (evalWeight) המשקל שניתן לEvaluation סטטי.

n – מספר הסימולציות / איטרציות שנעשו בתת העץ.

– קבוע המחליט כמה diversity נרצה. exploration-exploitation.

t – סך כל מספר ה-simulations שנעשו.

כפי שניתן לראות – ככל שמספר הניצחונות היחסי של צומת משחק גדול יותר, כך ערך ה-UCT שלו גבוה יותר. כמו כן, ככל שעשינו יותר סימולציות באופן כללי על שאר הצמתים לעומת על הצומת הספציפי, ערך ה-UCT של הצומת יהיה יותר גבוה. בנוסף, הקבוע c קובע כמה נרצה לקחת בחשבון את ה-exploration, כמה נתחשב בכך שככל שחקרנו פחות את הצומת, כך ערך ה-UCT שלו יעלה.

אפוא, בחירה של צומת משחק מעניין בתחתית העץ נעשית באופן איטרטיבי מהשורש למטה תוך שימוש באופן חישוב של UCT שממשקל כמה צומת מעניין תוך התחשבות בקבוע diversity.

Expansion:

לאחר שבחרנו צומת מעניין בעץ, נרצה לפתח את העץ ממנו. לצורך כך, עלינו לבחור בן 'מעניין' שלו אותו נוסיף לעץ וממנו נעשה סימולציית משחק.

בעצי Monte-Carlo רגילים בוחרים בן לפיתוח באופן רנדומלי. לעומת זאת, אנחנו שאבנו השראה מהבינה המלאכותית של AlphaGo, שם עשו Expansion באופן חכם באמצעות רשת נוירונים מלאכותית שאימנו מראש. עם זאת, אנחנו הדרכנו את הפתיחה של הבן באמצעות ההיוריסטיקה של Stockfish, שהנחתה אותנו איזה בנים כדאי לפתוח ולהוסיף לעץ: מומלץ לפתח תחילה מהלכים שמעלים את הערך היוריסטי של הלוח.

בהינתן בן שרצינו לפתוח, ייצרנו את הבנים שלו וחישבנו עבורם היוריסטיקה. את ערכי היוריסטיקה נרמלנו לערכים אי שליליים ולאחר מכן לפונקצית הסתברות בעזרת וריאנט של Softmax, ועליהם הפעלנו Rolette Wheal Selection – 'גלגל המזל' כפי שנלמד בהרצאה עם שי (sampling). כך, ללוחות עם ערכי היוריסטיקה גבוהים יש יותר סיכוי להיבחר תחילה לפתיחה בעץ.

Simulation:

לאחר שפתחנו בן עלינו לסמלץ עבורו משחק עד לניצחון, הפסד או פט. בעצי מונטה קרלו רגילים מסמלצים משחק באופן רנדומי. לעומת זאת, ב-AlphaGo הם סימלצו משחק 'חכם' שהודרך על ידי רשת נוירונים שאומנה מראש. אצלנו, את סימולציית המשחק הנחתה היוריסטיקה של Stockfish, שחזתה מיהו הצומת טוב ביותר שכדאי לשחק דרכו. כך, למשחק שסומלץ הייתה משמעות מבחינת המשחק.

חשבנו גם לעשות במקום זאת ריצה רגילה של Stockfish עד לעומק נמוך מהצומת הרלוונטי.

Back Propagation:

בהתאם לתוצאת הסימולציה – הפסד ניצחון או פט, ובהתאם לשחקן שמשחק – שחור או לבן, מעדכנים עד לשורש העץ את הסטטיסטיקה שנצברה עבור המשחק. סטטיסטיקה זו תשמש את האיטרציות הבאות לצורך חישוב ערכי ה-UCT וקביעה אילו לוחות הם 'מעניינים', איזה אזורים בעץ כדאי לפתח, ואילו אזורים פחות.

בסופו של דבר, לאחר שהחיפוש מופסק, בוחרים את המהלך ממנו ביצענו הכי הרבה סימולציות. זהו המהלך שאותו חקרנו הכי הרבה, ואצלו עובר ה- principle variation. ה-UCT הנחה את החיפוש לחקור יותר אזורים בהם סיכויי הניצחון שלנו גבוהים, ולכן המהלך שנבחר לבצע בסופו של דבר הוא המהלך שבתחילת ה-principle variation אשר מגדיל את סיכויי הניצחון שלנו.

בנוסף בד"כ מחכים עוד קצת זמן עד שהצומת שביקרנו בו הכי הרבה והצומת בעל הערך הגבוה ביותר מתלכדים.

**ארכיטקטורה**

המנוע Stockfish הקיים אותו הרחבנו רשום ב C++. נדרשים מהמנוע ביצועים מהירים ככל שניתן. יש לציין שכאשר המנוע המקורי חיפש מהלך לביצוע הוא לא ביצע שום הקצאה דינאמית. אין שום malloc, new או free, הכלב באפרים בגודל קבוע על הStack. ניסינו לשמר תכונה זו, אך נדרשנו להקצות במקום יחיד: יצירת צמתים בעץ הMonte-Carlo. (היה באג זכרון לא נעים מכך שניסינו לעשות זאת על הStack – הזזות זכרון לא יציבות)

פעולת המנוע שלנו:

**קלט/פלט:**

MainTherad::search:

שינינו את פעולת החיפוש של Stockfish מהתחלת חיפוש של minimax לתחילת חיפוש מהסוג שלנו – של חיפוש מונטה-קרלו

**void** MainThread::search() {  
 mcts\_main\_search();  
}

פונקציה ראשית זו שימרה חלק ממנגנוני הניהול של ממשק המשתמש שהיה בתוך stockfish, אך בנוסף גם הוספנו פונקציית דיווח מצב משלנו לUCI הנמצאת ב

mcts\_pv.cpp: std::string mcts\_pv\_print(MCTS\_Node& root) {

...  
MCTS\_PV pv = mctsPv(&root, pvMoves);

...

Value v = Value(pv.score); *// Same scale as Eval::evaluate - in pawns.*

...  
ss << **"info"** << **" depth "** << pv.depth / ***ONE\_PLY*** << **" seldepth "** << pv.depth  
 << **" multipv "** << 1  
 << **" score "** << UCI::value(v);  
  
ss << **" nodes "** << nodes\_searched  
 << **" nps "** << nodes\_searched \* 1000 / elapsed;  
  
ss << **" tbhits "** << TB::Hits  
 << **" time "** << elapsed  
 << **" pv"**;  
  
**for** (Move m : pv.moves)  
 ss << **" "** << UCI::move(m, **false**);

...

}

mctsPv:

פונקציה זו יוצרת כביכול PV, מסלול שלדעתינו הכי טוב שידווח לUCI: (בחירה חוזרת של selectBest ובניית MCTS\_PV)

MCTS\_PV mctsPv(MCTS\_Node\* node, std::vector<Move>& pvBuffer) {  
 **if** (node == **nullptr** || node->totalVisits < Search::pvThreshold ||

!node->fully\_opened() */\*leaf\*/*) {  
 **return** MCTS\_PV(std::vector<Move>(0), 0 */\*changed in rec\*/*,

Time.elapsed() + 1, 0, Threads.nodes\_searched());  
 }  
 MCTS\_Edge\* bestEdge = node->selectBest();  
 **if** (bestEdge != **nullptr**) {  
 pvBuffer.push\_back(bestEdge->move);  
 MCTS\_PV childPv = mctsPv(&bestEdge->node, pvBuffer);childPv.moves = pvBuffer;  
 childPv.depth++;  
 **if** (node->incoming\_edge == **nullptr**)  
 childPv.score = bestEdge->score();  
 **return** childPv;  
 } **else** {  
 **return** mctsPv(**nullptr**, pvBuffer);  
 }  
  
}

selectBest: בחירת הבן הכי טוב מצומת; הבן שבוקר הכי הרבה.

**Mcts.cpp:**

בקובץ זה קורת הרצת האלגוריתם MCTS עצמו.

תחילה, יש לנו מספר קבועים:

**const double** cpuct = 0.01;  
**const double** explorationExponent = 0.5;  
**const float** evalWeight = 0.5f;  
**const int** pvThreshold = 7;  
**const float** normalizationFactor = 200; *// Something like a pawn*

cpuct הוא קבוע המוכפל בprior, לאיזון בין הevaluation האמיתי לבין הניחוש הראשוני.

explorationExponent הוא החזקה של השורש בנוסחת הUCB

evalWeight הוא הגורם הלינארי המאזן בין הstatic evaluation לבין הdynamic evaluation בכל צומת.

pvThreshold: מהו מספר הביקורים המינימאלי בצומת כך שהוא בעל משמעות; ניתן להציג אותו בpv.

normalizationFactor: קבוע מנרמל עבור הevaluation בsigmoid, דרוש לשם יציבות.

mcts\_chess\_playing:

קובץ קוד המכיל הרחבות לאחזור מידע לגבי הפוזיציה הנוכחית של הלוח.

Table bases:

בתחילת ההרצה של פונקצית החיפוש הראשית שלנו, mctsSearch, אנו מאתחלים את הTable bases בדומה לאיך שהם אותחלו בStockfish:

**void** initTableBase() {  
 TB::Hits = 0;  
 TB::RootInTB = **false**;  
 TB::UseRule50 = Options[**"Syzygy50MoveRule"**];  
 TB::ProbeDepth = Options[**"SyzygyProbeDepth"**] \* ***ONE\_PLY***;  
 TB::Cardinality = Options[**"SyzygyProbeLimit"**];  
  
 *// Skip TB probing when no TB found: !TBLargest -> !TB::Cardinality* **if** (TB::Cardinality > TB::MaxCardinality) {  
 TB::Cardinality = TB::MaxCardinality;  
 TB::ProbeDepth = ***DEPTH\_ZERO***;  
 }  
}

getGameResult: מחזיר האם יש להמשיך את המשחק, נצחון, הפסד או תיקו.

מתחשב בtableBases.

PlayingResult getGameResult(Position& pos, **int** numMoves) {  
 PlayingResult res;  
 **if** (isInTableBase(pos, &res))  
 **return** res;  
  
 Bitboard promoted = promotedPieces(pos);  
 Color sideToMove = pos.side\_to\_move();  
 Bitboard ourPromoted = pos.pieces(sideToMove) & promoted;  
  
 **if** (ourPromoted)  
 **return *Win***;  
 **if** (promoted */\*&& !ourPromoted\*/*) {  
 **return *Lose***;  
 }  
  
 **if** (numMoves == 0) {  
 **if** (pos.checkers())  
 **return *Lose***;  
 **else  
 return *Tie***;  
 }  
  
 **if** (pos.is\_draw())  
 **return *Tie***;  
 **return *ContinueGame***;  
}

הערה: ברגע שהגענו למצב שיש לנו מלכה, אנו מעבירים את המשחק לstockfish כדי שישיג את המט.

**Evaluation & Prior:**

בפונקצית הevaluation שלנו, עשינו גם quiescence search עם מעט minimax במקרה שאנחנו נכנסים לשח,

וזה קורה בפונקציה

Value qeval(Position& pos) {  
 **if** (!pos.checkers()) {  
 **return** Eval::evaluate(pos);  
 }  
  
 ExtMove evasionsBuffer[16];  
   
 StateInfo st;  
 CheckInfo ci(pos);  
  
 ExtMove\* evasions = evasionsBuffer;  
 ExtMove\* end = generate<***LEGAL***>(pos, evasions);  
  
 **if** (evasions == end) {  
 *// Mate, we don't have moves, we lost.* **return** -***VALUE\_MATE***;  
 }  
  
 Value bestValue = -***VALUE\_INFINITE***;  
 **while** (evasions != end && \*evasions != ***MOVE\_NONE***) {  
 *// rec on the move, max.* Value currentValue;  
  
 **bool** isCheck = pos.gives\_check(\*evasions, ci);  
 pos.do\_move(\*evasions, st, isCheck);  
 **if** (!isCheck) {  
 currentValue = Eval::evaluate<**false**>(pos);  
 } **else** {  
 currentValue = -qeval(pos);  
 }  
 pos.undo\_move(\*evasions);  
  
 **if** (currentValue > bestValue) {  
 bestValue = currentValue;  
 }  
  
 evasions++;  
 }  
  
 **return** bestValue;  
}

saveEval מחליט אם אנחנו בשח ויש צורך לבצע את qeval או לא. בסופו של דבר הוא מחזיר את הevaluation לפי ההוריסטיקה הרגילה של Stockfish.

Calc\_priors מחשב Softmax על כל ההיוריסטיקות בשביל ליצור הסתברויות לpriors ולשם הבחירה האקראית לפי RWS Sampling.

לשם קבלת תוצאות הגיוניות, יש לנרמל לפי הערך הגדול ביותר ולהקטין את כל הערכים כך שההבדל ביניהם לאחר הSoftmax לא יהיה אכזרי מדי; כאן בא קבוע הנרמול normalizationFactor שהוצג קודם לכן.

*// e^(x/t - max)***void** calc\_exp\_evals(Position& pos, ExtMove\* moves, **int** size) {  
 StateInfo st;  
 CheckInfo ci(pos);  
  
 **int** count = 0;  
 **for** (**int** i = 0; i < size && moves[i] != ***MOVE\_NONE***; i++, count++) {  
 *// calculate move values (heuristics)* Value eval = safeEval(pos, moves[i], ci, st);  
 moves[i].setPrior(**float**(eval) / Search::normalizationFactor);  
 }  
  
 **float** max = -***VALUE\_INFINITE***;  
 **for** (**int** i = 0; i < count; i++) {  
 max = std::max(max, moves[i].getPrior());  
 }  
  
 **for** (**int** i = 0; i < count; i++) {  
 **float** eval = std::exp(moves[i].getPrior() - max);  
 moves[i].setPrior(eval);  
 }  
  
}  
  
**void** calc\_priors(Position& pos, ExtMove\* moves, **int** size) {  
 *// e^(x/t - max) / sum* calc\_exp\_evals(pos, moves, size);  
  
 **double** expSum = 0;  
 **for** (**int** i = 0; i < size; i++) {  
 expSum += moves[i].getPrior();  
 }  
  
 **for** (**int** i = 0; i < size; i++) {  
 moves[i].setPrior(**float**(moves[i].getPrior() / expSum));  
 }  
}

בנוסף, רצינו להשתמש בייצוג המקורי של Stockfish לתוצאת פלט של היוריסטיקה למהלך, אך היה שם שימוש בint32 ולא בfloat כרצונינו.

לכן השתמשנו בunion type כדי לקרוא ולרשום את הביטים שרצינו בתוך הrecord הנ"ל.

**float** getPrior() {  
 **union** {**float** f; **int** i;} eval;  
 eval.i = value;  
 **return** eval.f;  
}  
  
**void** setPrior(**float** prior) {  
 **union** {**float** f; **int** i;} eval;  
 eval.f = prior;  
 value = Value(eval.i);  
}

UnopenedMove:

**struct** UnopenedMove {  
 Move move;  
 **float** expPrior;  
 **float** relativePrior; *// In [0, 1], divided by the sum of e^x for all yet*

*unopened moves.* **float** absolutePrior; *// In [0, 1], divided by the sum of all e^x*

...

}

שמרנו פה את סכום הe^x-max של הevaluations, כך שברגע שמוציאים מהלך מתוך רשימת המהלכים שעוד לא פתוחים, נוכל לקבל את פונק' ההסתברות החדשה בזמן O(1), ובלי חישוב מחדש של כל ההסתברויות של שאר הצמתים.

**UnopenedMoves** בתוך mcts.h מנהל את רשימת הבנים הלא פתוחים,

כולל אתחול, מחיקה, מספר הבנים וכו'.

**MCTS\_Node:** בתוך mcts.h הוא צומת יחיד בעץ הMonte-Carlo.

ישנם מספר מצבים בהם יכול להימצא צומת בעץ:

*// States: Uninitialized - Edges empty & unopened moves empty  
// Not fully opened - Edges not empty & unopened moves not empty  
// Fully opened - Edges not empty & unopened moves empty*

אנחנו חוסכים זכרון בכך שאנחנו לא פותחים את כל הבנים על ההתחלה, אלא רק כשאנחנו נדרשים לכך.

להלן השדות שנשמרים, בנים + הורה + סטטיסטיקה בסיסית:

**bool** initialized;  
std::vector<MCTS\_Edge\*> edges;  
UnopenedMoves unopened\_moves;  
NumVisits maxVisits;  
NumVisits totalVisits;  
MCTS\_Edge\* incoming\_edge;

בנוסף יש לנו קשתות בעץ, MCTS\_Edge:

**struct** MCTS\_Edge {  
 MCTS\_Node node;  
 Move move;  
 **float** prior;  
 EvalType evalSum;  
 NumVisits numEvals;  
 NumVisits rolloutsSum;  
 NumVisits numRollouts;  
 **float** overallEval;  
  
 **int** score() {  
 **return** (**int**) ((overallEval \* 30) \* **int**(***PawnValueEg***));  
 }  
  
 **void** update\_stats(**int** rolloutResult, **double** evalResult, **float** evalWeight) {  
 rolloutsSum += rolloutResult;  
 numRollouts += 1;  
 evalSum += evalResult;  
 numEvals += 1;  
 overallEval = compute\_overall\_eval(evalWeight);  
 }

...

}

**בmcts.cpp יש את אלגוריתם הMonte Carlo Tree Search עצמו.**

סה"כ מימוש די סטנדרטי, אך כזה שפועל תחת כל הAPI של Stockfish, בתוספת לכך שאנחנו נמנעים מרקורסיה בהתאם לStockfish, עובדים עם אובייקט Position יחיד ומשתמשים בstack מנוהל ידנית ישירות המוקצה על הStack בגודל קבוע.

**קצת תיאוריה עם סיגמואידים:**

לפני בפרוייקט אחר לפני פרוייקט זה, פיתחנו נוסחאות מתמטיות לסיגמואידים מכיוונים שונים, כך שלדוגמא יהיה אפשר לחשב את מקדם השיפוע של הסיגמואיד בהינתן שינוי בערך הקלט delta x מאמצע הסיגמואיד (שעצמו יכול להשתנות), והערך ההסתברותי בפלט p:

זה נחמד שיוצאות נוסחאות כ"כ נקיות לזה.

קוד הלולאה הראשית של MCTS:

**void** mctsSearch(Position& pos, MCTS\_Node& root) {  
 **const int** printEvery = 1000;  
 *// Updated by check\_time()* initTableBase();  
  
 StateInfo sts[MAX\_PLY];  
 StateInfo\* lastSt = sts + MAX\_PLY;  
 ExtMove moveBuffer[128];  
 MCTS\_Node\* moveHistoryBuffer[MAX\_PLY];  
 MCTS\_Node\*\* moveHistory = moveHistoryBuffer;  
  
 **int** iteration = 0;  
 **while** (!Signals.stop) {  
 MCTS\_Node\* node = &root;  
 StateInfo\* currentSt = sts;  
  
 **int** rolloutResult;  
 **double** evalResult;  
  
 **int** numMoves = node->getNumMoves(pos, moveBuffer);  
 PlayingResult gameResult = getGameResult(pos, numMoves);  
 **while** (gameResult == ***ContinueGame*** && !node->isLeaf()) {  
 MCTS\_Edge\* child = select\_child\_UCT(node);  
 do\_move\_mcts(pos, node, currentSt, child, moveHistory);  
  
 *// If we reach the maximum depth, assume repeat or whatever.* **if** (currentSt == lastSt) {  
 gameResult = ***Tie***;  
  
 } **else** {  
 gameResult =

getGameResult(pos, node->getNumMoves(pos, moveBuffer));  
 }  
 }  
  
 **if** (gameResult != ***ContinueGame***) {  
  
 rolloutResult = gameResult;  
 evalResult = rolloutResult;  
  
 } **else** { *// at leaf = not fully opened.* MCTS\_Edge\* childEdge = node->open\_child(pos, moveBuffer);  
 do\_move\_mcts(pos, node, currentSt, childEdge, moveHistory);  
 **if** (currentSt == lastSt) {  
 rolloutResult = ***Tie***;  
 } **else** {  
 rolloutResult = rollout(pos, currentSt, lastSt, moveBuffer);  
 }  
 evalResult = eval(pos);  
 }

*// Back propagation* **while** (node->incoming\_edge != **nullptr**) {  
 node->incoming\_edge->

update\_stats(rolloutResult, evalResult, evalWeight);  
  
 MCTS\_Edge\* childEdge = node->incoming\_edge;  
  
 undo\_move\_mcts(pos, node, currentSt, moveHistory);  
  
 *// Update max stats in the parent* node->update\_child\_stats(childEdge);  
 *// Nega-max* rolloutResult = -rolloutResult;  
 evalResult = -evalResult;  
 }  
  
 *// optionally print move* **if** (iteration % printEvery == 0) {  
 mcts\_check\_time();  
 }  
 **if** (Time.elapsed() > 1000 && iteration % printEvery == 0) {  
 **sync\_cout** << mcts\_pv\_print(root) << **sync\_endl**;  
 **if** (debug\_UCT) {  
 **for** (MCTS\_Edge\* edge: root.edges) {  
 **sync\_cout** << UCI::move(edge->move, **false**) << **": "** <<

edge->numRollouts << **" = "**;  
 printf(**"%.2f"**, edge->overallEval);  
 std::cout << **", "**;  
 }  
 std::cout << **sync\_endl**;  
 }  
  
 }iteration++;  
 }  
}

שלב הRollout:

PlayingResult rollout(Position& pos, StateInfo\*& currentStateInfo, StateInfo\* lastStateInfo, ExtMove\* moveBuffer) {PlayingResult result = getGameResult(pos, getNumMoves(pos, moveBuffer));  
 Move movesDone[MAX\_PLY];  
 **int** filled = 0;  
 **while** (result == ***ContinueGame***) {  
 ExtMove\* startingMove = moveBuffer;  
  
 ExtMove\* endingMove = generate<***LEGAL***>(pos, startingMove);  
 **int** movesSize = countValidMoves(startingMove,

**int**(endingMove - startingMove));  
  
 calc\_priors(pos, startingMove, movesSize);  
  
 Move chosenMove = sampleMove(pos, startingMove);  
 pos.do\_move(chosenMove, \*currentStateInfo,

pos.gives\_check(chosenMove, CheckInfo(pos)));  
 movesDone[filled] = chosenMove;  
 filled++;  
 currentStateInfo++;  
 **if** (currentStateInfo == lastStateInfo) {  
 result = ***Tie***;  
 **break**;  
 }  
 result = getGameResult(pos, getNumMoves(pos, moveBuffer));  
 }  
  
 **for** (**int** i = filled - 1; i >= 0; i--) {  
 currentStateInfo--;  
 pos.undo\_move(movesDone[i]);  
 }  
  
 *// Computed in getGameResult, notice reference above.* **return** result;  
}

בחירת עלה לפי UCT:

MCTS\_Edge\* select\_child\_UCT(MCTS\_Node\* node) {  
 *// attest( ! children.empty() );* **double** max\_UCT\_score = -***VALUE\_INFINITE***;  
 MCTS\_Edge\* bestEdge = **nullptr**;  
 **for** (MCTS\_Edge\* child: node->edges) {  
 **double** score = child->overallEval +  
 cpuct \* child->prior \* std::pow(node->totalVisits, explorationExponent) / (1 + child->numRollouts);  
 **if** (score > max\_UCT\_score) {  
 max\_UCT\_score = score;  
 bestEdge = child;  
 }  
 }  
  
 **return** bestEdge;  
}

פתיחת בן חדש:

MCTS\_Edge\* MCTS\_Node::open\_child(Position& pos, ExtMove\* moveBuffer) {  
 *// Precondition: not terminal, leaf => possible moves not empty* initialize(pos, moveBuffer);  
 *// unopened\_moves not empty.  
 // sample move according to prior probabilities / take maximal probability* UnopenedMove move = sampleMove(pos, unopened\_moves.unopened\_moves,

unopened\_moves.numMoves);  
 *// remove it from unopened\_moves and insert to edges.* unopened\_moves.remove(move);  
 MCTS\_Edge\* childEdge = **new** MCTS\_Edge(move.move, move.absolutePrior);

*// Notice Allocation here!* edges.push\_back(childEdge);  
 **return** edges[edges.size() - 1];  
}

**סטטיסטיקה ובקרת איכות**

המנוע שלנו כל הזמן ניצח אותנו (האנשים) במשחק מולו, אך אנחנו מרגישים שאנחנו ברמה של חובבנים עדיין במשחק.

לעומת זאת, המנוע שלנו בד"כ מפסיד לStockfish מבחינת זה שהוא לא בוחר את המהלכים הטובים ביותר, והניקוד היחסי שלנו יורד קצת לרעתינו במהלך המשחק.

לדעתינו זה הישג טוב מאוד.

**סיכום**

כחודש לפני ההאקתון שי בושינסקי הציע לנו לקחת חלק בהאקתון ולפתח בו בינה מלאכותית לשח רגלים תוך שימוש ברעיונות שדומים לבינה המלאכותית של AlphaGo. לקחנו משימה זו על עצמינו והתחלנו בלמידה קדחתנית לקראת ההאקתון.

טרם להאקתון, היה עלינו להיכנס לנבכיו העמוקים של המנוע של Stockfish, ולהשקיע שעות על גבי שעות של למידתו על בוריו. יתר על כן, היה עלינו לקרוא באופן יסודי את המאמר של AlphaGo, ובזמן ההאקתון לפתח באופן דקדקני מנוע משלנו.

בהאקתון עצמו ניתנו לנו עשרים וארבע שעות בלבד לפתח את מנוע השח-מט שלנו – זמן מועט וקצר ביותר. למרבה השמחה, עמדנו ביעד.

לסיכומו של דבר, הצלחנו להרים מנוע טוב למדי שפועל ומשחק. היה עלינו ללמוד טכניקות חדשניות ופורצות דרך בתחום הבינה המלאכותית, תוך התמקצעות בקוד קיים, והתמצאות במרחב בלתי מוכר. זו הייתה חוויה מאתגרת מעשירה ומרחיבת אופקים, ואנחנו שמחים שיצא לנו לקחת בה חלק.