Justin Domingue

260588454

Project Report

COMP 424

Presented to Prof. Joelle Pineau

McGill University

Thursday, March 31, 2016

INTRODUCTION

In order to solve Hus, I have consulted many resources (textbooks, papers, TAs and others). I have started with Minimax with average results. I then extended added Alpha-Beta pruning as explained in the course textbook [1]. This player proved to be surprisingly strong without any enhancements. I did consider many improvements, however, as discussed later in this report. Finally, after much work on Alpha-Beta pruning, I shifted gears and investigated using Monte Carlo Tree Search. With highly tuned parameters, this player turned out to be my strongest one. In this report, I will first discuss my final player, then provide the theoretical basis for that player and the advantages and disadvantages of the approach. I will then present the other approaches tested. Finally, I will propose various areas of improvements.

A quick word on optimization. Throughout the development on my agent, I have thoroughly tuned every parameter presented in the literature or devised on my own through self play involving at least 30 distinct games (and more than 100 for critical parameters). I wrote a bash script to automate the testing of multiple values. For example, `./run_simulation.sh explorationconstant 0.1 0.5 1` would play Monte Carlo Tree search against itself with values 0.1 vs 0.5, 0.1 vs. 1 and 0.5 vs. 1. See Annex B for the script.

OVERVIEW AND TECHNICAL BASIS OF MY APPROACH

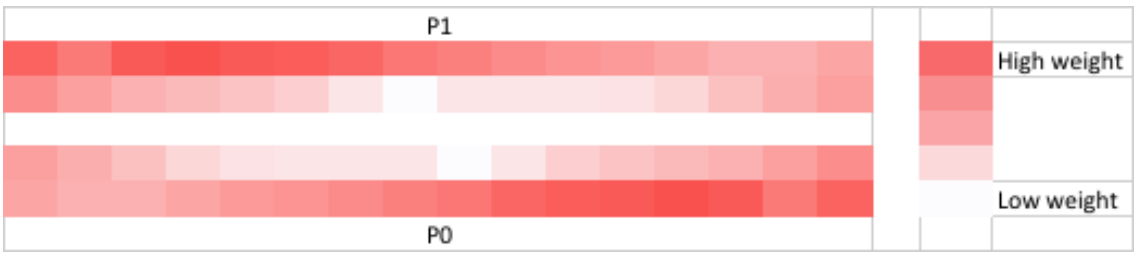
Best Player. My best player uses the Monte Carlo Search Tree (MCTS) method. MCTS is a "method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results" [2]. As described by Gelly in his 2011 paper [3], MCTS is sequentially best-first, allowing a constant shift of the focus of attention to the most promising part of the tree. The search tree is composed of a variable number of nodes, each with information on the running number of wins and visits, reference to its children and parent, as well as other bookkeeping information. A simulation is performed in four steps: 1) selection: until a leaf node is reached, select nodes from the tree based on a *tree policy*; 2) expansion: add a (single) new node to the tree; 3) rollout: simulate the game from the newly added node to game over using a *default policy* and compute the reward (1 for win, 0 otherwise in my implementation); 4) back-propagation: propagate the reward up the tree, properly incrementing the number of visits and updating the win counts. It is very important to properly devise the tree and default policy – and this is where I spent most of my time optimizing.

Tree policy. The tree policy is used in the selection phase to select a path within the search tree to explore. There is an obvious tradeoff between exploiting a path we know is promising and exploring new

paths which might be more optimal. The UCT algorithm has been suggested as a way to balance exploration and exploitation [2]. Experimentally, a value of 0.45 for the exploration constant was optimal. Furthermore, I have determined that expanding only the k=15 *best* children ordered by their heuristic value (instead of all) in the expansion phase boosted the performance of my agent significantly.

Default policy. The simplest default policy is choosing a move uniformly at random [3]. However, injecting domain-specific knowledge has proven quite beneficial in my experiments. I have determined the following best policy: for the first 30 moves (15 turns), greedily select uniformly at random from the k=4 best children ranked by heuristic value after which, play uniformly random moves. At this point, it should be said that MCTS has been shown to approximate Minimax search at the limit [3]. It seems that by first playing using game knowledge orients the simulations towards playing like an optimal player. However, evaluating the heuristic value of each node is quite costly – hence the switch to a random policy near the end of the simulations. This policy accomplishes an optimal balance between playing like an optimal player and playing many simulations (and thus approaching Minimax optimality).

Heuristic function. I have tried many different heuristic functions but the one that fared the best is the sum of seeds in every pit of the current player. Using ridge regression (stochastic gradient descent), I have learned the weight associated with each pit on an automatically generated dataset[1]. The dataset comprises of 300,000 randomly generated states with turn number from 0 to 35. Each state was tagged with an outcome computed by running 10,000 simulations where each move was uniformly chosen from amongst the k best based on the heuristic value. It should be noted that I have learned the L2-norm regularization parameter through 10-fold cross validation. My final model had a coefficient of determination of 0.91. Here is a visual representation of the 32 learned weights (P1 side is a rotation of P0). Each colored square is a pit. A dark color means a heavy weight while a light color means a low weight.



[1] Python code available in the zip file attached to this document in allcode/ML.

It is interesting to note that with these weights, the heuristic will favor states with little seeds in vulnerable pits (the inner row is subject to attacks from the opponent) and favor states with more seeds in the right corner of the outer row, where the player can launch attacks. In order words, the heuristic was optimized for defense and attack at the same time.

Selecting the winning action. Much research has been done in determining how to return a move once the time is up [2]. I have tested selecting the root child with 1) the highest reward and 2) the most visits. It turns out that the two are usually the same but 1) was slightly more favorable in practice.

Other optimizations. At each turn, Monte Carlo tree search builds a search tree. At the beginning of every turn, I try to recuperate the previous search tree by inferring the opponent's move. With this, I can salvage about 1000 simulations every turn. Also, instead of using `HusBoardState.getLegalMoves()`, I wrote my own functions to efficiently sample the list of legal moves using a min-heap – up to five times faster.

Fruitless optimizations. A brief word on optimizations which didn't improve the performance. A problem with the UCT value is that until a few simulations have been performed, it is quite imprecise. *Progressive bias* has been suggested to mitigate that: add a term computing the heuristic value to the UCT formula [2]; Decaying reward in the back-propagation phase: weight states close to the end of the game more heavily than those far way; Cutoff after X moves in the default policy.

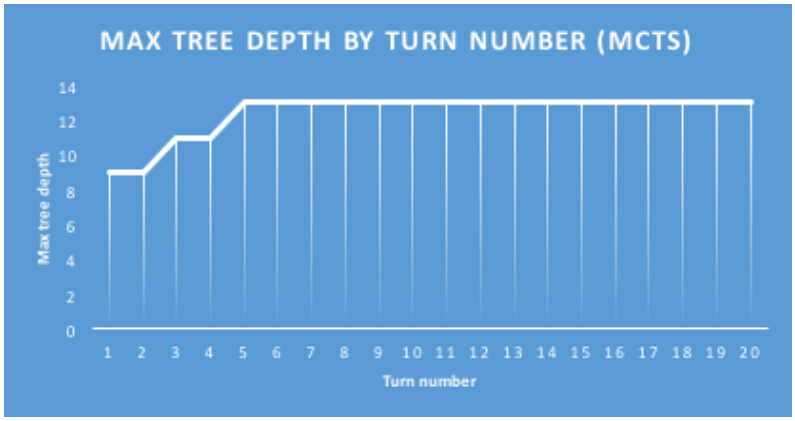
ADVANTAGES AND DISADVANTAGES OF MY APPROACH

Theoretical analysis of the game of Hus. Hus has a branching factor m≈16, with an average game duration of ~100 for two good players. **Table 1** presents the number of reachable states for depths smaller

Table 1. Number of reachable by depth

Depth	#states	Factor
0	1	-
1	24	24
2	576	24
3	11040	19
4	207443	19
5	3439088	17
6	55769962	16

Figure 1. Depth of longest path in Monte Carlo search tree



than 6. We see that the factor of ~20 at the beginning of the game limits the usability of Alpha-Beta. My implementation of Alpha-Beta pruning can just barely complete depth 6 (where depth 0 is the root depth) in under 2 seconds. On the other hand, **Figure 1** shows that Monte Carlo tree search, by its asymmetry property, can reach up to depth 13 in its longest path, thus providing a much more accurate representation of the game.

Other advantages:

- Not completely dependent on the heuristic: since Monte-Carlo tree search simulates game until the end, it gets an exact outcome and makes decision based on that;
- Anytime algorithm: as soon as new information is available, the back-propagation is performed – meaning the search tree is always up-to-date. This is especially important in our case because we have only 2 seconds to return a move: MCTS can always return a move within few milliseconds;
- Asymmetric: the search tree grows the more promising part of the tree first. With our relatively large branching factor, this is crucial in good performance;
- Monte-Carlo tree search has much more room to optimization, thus fitting the algorithm to our problem space very easily;

Disadvantages:

- We've discussed earlier how MCTS converges to Minimax at the limit. However, in 2 seconds, it is crucial to optimize every single method calls so that we get as many simulations as possible. Still, two seconds might just be too small for MCTS to converge;
- Monte-Carlo tree search has many more parameters to tune than Alpha-Beta pruning. This can be time consuming in the development phase.

OTHER APPROACHES TESTED

Minimax. In its basic form, Minimax works by playing games until the end and selecting the optimal move based on perfect information. Obviously this is too much time consuming. Enter Alpha-Beta pruning.

Alpha-Beta pruning offers to prune parts of the tree that we know won't be selected. A crucial property of Alpha-Beta pruning is its optimality. It usually makes use of a heuristic function to evaluate nodes. To increase pruning, I sort the moves before evaluating them. This has given me an extra depth, as well as greedily breaking tie. My Alpha-Beta player could complete depth 6-7 in 2 seconds.

Variants of Alpha-Beta pruning. I have tried few variants of Alpha-Beta pruning, namely transposition tables, aspiration search and Negascout. Unfortunately, I have found that there are too many distinct states for a transposition table to be efficient. Indeed, until depth 4, there are no transpositions at all and only a few starting form depth 5. Aspiration search suggests using a narrow alpha-beta window and Negascout is a variant of that: the first child of a node is evaluated with a full window. The bounds [alpha-1,alpha] are then used to search the remaining children. The idea is that if the first move was indeed the best, then the pruning is going to be optimal, resulting in an increase speed. I have not found this algorithm to increase the performance of my player.

Alpha-Beta-Monte-Carlo Hybrid. Based on my intuition after playing thousands of games, Monte-Carlo tree search performs badly at the beginning of the game and very good near the end while Alpha-Beta pruning performs fine at the beginning[2]. I have thus tried using Alpha-Beta pruning at the beginning and Monte-Carlo tree search near the end but surprisingly it didn't help improve the performance (perhaps the move selection is just too different that Alpha-Beta pruning didn't put Monte-Carlo is a good situation?).

AREAS OF IMPROVEMENTS

I believe a better heuristic function could be trained on a dataset of expert moves and then improved using reinforcement learning. A neural network could also help catch more complex features – I have actually implemented a neural network alongside linear regression but I didn't get satisfying results. Also, I have tuned to parameters with self play but it would be interesting to balance self play and playing against Alpha-Beta, as most students in the class seem to use Minimax.

I have spent most of my time on Monte-Carlo tree search, but I believe that Alpha-Beta pruning would be worth investigating more. Perhaps use more cut-off and a better heuristic function to get more pruning. Finally, an open book could help MTCS in the beginning of game[3].

---

[2] See Annex A.

[3] I have actually implemented that too, but the book is still being generated a day after the submission, while I am writing this report. Basically, I mapped states from the first four depths (pretty much the furthest I can store in a file of less than 10mb) to a move.

REFERENCES

[1]Russell, S., Norvig, P. and Davis, E. 2010. *Artificial intelligence*. Prentice Hall.

[2]Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intell. AI Games*. 4, 1 (2012), 1-43.

[3]Gelly, S. and Silver, D. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*. 175, 11 (2011), 1856-1875.

[4]A. Rimmel, *Improvements and Evaluation of the Monte-Carlo Tree Search Algorithm*, Ph.D. thesis, Laboratoire de Recherche en Informatique, France, Paris, 2009.

ANNEX A. MONTE-CARLO TREE SEARCH VS. ALPHA-BETA PRUNING

Figure a. Monte-Carlo Tree Search (up) vs. Alpha-Beta pruning (down), turn 24

MCTS is loosing by 32 seeds after 24 turns.

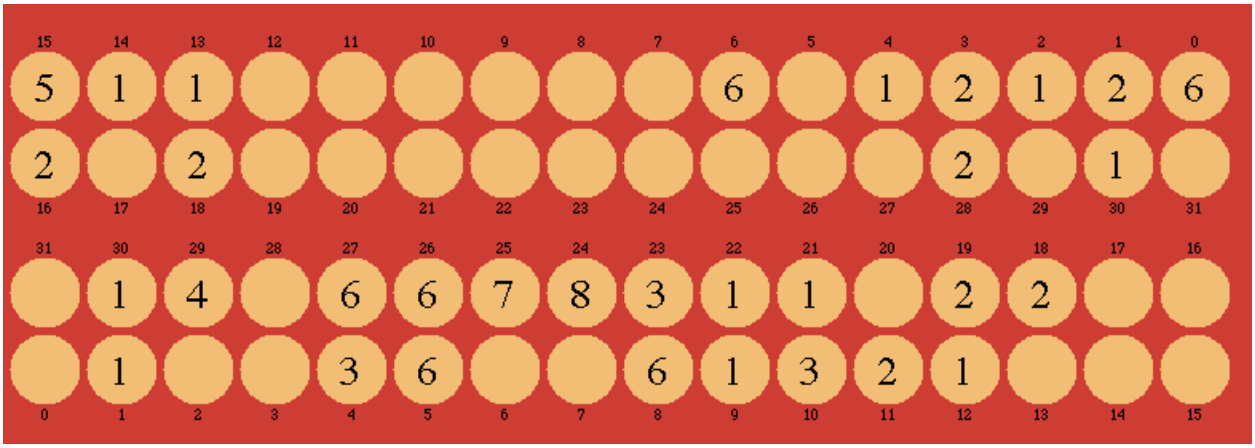
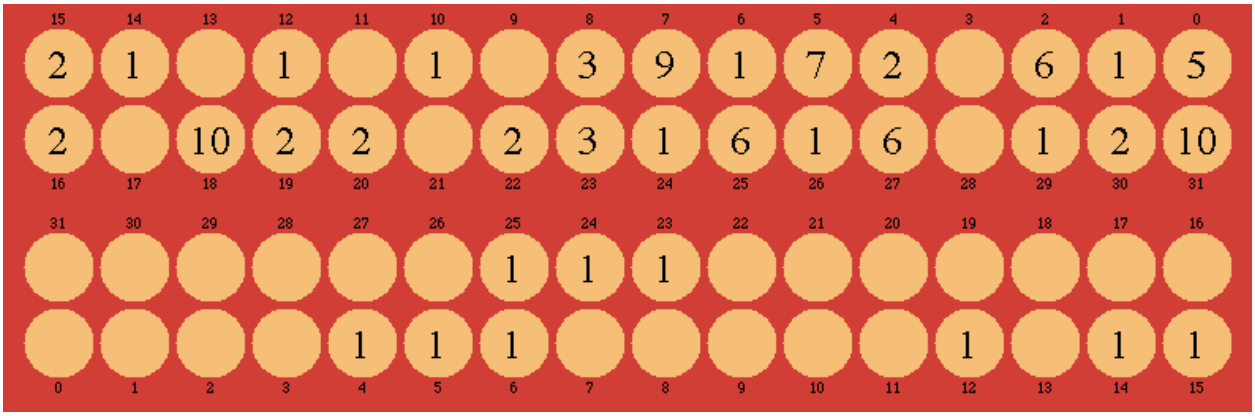


Figure b. Monte-Carlo Tree Search (up) vs. Alpha-Beta pruning (down), turn 171

Surprisingly, MCTS won by more than 120 seeds after 171 turns.



```
#!/bin/bash
# Runs simulations on a set of parameters

# How to use
#   ./run_simulation.sh NAME value1[, value2, value3, ...]
#     where NAME=name of the parameter being tested
#     value1, [value2, value3, ...] is a list of values to test
#
#   Example:
#     $ ./run_simulation.sh C .1 .01
#     >> AUTOPLAY WITH C=.1
#     >> Copying results to simulations/1457186564-C-.1.txt
#     >> AUTOPLAY WITH C=.01
#     >> Copying results to simulations/1457186564-C-.01.txt

# setup the environment
LOGDIR=logs
SIMDIR=simulations
DATE=`date +%s`
NGAMES=30
JAVA_CONSTANTS_FILE="constants.txt"

echo "Setting up the environment (creating simulations and emptying logs)"
mkdir -p simulations
rm -f $LOGDIR/*

# Grab the name of the parameter being tested
NAME=$1
shift

echo "Compiling..."
ant compile

# Iterate over every value provided in
for i
do
  shift   # (i,j) is the same as (j,i)
  for j
  do
    # Skip when param are the same
    if [ "$i" == "$j" ]; then
      continue
    fi

    echo "Writing constant value to $JAVA_CONSTANTS_FILE"
    echo "$NAME:$i" > "$JAVA_CONSTANTS_FILE"
    echo "$NAME:$j" >> "$JAVA_CONSTANTS_FILE"

    # echo "Emptying outcomes.txt"
    cat /dev/null > $LOGDIR/outcomes.txt

    echo "AUTOPLAY WITH $NAME=$i vs. $NAME=$j"
    ant autoplay -Dn_games=$NGAMES

    echo Copying results to "$SIMDIR"/"$NAME"-"$i"_vs_"$j".txt
    cp "$LOGDIR"/outcomes.txt "$SIMDIR"/"$NAME"-"$i"_vs_"$j".txt
  done
done
```

ANNEX C. SUBMISSION STRUCTURE

Directory structure:

report.pdf: final report

submission: final submission

allcode: all code used in this project

- hus
  - run_simulation.sh: script used to run simulations to tune parameters
  - src: only includes my code
    - student_player
      - mytools
        - BaseSolver.java: base class that both my Alpha-Beta and Monte-Carlo Tree Search solver extend
        - MonteCarlo.java: implementation of Monte-Carlo Tree Search
        - MCParamOptimizer.java: subclass of MonteCarlo.java that allows to load parameters from files (used in self play to tune the parameters)
        - AlphaBeta.java: implementation of Alpha-Beta pruning
        - ABMC.java: Alpha-beta-Monte-Carlo hybrid
        - BoardStateSerializable.java: extension of HusBoardState to implement serializable (used for the open book)
    - book_builder
      - Builder.java: generates open books based on some given strategy
- ML: python package containing code used to learn the heuristic function weights
  - data/: folder containing my data sets
  - main.py: package using nn.py and regression.py to train on data in `data/`
  - nn.py: implementation of neural networks
  - regression.py: implementation of linear regression, ridge regression and k-fold cross validation