On the Efficiency and Efficacy of Various Utility Functions for *The Captain's Mistress*

Cassandra Bleskachek

April 2019

1 Abstract

The Captain's Mistress (Connect Four) has been solved to a limited degree in the past, and is ripe for a new approach. The game and agents to play it were implemented in OCaml. Various utility functions for evaluating game states were explored and tested. Greedy best-first, depth-limited minimax, and depth-limited alpha-beta search applied potential utility functions to play versus random opponents and other successful algorithms. Effective functions were selected and permuted for further testing over three generations. Utility evaluation strategies included deconstructing the board into and scoring 69 sub-boards (with and without differential weights), threat analysis as described by Victor Allis, and binary search tree lookup in a table mapping 8-ply boards to best-play expected winner, among other functions.

2 Defining the Problem

The Captain's Mistress (sold under the brand name Connect Four in the United States) is an abstract strategy game for two players, in which players drop pieces into a seven-column, six-row grid, and if a player connects four of their pieces in a row, they are victorious. It is relatively simple insofar as all game pieces are identical, and there is perfect information. The size of the standard board limits the maximum branching factor of the tree representing the state space to 7 and a maximum depth of 42, although this would obviously be different on different board sizes. Nevertheless, on a 7x6 board there are 4,531,985,219,092 legal boards of play, as computed by Edelkamp and Kissmann [2]. The exponential growth of the state space makes search more than 5 turns ahead practically impossible on consumer hardware, and thus depth-limited search is necessary.

Unlike chess, The Captain's Mistress has no standardized or particularly accurate heuristic for evaluating the utility value of a non-terminal game state. so successful examples of depth-limited search of the state space are rare. Because of this dearth, most solutions to the game have been based in logically proven constraints on move choice [1] or state space representations efficient enough to

allow exhaustive search to the endgame [2]. Complete search of the state space was achieved by Edelkamp and Kissmann, but their solution is slow, not open source, and not fit to run on a laptop. Evaluating the utility of boards past a certain depth and sending their value back up the tree solves the speed problem of complete search.

A comparison was made between various utility functions, highlighting move computation time and the total win rate. Each agent was compared against an opponent choosing random moves, as well as other agents. The utility function must be fast and functional; our specification for Generation One defined functional as consistently beating a random-move opponent, and defined fast as playing an entire game against said opponent in less than 3 minutes. The goal for Generation Two became beating the successful agents of Generation One, and computing games in under a minute. Additionally, the algorithms were compared when starting first or second, to quantify the advantage/disadvantage of moving first, and identify if certain evaluation functions work better as white or black. Allis, with one of the first definitive works on The Captain's Mistress, established an assumption that the first mover has an advantage [1], but Douglas Micklich recently published a more recent paper disputing the notion, and our results seem to support Micklich [4]. Our ultimate goal was to designate a moderately accurate function for analyzing the utility value of any given board of The Captain's Mistress, so as to enable search-based agents that doesn't rely on approximations and workarounds.

3 Previous Work

Game-playing algorithms, as a subject, have been well-trodden. Most simple games have been solved many times over, and The Captain's Mistress (also known as Connect Four) has been weakly solved for decades. The game state is deterministic and fully observable, simplifying the implementation of an algorithm. The state space is a tree with branching factor 7 and maximum depth 42, making a simple endgame search impossible in real-time. Stefan Edelkamp calculated that there are 4,531,985,219,092 legal boards of play, thus postponing a strong solution until 2017 when he published a paper describing a set of state space compression techniques that make choosing moves from a lookup table possible [2]. Prior to Edelkamp's work, the most complete solution to The Captain's Mistress was from Victor Allis, as previously mentioned.

In 1988, Allis published his masters thesis on Connect Four, designing a strategy program named VICTOR [1]. Based on the ideas of Claude Shannon, VICTOR produced nine "rules", in his words, for move choice that could ensure a black win, given white's first move was not in the center. These rules rely heavily on the control of the zugzwang, a German word for the ability to choose the opponent's moves through forced play. His program can always win if it gets to go first, and uses the database he had of over 500,000 board positions. He states that this is the strongest solution possible, but this conclusion relies on the premise that the first player has an irreversible advantage, which is contradicted

by his own strategic rules, which can be used to "steal" the zugzwang from the opponent. The existence of a first player advantage at all has also been called into question by other academics [4], and a stronger solution has been achieved using modern supercomputers and data structures [2].

Allis provides detailed analysis and proofs for each of his rules, but they do not provide particularly strong solution to the game. The rules are described in terms of a precondition pattern on the board, a possible place to drop a piece, and which of the 69 possible solutions (winning row of four) the piece could be included in. Before play, a directed graph of the actions is made, with best actions chosen through depth-first traversal of a search table. Although Allis' strategy is well-crafted and seminal to literature on The Captain's Mistress solvers, his solution was weak and complex. Multiple implementations of VICTOR exist; Velena is a defunct example, but it was rebuilt as part of the game-solving program DAna, among others.

In 1995, John Tromp used 40,000 hours of supercomputer processing to produce a publicly available data set of 67,557 8-ply boards (all positions after exactly 4 turns where the next move is not forced) and the expected winner of each game, assuming best play on both sides [5]. This database provides endgame simulacra; state-space search algorithms can find the utility value of an 8-ply board more accurately and quickly through searching this database than further exploration of the tree or evaluation functions.

Almost 30 years after the development of VICTOR, Stefan Edelkamp and Peter Kissmann demonstrated the use of binary decision diagrams and perfect hashing to compress the state space representation enough to produce a strong solution to the game [2]. While Allis' thesis approached the problem using clever inference and constraint propagation to strategically evaluate plays, Edelkamp and Kissmann used advances in data efficiency and processing power to enable a full-depth search of the state space.

4 Algorithms and Implementation

As of yet, the only successful state evaluation used in published works for The Captain's Mistress relies on minimax endgame search, which is hardly apt for quick evaluation of midgame boards. Developing alternative fitness functions would seem to better advance the field of computer analysis on the game than retreading another endgame-search or rule-based solution. This project developed multiple approaches for calculating utility, and permutations of each utility function, as well as a handful of search functions to test the evaluations and filter potential moves. To that end, a shell-based version of The Captain's Mistress was implemented from scratch in OCaml, including board analysis and agents to play against the user or each other. Greedy search, minimax search, and alpha-beta pruning were all implemented, as well as 10 utility functions to evaluate boards once the maximum depth has been reached.

In addition to the code, the 46,000 board database of 8-ply boards published by John Tromp was downloaded, sorted, and converted into a new binary

tree format for reference and comparison against our computed utility values. Tromp's database of 8-ply boards and their expected best-play outcome can be treated as an endgame for the purposes of state-space traversal. Normally, searches without a depth limit will continue expanding children until they reach a win, loss, or draw (maximum depth 42). However, using binary tree search in a sorted permutation of Tromp's data to determine the utility value of 8-ply board, rather than running an evaluation function or expanding 34 layers of children, saves significant processor time and produces results virtually equivalent to brute force search.

Cells on the board and players were represented by a disjoint type of Blank, P1, or P2. The board was represented as a 42 length array of spots, grouped by column, thus spot (x,y) was stored in array.(6*x + y). The code included the following (function names are stated for functions without related helper functions):

- game.ml:
  - Conversions between coordinate-based and index-based board representations
  - printBoard: printing the board
  - Determining the current ply and player based on the number of pieces played
  - isFull: checking if the board is full
  - Checking the result of a potential move
  - posMoves: determining the list of possible moves
  - winLines: extracting the spot values of the 69 lines where a win is possible from a given board
  - gameOver: determining if the game is over, whether it was a draw, and the winner
  - randomSelect: making a random legal move
- util.ml
  - A win for the first player is represented as 100, and a win for the second player as -100.
  - Functions for processing data including the following:
    - Manipulating files
    - Converting data between Tromp's format and the data structure used in this project
    - Board comparison and sorting
    - utilSortTromp: binary tree search to find the winner Tromp showed for a given 8-ply board

  - These functions traverse the 69 possible lines on which a player can win:
    * willLose: determining if there is a line with 3 opponent tokens and a blank token
    * utilLines: default board value 10; each line with no tokens or tokens from both players does not adjust the value; each line with three tokens from player one adds 40 and three tokens from player two subtracts 40; each line with two tokens adjusts the value 9
    * utilDots: default board value 10; each line with no tokens or tokens from both players does not adjust the value; each line with three tokens adjusts by 35; each line with two tokens adjusts the value 8; each line with one tokens adjusts the value by 0.5
    * utilHeightLines: same computations as utilLines, with each line weighted more strongly the lower it is
    * utilHeightDots: same computations as utilDots, with each line weighted more strongly the lower it is
    * utilWeightLines: same computations as utilLines, with each line weighted more strongly the closer to the center it is
    * utilWeightDots: same computations as utilDots, with each line weighted more strongly the closer to the center it is
    * utilHWLines: same computations as utilLines, with each line weighted for both the height and proximity to center
    * utilHWDots: same computations as utilDots, with each line weighted for both the height and proximity to center
    * utilCowardLines: same computations as utilLines, with opponent success weighted 1.5x as strongly as personal success
    * utilCowardDots: same computations as utilDots, with opponent success weighted 1.5x as strongly as personal success
    * Partial implementations of MonteCarlo tree search

- search.ml contains the search algorithms and introduces the Player signature, which has functions toString and nextMove. Modules following the Player signature type include:
  - RandomPlayer: chooses random legal moves
  - Human: takes input from the user to choose moves
  - GreedyXXX: calls greedy best-first search with XXX utility function
  - MinimaxXXX: calls depth-limited minimax search with XXX utility function at depth 5
  - AlphabetaXXX: calls depth-limited alpha-beta search with XXX utility function at depth 5

- play.ml defines interactions between players in the Play(player1)(player2) functor. Functions include:
  - History functions: unused, allow the player to look at the history of moves made during the game
  - quickGame: runs a game between the players and prints the outcome
  - gameLoop: similar to quickGame, but prints the board at each ply
  - tournament: runs a set of 20 games, 10 with each player starting
- test.ml is the file used to run this program from the interpreter, although main.ml contains the testing command for when it is compiled. The code in test.ml includes:
  - 5 boards chose from random points in Tromp's dataset, used for comparison against the utility functions
  - testUtility: runs the given function on the five boards, prints the results
  - vsRandom: runs 30 games with the given agent as first player against RandomPlayer, then 30 as second player. The average CPU time spent running each game and the win-loss-draw record are printed.
  - Test.vs: runs the two given agents against each other, and prints the same information as vsRandom
  - tests: runs the tests specific to the current generation

5 Experiment Design and Results

The code and agents can be divided into generations; Generation One only included the encoding of the game itself, greedy best-first, and the first four utility functions. During Generation Two, we added depth-limited minimax search and depth-limited negamax search, but later collapsed them into a single function. Generation Two also included the other six utility functions, and the first round of competent algorithms. Generation Three included depth-limited minimax search with alpha-beta pruning and code optimizations.

To be successful, an agent must make moves quickly and strategically. Our specification for Generation One defined functional as consistently beating a random-move opponent, and defined fast as playing an entire game against said opponent in less than 3 minutes. The goal for Generation Two became beating random at 95% of the time, beating the successful agents of Generation One, and computing games in under 30 seconds. By Generation Three, certain algorithms were winning 100% of games, even against other successful algorithms, and all agents were finishing games in under one second.

Utility functions were tested on the five boards selected from Tromp's dataset (the 1st, 10,000th, 30,000th, 50,000th, and 65,000th entries), and all algorithms maintained the same sign as Tromp's proven outcome, but the magnitude varied

widely. Nevertheless, this data is hard to interpret without context, so we played agents against each other. Each "match" between two players included a "set" of 30 games with one player as first player, and 30 games with the opposite. The first two generations were tested against RandomPlayer, and all 10 Generation Three alpha-beta agents were also tested against each other. Generation One certainly fulfilled the speed requirement (t, but many of the algorithms had a win rate of only 70-80% against a random-move-selector. By Generation Three, every agent pair completed their average game in less than 0.25 seconds, and all sets were won completely by a single player; no more fair competition remained. All test results are included with the code. The test results for Generation Three were as follows:

Generation Three Alpha-Beta Data:
For the same algorithm against itself, the combined score is shown.
(Top: first player; Side: second player; Cells: Win-Loss-Draw)

	Lines	Dots	HLines	HDots	WLines	WDots
Lines	30-30-0	30-0-0	30-0-0	30-0-0	30-0-0	30-0-0
Dots	0-30-0	0-0-60	0-0-30	0-0-30	0-0-30	0-0-30
HLines	0-30-0	0-30-0	30-30-0	30-0-0	30-0-0	30-0-0
HDots	0-30-0	0-30-0	0-30-0	30-30-0	30-0-0	30-0-0
WLines	0-30-0	30-0-0	0-30-0	0-30-0	30-30-0	0-30-0
WDots	0-30-0	0-30-0	0-30-0	0-30-0	30-0-0	0-0-60
HWLines	0-30-0	0-30-0	0-30-0	0-30-0	0-30-0	0-0-30
HWDots	30-0-0	0-30-0	0-30-0	0-30-0	30-0-0	30-0-0
CLines	0-0-30	0-0-30	0-0-30	30-0-0	30-0-0	30-0-0
CDots	30-0-0	0-30-0	0-30-0	0-30-0	0-30-0	30-0-0

	HWLines	HWDots	CowardLines	CowardDots
Lines	30-0-0	30-0-0	30-0-0	30-0-0
Dots	0-0-30	0-0-30	0-0-30	0-0-30
HLines	30-0-0	30-0-0	30-0-0	30-0-0
HDots	30-0-0	30-0-0	30-0-0	30-0-0
WLines	0-30-0	0-30-0	0-30-0	0-30-0
WDots	0-0-30	0-0-30	0-0-30	0-0-30
HWLines	30-30-0	0-30-0	0-30-0	0-30-0
HWDots	30-0-0	30-0-0	0-30-0	0-30-0
CLines	0-0-30	30-0-0	30-30-0	30-0-0
CDots	0-30-0	30-0-0	0-30-0	0-0-60

6 Analysis of Results

The results of this experiment were promising, and exceeded the goals set for each generation. The end of Generation Two was the first set of games not

using random opponents, and the results were much more consistent for obvious reasons; the score of every set between the alpha-beta agents was 30-0-0, 0-30-0, or 0-0-30. Even the same algorithm playing against itself favored one player, the controller of the zugzwang, but the player who controlled it was different for each function. When playing against themselves, the first utilHW-Lines agent to move won every game, utilDots forced a draw every game, and the utilLines moving second always won. There is also a rock-paper-scissors mechanism at play between the algorithms; for instance, utilDots beat utilLines, utilLines beat utilCowardDots, and utilCowardDots beat utilDots. This relation between algorithms gives them dynamic value; certain algorithms are poor in the general case but useful for beating specific strategies, and others perform well in general but can be easily fooled. Therefore, no specific utility evaluation function is ideal for every situation, but an "approximate best" can be determined; the function that beat the most other functions was utilWeightDots. The only function against which utilWeightDots could not at least force a draw was utilWeightLines.

Contrary to Allis' presumption, the second player is not at a definitive disadvantage. In fact, more algorithms won as the second player than as the first. When the same successful algorithm played against itself, the second player was usually the winner, because they "control the zugzwang". When playing the same pair of agents against each other in both play orders, some algorithms performed optimally as the second player, and some performed better going first. The necessity of situational strategy makes The Captain's Mistress exceedingly interesting to study, and makes game-solving approaches using a single utility function particularly tricky to optimize.

7 Conclusions and Future Work

The majority of the time and effort we devoted to this research was concentrated on developing the basic game implementation and the first generation of agents. After that portion was complete, generations Two and Three were rather quick to produce, and made significant improvements on Generation One. Further iterations and mutations could be swiftly implemented and still advance the study of utility values for The Captain's Mistress. Improving Generation Three's results would be the obvious next step, but not the only one. Ordering the possible moves returned by posMoves so that central columns are first in the list would increase the amount of pruning done by alpha-beta searches, therefore improving the overall speed.

Utilizing Tromp's dataset in the agents themselves, rather than as a comparison, would also decrease search time. We partially implemented an evaluation function that relied on searching to either Tromp's data or the endgame, treating them as equivalent, insofar as they have set utility values that can be passed back up the tree. Although this function on its own would not be very useful, treating Tromp's dataset as the end of the "opening subtree" would allow all other depth-limited utility functions to search the dataset for a quick and proven

best-play utility value, improving speed and accuracy of the first 4 moves. However speed is not necessarily a large factor; any of the alpha-beta pairings can play through thousands of games in an hour, so functionality should also be expanded.

Another feature we partially implemented would approximate the rules Allis laid out for his program VICTOR. A complete replication of his program would be simplest using a language more suited to that task, such as the logic solving language Potassco maintains known as clingo [3]. The original VICTOR, and the recreation of it by Giuliano Bertoletti called Velena (which is no longer available on the web) were both done in C, based on the Shannon-type game solving approach. The modern DAna software, which includes a copy of Velena, is written in C++; no implementations of VICTOR are based in functional programming. Finishing our recreation of VICTOR in OCaml would be a worthwhile expansion to the types of solvers for the game. In addition to the rules-based agent, Allis' rules could be used as heuristics and combined to evaluate board utility, so traditional search algorithms might incorporate Allis' analyses. Even independent of his nine major rules, Allis' ideas of zugzwang and even-odd threats would still be helpful to implement as the bases of utility functions.

While the focus of this work was utility evaluation functions and the advantage of going first, general strategy is also important to game-playing agents. Knowing this, a strong set of general tactics used by each agent would be a smart addition. Such a strategy would be applied before search, and if no move is determined by the strategy, search would proceed. Examples of tactics include going in the center on the first turn, taking center squares over edges and corners, taking corner squares over edges, etc.

While there is more work to be done on utility functions for The Captain's Mistress, we have found an effective utility function for the game. The depth-limited minimax search with alpha-beta pruning using utilWeightDots outperformed almost every other algorithm, and computed turns in less than 0.2 seconds. This result exceeds the goal set, and thus our work comes to a close.

8 References

References

[1] Victor Allis. A Knowledge-based Approach of Connect-Four. Master's thesis, Department of Mathematics and Computer Science, Universiteit Vrije, 10 1988.

[2] Stefan Edelkamp and Peter Kissmann. Bdds for Minimal Perfect Hashing: Merging Two State-Space Compression Techniques. University of Bremen, 4 2017.

[3] M Gebser, R Kaminski, B Kauffman, M Ostrowski, T Schaub, and M Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications: The European Journal on Artificial Intelligence (AICOM)*, 2011.

[4] Douglas L. Micklich. Do First Mover advantages exist in competitive board games: The Importance of Zugzwang. *Developments in Business Simulation and Experiential Learning*, 36:270–274, 2009.

[5] John Tromp. Connect-4 Data Set. UCI Center for Machine Learning and Intelligent Systems, 2 1995.