Implementing an AI player for the game Odd

Mouhyi Eddine El Bouhali

Introduction

The purpose of this project was to implement an AI player for the game 'ODD' [1]. We present and discuss the effectiveness of the different approaches that were tried in order to implement a powerful AI player. We first describe the minimax family algorithms: AB Negamax, NegaScout and MTD(f). Then we discuss the Monte Carlo algorithms (MCTS, UCT, UCT-RAVE) and provide experimental results to compare the different AI agents.

Note: The algorithms are described briefly here and the reader is encouraged to refer to the source code for a detailed implementation.

1. Game Analysis

One deduces the following properties by analysing the rules of Odd [1]:

- Board game (board size = 61).
- Two-player turn based game.
- Zero-sum game: one's win implies his opponent's loss.
- Perfect information : both players know everything about the state of the game.
- Deterministi
- State space:
  Each board cell has 3 possible values: white stone, black stone or empty. Therefore we have | State space | = | board positions | = 3^61 ~ 10^29. Since the state space is exponential, it is impossible to store it entirely neither during the game nor for reinforcement learning. Hence, we need to find an appropriate set of features for the board positions.
- Branching factor:
  The branching factor = number of valid moves at each board position.
  It starts at 61*2=122 and decrements by 2 after every move.
- Depth of the search tree = total number of moves = 61.
- Transposition: it is possible to reach the same board position from different sequences of moves.

These properties influenced the choice of the AI approaches discussed in the following sections.

2. MiniMax Player

The first approach that was tried is the use of a minimax search strategy.

2.1 Alpha Beta Negamax

Since the game is a zero sum two player game. If a player gets a reward −1, then the opponent should get +1. This property of 'Odd' is used to simplify the classic alpha beta minimax algorithm. At each recursion stage, instead of choosing either the min or max value, all the scores from the previous stage have their signs inverted and the maximum score is chosen. Moreover, instead of alternating checks against alpha and beta at each level, the AB negamax swaps and inverts the signs of the alpha and beta values . It only uses the beta value to prune[2].

Static evaluation Function

The first minimax agent implementation used a simple heuristic H0 that gives 0 for internal nodes, +100 for winning terminal nodes and -100 for losing nodes. Integer values were used because integer arithmetic is faster than floating point arithmetic on most machines [2]. This simple heuristic was used as a benchmark to compare the different minmax AI agents. Improved heuristics are discussed in later sections.

Performance

The algorithm is O(d) in memory, where d is the maximum depth of the search, and O(bd) in time, where b is the branching factor. [2]

The Alpha Beta Negamax achieved a 60% winning rate against the Random Player over 1000 simulations.
The algorithm provides a framework for an AI agent. The next sections describes the different enhancements that were tried.

2.2 NegaScout

The AB minimax Search Window

The search window is defined as the interval [α , β]. The window is initialized to $(-\infty, +\infty)$ in an AB minimax search. Minimax prunes all the nodes outside this window. Therefore, a narrower window results in pruning more nodes and a better performance. The null search window (α = β -1) is the narrowest window causing the most cutoffs [2].

Aspiration search uses the idea of an estimated search window based on a previous search to run the AB minimax. Knuth and Moore describe the behavior of the aspiration search [3, 4]:
g = ABminimax (board, maxDepth, α , β);

- if α < g < β then g is the minimax value of the game tree.
- if g ≤ α (failing low) then g is an upper bound on the minimax value
- if g ≥ β (failing high) then g is a lower bound on the minimax value

The algorithm

We used the concept above to implement an AI agent using the Negascout[5] algorithm. It is an approach that combines minimax and the Scout algorithm[6]. Negascout works by performing a search of the first move with an open window so that the search does not fail. Then it searches the remaining moves using a scout pass (null search window based on the score from the first move)[2].

The NegaScout Player, with the static evaluation function H0, achieved a 75% winning rate against the Random Player over 1000 simulations.

2.3 MTD(f)

2.1 Transposition Table

'Odd' has the Transposition property: it is possible to reach the same board position from different sequences of moves. The previous algorithms do extra work re-expanding previously searched nodes. To avoid this, we decided to use a Transposition table (hash table) to store searched states in memory. When the algorithm is examining a board position, it first looks it up in the table to use its stored value if possible [2].

Minimax trees are exponential in size. However, it is shown that the AB minimax search tree can be stored in memory [4].

An issue that arose was that hash tables require state comparison that can often be expensive. Therefore our hash table implementation uses the Zobrist hashing [7] technique to make the state comparison more efficient. We also encode the "current player" as part of the state, given the two player property of the game.

2.2 The algorithm

We used the idea of the Transposition table to implement an AB minimax agent with memory. The algorithm is known as MTD(f) (Memory-enhanced Test Driver with node *n* and value *f*). MTD(f) simply uses a zero-width AB minimax search and a transposition table to store the search results. Each AB minimax search returns a bound on the minimax value. The bounds (upper or lower) are then stored to estimate an interval where the minimax value lies [8].

2.3 Iterative Deepening

Out MTDPlayer uses an iterative deepening MTD(f). This is appropriate for the time constrained nature of the game: when the time runs out the algorithm returns the best move from the previous search. Moreover, the algorithm uses the values from the previous shallower searches as guesses for MTD(f) [2].
The MTDPlayer, described above, achieved a 85% winning rate against the Random Player using the static evaluation function H0, over 1000 simulations.

2.5 Static Evaluation Function

At this point, We have developed a fairly sophisticated minimax algorithm. However, with the winning rates against the random player being unsatisfactory, the only solution is to improve the static evaluation function.

Different ideas were considered:

- Reduce MIN_CLUSTER_SIZE from 5 to 3 or 4 and evaluate the state as if it were terminal state.
- Allow empty board cells to be part of the clusters (for both colours)and evaluate the state as if it were terminal state.
- Merge clusters of the same colours if the number of the disjoint paths made of empty/same colour cells between the two clusters exceeds a threshold value. Then evaluate the state as if it were terminal state.
- Run a Monte Carlo simulation to get an estimate of the state value.

The last idea was implemented and provided a winning rate of 100% against the random player over 1000 simulations. However, it suffered from the horizon effect and had a very small search depth at the beginning given the high branching factor.

Possible ways to deal with this is to use quiescent search to mitigate the horizon effect, find a better static evaluation function function, or use a history heuristic to improve the move ordering and hence the cutoffs in the algorithm.

However, the difficulty of designing a good static evaluation function and the success that resulted from using a Monte Carlo simulation encouraged the exploration of a Monte Carlo approach for the game. The next section describes the AI players implemented based on Monte Carlo methods.

3. Monte Carlo Player

3.1 Greedy MCST

MCTS (Monte Carlo Tree Search) offers a number of advantages for this particular game[9]:
Aheuristic: whereas an accurate static evaluation function is needed for the minimax algorithms, MTCS doesn't need any game specific knowledge.
Asymmetric tree growth: the MTCS tree selection policy explores the more promising states.
Graceful Exit: the algorithm can terminate gracefully when the times runs up and return the best move found so far.

Our first implementation of MCTS, uses a greedy tree policy with respect to the estimated value $Q(s,a)$ which is the average of all the simulations where action a was selected in position s [11]:

$$Q(s,a) = \frac{1}{n(s,a)} \sum_{i=1}^{N} I_i(s,a) z_i, \qquad (1)$$

[11]

where z_i is the outcome of the ith simulation: z_i = 1 if the player wins the game and z = 0 otherwise. $I_i(s.a)$ = 1 if move a was selected in state s during the ith simulation, and 0 otherwise.

We also use a random MC default policy. MCTPlayer achieved a 100% win rate against the Random player and a 80% win rate against the MTDPlayer on 1000 simulations against each opponent.

3.2 UCT (Upper Confidence Trees)

The next enhancement was to improve the Monte Carlo tree selection policy, for that we tried UCT[10]. UCT [10] treats every decision in the game as a multi-armed bandit problem. The tree policy was enhanced by balancing Exploitation Vs Exploration. The UCT tree policy is greedy with respect to the Upper Confidence Bound:

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}. \qquad (2)$$

where n(s) counts the total number of visits to position s, and c is the exploration constant.

A hill climbing approach was used to tune the parameter c, the optimal value was found to be $c = 0.4$.

UCTPlayer achieved a 100% win rate against the Random player and a 70% win rate against the previous MCTPlayer on 1000 simulations against each opponent.

3.3 UCT-RAVE

All-Moves-As-First Heuristic

The all-moves-as-first (AMAF) value $\tilde{Q}(s,a)$ is the average outcome of all simulations in which action a is selected at any turn after encountering the state s.

$$\tilde{Q}(s,a) = \frac{1}{\tilde{N}(s,a)} \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a) z_i,$$

[12]

where $\tilde{I}_i(s.a)$ = 1 if state s was encountered at any step t of the ith simulation, and action a was selected at any step $u \geq t$, or 0 otherwise; $N(\tilde{s,a})$ is the number of simulations where s was encountered [12].

This heuristic shares knowledge between different states at the expense of introducing a bias in the estimations. [11]

Combining UCT and RAVE

The RAVE (Rapid Action Value Estimation) algorithm uses the AMAF value $\tilde{Q}(s,a)$ to estimate the value of an action a in state s. We combined the RAVE technique and the previous UCT algorithm by contributing both the of the MC value $Q(s,a)$ and the AMAF value $\tilde{Q}(s,a)$ with different weights to estimate the value of an action a in state s [12] :

$$Q_\star(s,a) = (1-\beta(s,a))Q(s,a) + \beta(s,a)\tilde{Q}(s,a) \quad [12]$$

where $\beta(s,a)$ is a parameter defined as follows:

$$\beta(s,a) = \sqrt{\frac{k}{3N(s)+k}} \quad [12]$$

We can see that early in the algorithm run, when the number of simulations is small, $\beta(s,a) \simeq 1$ and the AMAF value has more weight; whereas when the number of simulations goes high $\beta(s,a) \simeq 0$ and the MC value has more weight [12].
Again, using a **hill climbing** approach was used to tune the equivalence parameter k, the optimal value was found to be $k = 50000$.

UCTRAVEPlayer achieved a 100% win rate against the Random player and a 68% win rate against the previous UCTPlayer over 1000 simulations against each opponent.
Moreover, UCTRAVEPlayer achieves a win rate of 99% against a beginner human player.

Improving the MC default policy

At this point, the MC tree policy using UCT and RAVE was performing very well, so we tried to improve the default policy. In our UCTRAVE, the moves beyond the search were played randomly. Here we tested combinations of the following heuristics as default policies:

- The Proximity Heuristic : choose a move within a fixed distance from the last move played [13].
- The First-Order History Heuristic : this is a history heuristic that favors the moves that were chosen more frequently earlier by the algorithm [13].
- The grandfather heuristic : this heuristic assumes that the value of a state is close to the value of it's grandfather (i.e last move played by the current player) [11].
- The even-game heuristic, which assumes $Qh(s,a) = 0.5$ for non terminal states [11].
- Domain specific heuristics to 'Odd' (see 2.5).

Different combinations of the heuristics above were tested, but the UCTRAVE agent with a random default policy outperforms all the heuristic agents. This might be due to the fact that a non random default policy wastes time in the MC simulation phase that could be used in running more simulations.

4. Conclusions and Further Improvements

Several AI players were tested and compared in this project, the Monte Carlo family of algorithms outperforms the minimax family due to the difficulty of finding a good heuristic. UCTRAVE was found to be the best player implemented (see Table 1).

AI Player	Rank
UCTRAVE (k=50,000)	1
UCT (c=0.4)	2
MCTS	3
MTD(f)	4
Scout	5
AB Negamax	6

Table 1: Comparison of the AI players

Further improvements include the use of reinforcement learning by designing a feature set and using function approximation. Another improvement could be the use of the opponent's turn as thinking time. Finally, the aspect that still needs major enhancements is the heuristics used by the AI players.

References

[1] Nick Bentley, http://nickbentleygames.wordpress.com/2010/02/11/one-of-my-better-games-odd/
[2] Ian Millington and John Funge. Artificial Intelligence for Games, second edition. 2009.
[3] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. Artificial Intelligence, pages :293-326, 1975.
[4] Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie de Bruin. An Algorithm Faster than NegaScout and SSS* in Practice, 1995.
[5] Alexander Reinefeld. An Improvement to the Scout Tree-Search Algorithm. ICCA Journal, Vol. 6, No. 4, pp. 4-14, 1983.
[6] Judea Pearl . Scout: A Simple Game-Searching Algorithm with Proven Optimal Properties. Proceedings of the First Annual National Conference on Artificial Intelligence, 1980.
[7] Albert Lindsey Zobrist, A New Hashing Method with Application for Game Playing, Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1969.
[8] Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie de Bruin. A New Paradigm for Minimax Search. Department of Computer Science, The University of Alberta, Edmonton, Alberta, Canada, 1994.
[9] Cameron Browne, http://www.cameronius.com/research/mcts/about/index.html 2013.
[10] L. Kocsis and Cs. Szepesvári. Bandit based monte-carlo planning. In Machine Learning: ECML 2006, number 4212 in LNCS, pages 282–293. Springer, 2006.
[11]Sylvain Gelly, David Silver. Achieving Master Level Play in 9 × 9 Computer Go, 2008.
[12] Sylvain Gelly, David Silver: Monte-Carlo tree search and rapid action value estimation in computer, 2011.
[13] Peter Drake, Steve Uurtamo: Heuristics in Monte Carlo Go, 2007.