COMP 424 Final Project Report

Cailean Oikawa, ID 260836366

April 14, 2021

Upper Confidence Tree Search with No Unforced Loss Heuristic in Computer Pentago Twist

In their 2006 paper, *Bandit based Monte-Carlo Planning*,Levente Kocsis and Csaba Szepesvari introduce a new algorithm for performing search on Monte-Carlo search trees.

For large state-space Markovian Decision Problems Monte-Carlo planning is one of the few viable approaches to find near-optimal solutions. In this paper we introduce a new algorithm, UCT, that applies bandit ideas to guide Monte-Carlo planning. (Kocsis & Csaba, 2006)

The algorithm they present, UCT, is the basis for the Pentago Twist algorithm written for this final project. UCT is an improvement on a standard Monte Carlo Search Tree that leverages the notion of confidence intervals from bandit problem planning to justify selections of nodes in a search tree by treating each node in the tree as a k-armed bandit with k children. UCT aims to expand nodes based on an estimation of current value plus a fraction or multiple of the standard deviation associated with a nodes current value. The UCT is expanded by repeatedly applying this tree policy to select node children in the tree until a leaf is found, when a leaf is reached a default policy (typically random) rolls out the game to a terminal state and propagates the result back up to the root along the path it descended the tree. The expression Kocsis and Csaba present of UCT tree policy is typically the following

$$Q^*(s,a) = Q(s,a) + c\sqrt{\frac{log(N(s)}{N(s,a)}}$$

Here, c is an exploration constant, often root 2. The first term in the sum is the expected payoff of visiting the node based on it's current value and the second term in the sum is the expected payoff of exploring the node further.

UCT has outperformed minimax search with alpha-beta pruning in some games with especially high branching factor or with imperfect information such as Scrabble, Go, and computer RTS games such as Starcraft. The promising results of the UCT algorithm in the past and the lack of a need for expert information on the domain make it an excellent candidate for computer Pentago Twist which has a branching factor significantly higher than chess or checkers, two games where minimax is still the preferred algorithm.

As implemented for this project, Pentago Twist is a game between two players on a 6x6 board, where each turn a player either black or white places one of their pieces on any square and selects a 3x3 quadrant to rotate clockwise or flip across the vertical. White is given a first mover advantage and the game is guaranteed to terminate after 36 plays or 18 turns the advantage to white is fairly large, for random play white it expected to win about 55 percent of the time. The minimum branching factor is 8 and the maximum is 288. An efficient implementation only considers non-duplicate transition states of which on the first move there are only 36 and throughout the game an average of approximately 90 so the branching factor an agent must contend with is 90 to a depth of 36 plays. This makes the size of the state space on the first move approximately 2.25e70.

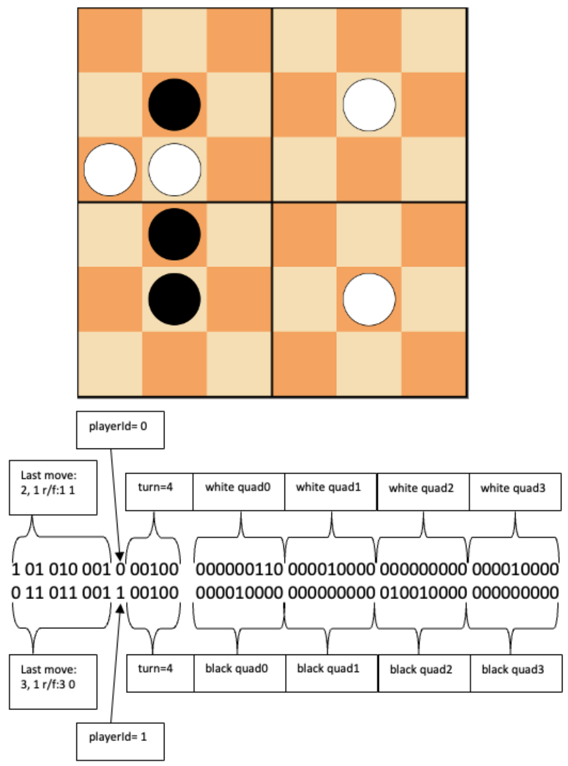
Algorithm Implementation

The following is a presentation of major features of the implementation of the Upper Confidence Tree with No Unforced Loss Heuristic. At the time of submission the algorithm plays very well against human opponents and it remains to be seen how it compares to other AI agents. Details of interest include the implementations impressive ability to search over 10 million game positions within allotted time and space capacities. Given the right position it can even find forced losses or wins up to 5 plays away which implies that out of around $90^5 = 6$ billion positions to search 99.8% of positions are correctly resolved and pruned.

Internal Representation

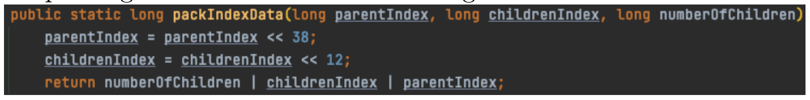
The constraints on the algorithm are that it must select each move in under 2 seconds and it cannot exceed 500mb of memory. In order to address the memory constraint it is important to use as few bits as possible to represent here we demonstrate a representation of board position equal to 128 bits.

For each square on the board we need to know if there is a white piece, a black piece, or no piece. Two bits is too few to represent one of three options so we require at least two bits per square, this gives us 36x2 bits to represent pieces. It is also important to keep track of turn number which requires 5 bits, playerId which requires 1 bit, and the last move played for each player which can be represented using 9x2 bits by the schema below.



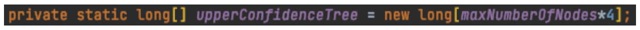
This technique of representation is often called a bit board and is very popular among top chess playing agents to represent board position. In most programming languages the bitwise operations are very fast since there exist CPU instructions to carry them out directly on hardware. Many of the operations that typically require some nested loops iterating over a collection (such as checking win conditions) can be implemented using single bitmasks and by checking for equality between 64 bit integers, in java these require only a handful of clock cycles.

An additional 128 bits are used to encode information about tree nodes. Here is shown a short method for packing index data into a 64 bit long.



```
public static long packIndexData(long parentIndex, long childrenIndex, long numberOfChildren)
    parentIndex = parentIndex << 38;
    childrenIndex = childrenIndex << 12;
    return numberOfChildren | childrenIndex | parentIndex;
```

To avoid wasting time on memory allocation the tree is represented as a large array and allocated into static memory instead of dynamically requesting 256 bit chunks.



```
private static long[] upperConfidenceTree = new long[maxNumberOfNodes*4];
```

One of the most expensive operations we perform are the evaluations of node expected exploration payoffs

$$c\sqrt{\frac{log(N(s)}{N(s,a)}}$$

It is very important to cache the results of logarithm and square root since in practice we apply them repeatedly to the same integer numbers many times in a row for both square root and square root of logarithm.

Upper Confidence Tree

To implement the UCT a root is initialized with defaut values and no children. A recursive function visits the root node of the tree, selecting children based on the standard tree policy for UCT until it reaches a leaf that has never been visited. Here the algorithm rolls out the game from the current position by randomly moving for each player until a win or loss or draw is made. The recursive function propogates back up the stack updating each node on the descent path with the result of the rolled out game. Child nodes are generated whenever visiting a leaf node for the second time and each child is checked for winning or drawing conditions after all children have been added to the tree one is selected to visit. These steps are repeated until the algorithms runs out of time or out of memory.

No Unforced Loss Heuristic

The important heuristic on top of standard UCT implementation has logic which attempts to prevent any unforced losses. So the algorithm should only lose if all the moves it could select from it's current position were guaranteed to lose, at the same time the heuristic will never not play a winning move if one is available. This heuristic was selected because it is powerful, lightweight and very difficult for a human to beat. Each node must keep track of a minimum of 3 boolean flags although the implementation tracks 6 for easier debugging. It must track whether it's position is lost, won, or drawn according to the following recursive definition.

Note than when an agent selects a move it is selecting a position for it's opponent to play. So rational agents pick "losing" nodes whenever possible and avoid choosing "winning" ones.

1. A node is lost if it's position is lost
(opponent has 5 in a row).

2. A node is lost if all it's children are won
(nowhere to run nowhere to hide).

3. A node is won if it's position is won
(it has 5 in a row).

4. A node is won if at least one of it's children is lost
(rational agents will select free win).

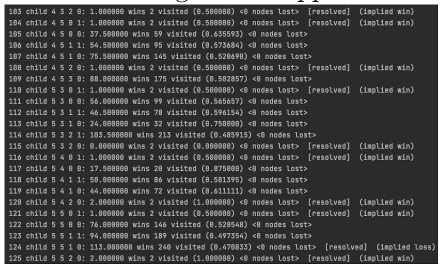
5. A node is drawn if both players have winning position or if no empty squares remain.

It is easy to see that this simple recursive definition can lead to both performance increase and better play. We set the evaluation for any resolved node to negative infinity so that time is not wasted rolling out decided nodes. And when choosing a node the algorithm is guaranteed to select an opponents losing node if one exists and will not select an opponents winning node unless all of opponents nodes are winning. After the tree is filled from the root a losing or winning node might be the result of thousands of computed lines of play throughout many levels identifying a way to force a win or for the opponent to win.

It is important to add logic which will select the ”best” losing move when all moves are losing so that an opponent is not handed a potentially undeserved win that would require very precise play to achieve. If all positions are losing the algorithm chooses the one that was visited most frequently and thus requires the most computation to find a win for.

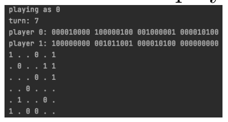
This heuristic is computationally and spatially very cheap, in terms of memory; only 3 bits per node are required. In terms of clock cycles, checking a position is winning can be performed with 32 bitwise AND operations per player against 32 different bitmasks of all possible 5 in a row configurations. This is done only once when a node is generated by it's parent and parents identify when for instance all children are winning when it is visited and the max evaluated child is worth negative infinity.

We can see in the debug log below showing evaluation from the root that the tree policy in not visiting nodes it considers to be resolved based on the above algorithm, and in fact as soon as it determines child 124 to be losing for the opponent the root is updated as winning and is never visited again.



```
103 child 4 3 2 0: 1.000000 wins 2 visited (0.500000) <0 nodes lost>  [resolved]  (implied win)
104 child 4 5 0 1: 1.000000 wins 2 visited (0.500000) <0 nodes lost>  [resolved]  (implied win)
105 child 4 5 0 0: 37.500000 wins 59 visited (0.635593) <0 nodes lost>
106 child 4 5 1 1: 54.500000 wins 95 visited (0.573684) <0 nodes lost>
107 child 4 5 1 0: 75.500000 wins 145 visited (0.520690) <0 nodes lost>
108 child 4 5 2 0: 1.000000 wins 2 visited (0.500000) <0 nodes lost>  [resolved]  (implied win)
109 child 4 5 3 0: 88.000000 wins 175 visited (0.502857) <0 nodes lost>
110 child 5 3 0 1: 1.000000 wins 2 visited (0.500000) <0 nodes lost>  [resolved]  (implied win)
111 child 5 3 0 0: 56.000000 wins 99 visited (0.565657) <0 nodes lost>
112 child 5 3 1 1: 46.500000 wins 78 visited (0.596154) <0 nodes lost>
113 child 5 3 1 0: 24.000000 wins 32 visited (0.750000) <0 nodes lost>
114 child 5 3 2 1: 103.500000 wins 213 visited (0.485915) <0 nodes lost>
115 child 5 3 2 0: 0.000000 wins 2 visited (0.000000) <0 nodes lost>  [resolved]  (implied win)
116 child 5 4 0 1: 1.000000 wins 2 visited (0.500000) <0 nodes lost>  [resolved]  (implied win)
117 child 5 4 0 0: 17.500000 wins 20 visited (0.875000) <0 nodes lost>
118 child 5 4 1 1: 50.000000 wins 86 visited (0.581395) <0 nodes lost>
119 child 5 4 1 0: 44.000000 wins 72 visited (0.611111) <0 nodes lost>
120 child 5 4 2 0: 2.000000 wins 2 visited (1.000000) <0 nodes lost>  [resolved]  (implied win)
121 child 5 5 0 1: 1.000000 wins 2 visited (0.500000) <0 nodes lost>  [resolved]  (implied win)
122 child 5 5 0 0: 76.000000 wins 146 visited (0.520548) <0 nodes lost>
123 child 5 5 1 1: 94.000000 wins 189 visited (0.497354) <0 nodes lost>
124 child 5 5 1 0: 113.000000 wins 240 visited (0.470833) <0 nodes lost>  [resolved]  (implied loss)
125 child 5 5 2 0: 2.000000 wins 2 visited (1.000000) <0 nodes lost>  [resolved]  (implied win)
```

Here is another example, the algorithm determines this position to be completely winning for white although most amateur players may consider the position good it is difficult for humans to find the forced win.



```
playing as 0
turn: 7
player 0: 000010000 100000100 001000001 000010100
player 1: 100000000 001011001 000010100 000000000
1 . . 0 . 1
. 0 . . 1 1
. . . 0 . 1
. . 0 . . .
. 1 . . 0 .
1 . 0 0 . .
```

Other Considered Approaches

Experiments were performed with a version of the program that utilized Rapid Action Value Estimation as described in (Gelly & Silver, 2011). However it creates very little advantage since the branching factor is just two large for sub trees of any real depth to be generated. These experiments led to weird behavior and no version of RAVE could be implemented that reliably outperformed the original approach. Because similar Pentago moves can be good or bad for very different reasons in different board positions it is difficult to align the intentions of descendent nodes with their parents in order to update parent scores according to a RAVE tree policy.

Improvements

There are many ways to find efficiencies in AI agents, for this program a precomputed batch of opening moves would help with consistency there are some deep lines of play from particular openings that if an unprepared opponent walked into could produce some easier wins especially when playing as white. Domain specific knowledge in the form of heuristics might aid the agent in finding good branches quicker based on theory rather than pure Monte Carlo simulations. AlphaZero is a notable example of a UCT algorithm which relies on domain specific knowledge in order to better evaluate positions.

Levente Kocsis and Csaba Szepesvari, (2006). Bandit based Monte-Carlo Planning. Conference: Machine Learning: ECML 2006, 17th European Conference on Machine Learning. DOI 10.1007/11871842_29

Sylvain Gelly and David Silver,(2011).Monte-Carlo tree search and rapid action value estimation in computer Go. Artificial Intelligence Volume 175, Issue 11, July 2011, 1856-1875. https://doi.org/10.1016/j.artint.2011.03.007