UTC in Noughts and Crosses: Reinforcement Learning

Robert Marc James-Stroud
School of Computer Science and Electronic Engineering
University of Essex
Colchester, UK
Email: rj18801@essex.ac.uk

Abstract

Machine learning is a hot topic at the moment, with research being conducted in both GGP and GVGP as well as other areas. Having a machine learn the most optimal action removes the need for hand-crafted rules, allowing a more general approach to AI. This paper will be reviewing how well UTC performs in the fully-observable, Markov Decision Process game Noughts and Crosses.

I. Introduction

Monte Carlo Tree Search (MCTS) has received much interest from researchers in multiple areas, including General Game Playing (GGP) [1], where it has been used to great success.

MCTS is an algorithm for returning a decision by creating a search tree of the domain, constructed by taking random samples of the search space. MCTS does not require domain knowledge to function although it may be helpful [1].

Research into Game AI is important, prior to MCTS databases were used to hold all possible game states. An example of database is presented by [2], a game of *American Checkers* in which there are 3.9x10^{13} entries, or states. Decision trees used in GGP and General Video Game Playing (GVGP) suffer from dimensionality.

A simple game, Noughts and Crosses has a branching factor of 4, and a full tree has 10 levels [3], compared with Chess which has a branching factor of 35 and rarely a full search tree [3]. Noughts and Crosses has unique states numbering in the thousands, it is clear that crafting rules for each different state is not feasible, especially when the branching factor in other games used in GGP/GVGP can be in the hundreds.

Firstly this paper will look at what MCTS is and previous work conducted in the field of Game AI, specifically MCTS. Secondly MCTS will be used in a Noughts and Crosses (also known as Os and Xs, OXO and Tic-Tac-Toe).

A. Monte Carlo Tree Search

MCTS is a tree search algorithm that constructs an asymmetric search tree based on actions, the tree is a biased representation towards more promising areas of the search space [4].

Through self-play MCTS estimates the value of a node from that point until it reaches a terminal node [4]. A node is a state, and the action is the move made that results in that state.

Each node might be visited multiple times having its value adjusted accordingly. A node that looks promising on the first run through could look less promising as it is visited more times, or become less promising in relation to other branches.

1) *Selection* - Using a policy traverse the search tree. Starting at a root node, the current state, select child nodes with promising values until a leaf node is reached.
2) *Expansion* - Once *Selection* has selected a leaf node, expand the current node and select one of its children so long as the current node is not terminal.
3) *Simulation* - Using the node expanded in the *Expansion* phase and a policy if defined, playout the game. *Playout* in this context means until completion or a result is achieved.
4) *Backpropogation* - After *Simulation* is complete the search tree is traversed in reverse order, propagating up the tree the result, updating the value of the nodes, this continues until the current node is the root node. The new values are then used in the *Selection* process.

MCTS will continue to cycle through these phases until it is stopped, or an 'execution budget is reached' [5]. Three ways are presented to achieve this, each with their own detraction:

- Time - cull MCTS after a certain amount of time has lapsed - if stopped too quickly it may not evaluate nodes accurately.
- Depth Culling- Stop MCTS after hit reaches a certain depth limit in the tree - stopping with depth may stop the search from ever reaching a terminal node.
- Iteration Limit - Give it a predefined number of cycles, after which it will return a result. One cycle is all four phases of MCTS.

Depending on when or how MCTS is stopped will effect how good the estimation is of the action to take. Using depth culling or an iteration limit is useful when comparing algorithms across inconsistent hardware and programming languages.

II. Previous Work

MCTS has received much interest from researchers, becoming somewhat of an umbrella term covering any implementation of MCTS. Browne *et al.* in [6] summarised the different MCTS implementations and enhancements up to 2011.

MCTS has been applied in co-operative scenarios [4], Realtime games [6], Non-deterministic games [7] and numerous non-game applications [6].

Game AI research used to focus on two-player zero-sum games of perfect information with alternating turns [6], every two-player zero-sum game has a solution [3]. Two main ways of evaluating states in a minimax scenario is to use heuristics, or statistics. The statistics approach can utilise MCTS to run thousands of playouts to find the optimal strategy [3].

A. *Upper Confidence Bounds*

UCB1 is a bandit algorithm with a logarithmic regret, proposed by Auer *et al.* [8]. In a multi-armed bandit problem the 'empirically best action' should be taken as often as possible [8].

However less explored options should not be ignored in favour of exploitation. Disregarding exploration may leave better actions unexplored and thus increase regret.

UCB1 policy dictates that arm j maximises the average reward [8]:

$$UCB1 = \overline{X_j} + \sqrt{\frac{2\ln n}{n_j}}$$

where $\overline{X_j}$ is the average reward from arm j, n_j is the number of times arm j was played and n is the total number of plays.

Reward $\overline{X_j}$ encourages exploitation of high-reward choices [8] and $\sqrt{\frac{2\ln n}{2_j}}$ encourages exploration of less visited actions [6].

B. *Upper Confidence Bounds for Trees*

The result of combining UCB1 and MCTS is UCT. Upper Confidence Bounds for Trees (UCT) is a well known and popular algorithm in the MCTS family [6]. Kocis and Szepesvari [9] proposed applying UCB1 to MCTS, using UCB1 as the tree policy [6].

UCB1 is efficient and simple [6] and a 'promising candidate to address the exploration-exploitation dilemma in MCTS' [6].

Treating the selection of a child node as a multi-armed bandit problem, using Monte Carlo simulations the the 'expected reward can be approximated'. Just as a node can be viewed as a multi-armed bandit problem so to can an action to be selected. Treated as an independent multi-armed bandit problem the child node j is selected to maximise [6]:

$$UCT = \overline{X_j} + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where C_p is a constant. The value of C_p can be adjusted to increase or decrease the amount of exploration MCTS does, however a default value of $C_p = \frac{1}{\sqrt{2}}$ is commonly used [6].

UCT balances exploration and exploitation with the exploration term, $2C_p\sqrt{\frac{2\ln n}{n_j}}$, where when the child node is selected the numerator increases making it less attractive to select again. However when a sibling is landed on the numerator increases, increasing the probability that the child node will be selected. The exploration term ensures that a child node will be explored by giving it a non-zero probability of selection [6].

Each node in UCT has four pieces of information associated with it, the state, incoming action, visit count and reward [6]. The state is a result of the incoming action, a state-action pair. The number of visits, $N_{(v)}$, is a counter of how many times that node has been selected. Reward, $Q_{(v)}$, is the value of all playouts that passed though this node, therefore $\frac{Q_{(v)}}{N_{(v)}}$ is an approximation of the node's value.

When a playout is complete *Backpropogation* will update $N_{(v)}$ and $Q_{(v)}$ for each node with the new values.

III. Methodology

Noughts and Crosses is a two-player zero sum game, as mentioned previously it will always have a solution. To determine the best move an agent can take UCT will be used.

UCT will function well in this environment because Noughts and Crosses has full observability, and presents the Markov Property [10]. A Markov state is one in which all useful information about the history is present in the current state, 'the future is independent of the past given the present' [10].

Constructing a search tree with UCT the current state will be the root node, from which the MCTS cycle can begin by expanding the node and simulating a game until a terminal point. After which reversing the order of descent, values attributed to each node will be updated as explained in 2.b. Data for the game Noughts and Crosses will be gathered from a UCT implementation built by Cowling *et al.* [11]. During the game all critical data will be logged to a csv file for later use in understanding if the agent is improving.

Critical data in Noughts and Crosses is the game state at the start of an agent's turn, the generated search tree, selected action and resultant state and which agent won, if either.

Analysing the data gathered from running many games will give a good representation of game states and how the agents handle it, whether they result in success, failure or a draw. To gather sufficient data the running of the framework will need to be automated, as running each game manually is not ideal due to the large set of data required to understand if learning is happening.

IV. Experiments

Although Noughts and Crosses is not a complex game and situations with low reward are obvious to human players, an AI does not have an intuition allowing it to eliminate low reward paths without further investigation. This is an issue as the branching factor increases, the difficulty of the task increases exponentially [12].

When collection of the data is complete and analysed it would be expected that a very high percentage of the games would end in a draw. Games routinely ending in a draw means that the agents have understood the state and how to minimax, as Noughts and Crosses will end in a draw if played perfectly [13]. It is only expected that an agent will win if the other agent makes a mistake.

V. Results

The first set of results from a thousand games show that player one won 3.3% of the time and player two won 9.1% of the time. This leaves an 87.6% draw rate.

	Wins	Losses	Draws
Player One	33	91	876
Player Two	91	33	876

The second set of results show a similar phenomena, with player one winning again 3.3% of the games and player two 8.4% of the games.

	Wins	Losses	Draws
Player One	33	84	883
Player Two	84	33	883

A third set of results show a similar story. 3.3% win rate on player one and 8.1% for player two.

	Wins	Losses	Draws
Player One	33	81	886
Player Two	81	33	886

VI. Conclusion

This paper has shown what MCTS, UCB1 and UTC is and how it can be applied to GGP through its use in Noughts and Crosses.

As predicted in section 4 and shown in section 5, the draw rate of the two agents is very high, combined they won a little over a tenth of the total games played.

This is because the more time a node is selected for playout the more confident the algorithm is about the theoretical value that node has. Usually the most visited node will be the node with the highest value, and this is the action the agent is likely to select when asked what the best node is.

VII. plan

References

[1] C. F. Sironi, J. Liu, D. Perez-Liebana, R. D. Gaina, I. Bravi, S. M. Lucas, M. H, M. Winands, “Self-Adaptive MCTS for General Video Game Playing”.

[2] A. Benbassat, M. Sipper “EvoMCTS: A Scalable Approach for General Game Learning,” *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, 2014.

[3] R. A. Bartle “Game Theory,” *CE810 Game Design Lecture 5*, 6 November 2018.

[4] P. R. Williams, J. Walton-Rivers, D. Perez-Liebana, S. M. Lucas, “Monte Carlo Tree Search Applied to Co-operative Problems,” *2015 7th Computer Science and Electronic Engineering Conference (CEEC).*

[5] R. D. Gaina, S. M. Lucas, D. Perez-Liebana, “Rolling Horizon Evolution Enhancements in General Video Game Playing, ” *IEEE Conference on Computational Intelligence and Games*, 2017.

[6] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Taverner, D. Perez, S. Samothrakis, S. Colton, “A Survey of Montro Carlo Tree Search Methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, 2012.

[7] P. I. Cowling, E. J. Powley, D. Whitehouse, “Information Set Monte Carlo Tree Search” *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, 2012.

[8] P. Auer, N. Cesa-Bianchi, P. Fischer, “Finite-Time Analysis of the Multiarmed Bandit Problem,” *Machine Learning*, Vol. 47, 2002.

[9] L. Kocsis, C. Szepesvari, “Bandit Based Monte-Carlo Planning”

[10] J. Walton-Rivers, “Reinforcement Learning,” *CE811 Game Artificial Intelligence Lecture*, 12 November 2018.

[11] P. Cowling, E. Powley, D. Whitehouse, “Python Code”, [online] available at: http://mcts.ai/code/python.html, 2012, [Accessed February 2019].

[12] S. Mohandas, M. A. Nizar, “A.I for Games with High Branching Factor,” *International CET Conference on Control, Communication and Computing (IC4)*, 2018.

[13] P. McOwan, P. Curzon, “Winning Games: the perfect tic-tac-toe player,” p3, Queen Mary University London available: http://www.cs4fn.org/teachers/activities/winatoxo/winatoxo.pdf.