CME 323, Report

Yifan Jin, Shaun Benjamin

1 Introduction

Monte Carlo Searching Tree(MCST) is a method 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. The idea of this method is given by Bruce Abramson in his 1987 PhD thesis. He experimented in-depth with Tic-tac-toe and then with machine generated evaluation functions for Othello and Chess. In 1992, B. Brügmann employed this method for the first time in a go-playing program. In 2006, R. Coulom described the application of the Monte Carlo method to game-tree search and coined the name Monte Carlo tree search, L. Kocsis and Cs. Szepesvári developed the UCT algorithm and S. Gelly et al. implemented UCT in their program MoGo. In 2008, MoGo achieved dan (master) level in 99 go and the Fuego program began to win with strong amateur players in 99 go. In January 2012, the Zen program won 3:1 a 1919 go match with John Tromp, a 2 dan player. MCTS has many attractions: it is a statistical anytime algorithm for which more computing power generally leads to better performance. It can be used with little or no domain knowledge, and has succeeded on difficult problems where other techniques have failed.

2 Original Monte Carlo Tree Search

The basic MCTS is conceptually very simple. A tree is build in an incremental and asymmetric manner. Each iteration of MCTS consists of four steps:

- Selection: Begin with some root R, a *tree policy* is used to find the most urgent child of R, then we successively select child till we reach a leaf L.
- Expansion: Unless the node L ends game, create one or more node of L and pick one of them, call it C.
- Simulation: play random playouts from C.
- Backpropagation: Update the information of the nodes in the path from C to R using the result of the random playouts.

The *tree policy* is the rule which is used to select a node amongst the children of the root. The main difficulty in selecting child nodes is maintaining some balance between the exploitation of deep variants after moves with high average win rate and the exploration of moves with few simulations. The idea is we should maximize the cumulative reward by consistently taking the optimal action. The simplest and most widely used policy is called UCT which is introduced by L. Kocsis and Cs. Szepesvári. They recommend to choose in each node of the game tree the move, for which the expression

$$\frac{w_i}{n_i} + c\sqrt{\frac{\log t}{n_i}}$$

has the highest value, where w_i is the number of of wins after the ith move, n_i is the number of simulations after the ith move, is the exploration parameter which theoretically equal to $\sqrt{2}$. t is the total number of simulations, equal to the sum of all n_i. These steps are summarized in the following pseudocode in Algorithm 1:

Algorithm 1 Original MCTS

function MCTSEARCH(s_0)
    create root node v_0 with state s_0.

while within computational budget **do**
$v_l \leftarrow$ *TreePolicy*(v_0)
$\Delta \leftarrow$*Simulate*$(s(v_l))$
Backprog$(\Delta,\ v_l)$
return *Bestchild*(v_0)

where textitBestchild leads to the best child of the root node v_0.

3 Data Representation

The data is represented as a graph and the graph is stored locally on the driver. This is done to speed up the node selection process since if the graph is distributed it would take much longer to search for the appropriate node to expand. In our setting the nodes of this graph correspond to different states of the game. For example, in the Nim game the state is encompassed by two numbers: the number of chips left and the current player. The edges in the graph are then the moves required to go from one game state to another game state. Finally, terminal nodes are states where the game has ended and no more moves are available to play. In our algorithm, there are three possible outcomes at the end of a game: a win, a loss and a draw.

4 Distributing the algorithm

4.1 Version 1

There are four main stages in the algorithm, selection, expansion, simulation and backpropagation. The most time consuming stages are selection, simulation and backpropagation. The selection and backpropagation process is done locally on the driver because they require traversing the graph and distributing a graph that is growing in an unexpected fashion prevents us from getting any speedups. This leaves the simulation stage which is the most straight forward to parallelize.
In order to parallelize the algorithm, we modify the expansion and simulation stages. After we have selected a node to expand, we expand the node into m random children rather than a single child. Furthermore, rather than simulating out the child state only once, we simulate each child state k times. Here k is a parameter than can be determined using m and N the number of nodes in the cluster, we choose k to be at least N/m. This is to ensure every node in the cluster is occupied during the simulation stage.
To simulate the m random children k times, we serialize each of m states k times and parallelize the states over the cluster. Then we map the states with a custom function that implements the random simulation. This function de-serializes the states and selects a random move for every turn until no more moves can be played. Once the simulation ends, the outcome is returned to the driver by a reduce call that appends all of the results of the playouts as a list. Finally, we iterate over the list and back pteropagate the results up the graph.
As it turns out, this modification only leads to a minor improvement in game playing strength even for large values of m and k. The first reason is that we only expand a single nodes children in each iteration. This means that during a single iteration of the search, we are only exploring what we currently think is the best move while ignoring other good moves. The second reason is that we do not expand a node past its children. This means that at the end of each iteration we only expand down a single level at a time.

4.2 Version 2

First we add the ability to explore more than just the current best move by modifying the tree policy. Recall that during the selection stage, we traverse the graph from the root by choosing the child with the largest UCT value. We now modify the policy to choose a child node randomly, but in proportion to its UCT value. This means the selection process is no longer deterministic which allows us to run this process several times and select multiple nodes for expansion. We know denote m to be the number of nodes we select for expansion, and we let k be the same

as before, the number of times we simulate each node. Next we allow expansion past a single layer of children. Previously, we could not expand past a single layer of children because we cannot apply the tree policy on the children nodes. In order to apply the policy we need the UCT values of the children nodes, but since they have yet to be simulated, their UCT values do not exist. Thus, we set nodes that have yet to be visited to a small UCT value (~ 0.1) so that we can allow deeper expansion of the search tree in a single iteration.

5 Result

- 4x4 Board: Distributed MCTS set to m = 100, k = 10 and iterations = 10 vs. regular MCTS with iterations = 100 $\rightarrow$ 24 games, 12 wins, 6 losses, 6 draws
- 6x6 Board: Distributed MCTS set to m = 100, k = 20 and iterations = 100 vs. regular MCTS with iterations = 1000 $\rightarrow$ 10 games, 3 wins, 7 losses
- 6x6 Board: Distributed MCTS set to m = 200, k = 1 and iterations = 100 vs. regular MCTS with iterations = 1000 $\rightarrow$ 10 games, 3 wins, 7 losses
- 6x6 Board: Distributed MCTS set to m = 100, k = 1 and iterations = 10 with custom UCT values vs. regular MCTS with iterations = 100 $\rightarrow$ 10 games, 5 wins, 5 losses

6 Analysis

6.1 Gameplaying Strength

First we see that the strength of the distributed monte carlo tree search does not linearly scale with the number of iterations. This is because the distributed algorithm performed well on the 4x4 board but with the same values of m and k, it performed poorly on the 8x8 board. This means that increasing the number of iterations for the distributed MCTS does not increase its gameplaying strength as much as it does for the regular MCTS. From the results we see that increasing k does not improve the strength of the MCTS since on the 6x6 board, a larger k performs poorly. This is most likely because for a large board, there are many possible end states and so the results from a random simulation is not very accurate in assessing the value of a node. Next we see that increasing m improves the strength of the algorithm. This is intuitive as increasing m allows us to expand a larger proportion of the search tree.

In order to match the regular MCTS with 1000 iterations we had to increase m to 200 and the number of iteration to 100. Using our analysis we realized that k is only important when the number of states remaining is small, and so we set $k = \min(10, m/s)$ where s is the number of states to be simulated. S can be smaller than m if there are fewer than m positions available to play on. This means that in the early stages of the game $s == m$ and $k == 1$, and so we will spend most of the time exploring the tree. In the later stages of the game when $s < m$, and $k > 1$ and so we will spend more time assessing the quality of each node. In conclusion, we find that to increase that the strength of the distributed MCTS scales by m and the number of iterations and that the scaling in gameplaying strength of the distributed MCTS is not as straightforward as the scaling of the gameplaying strength of the regular MCTS.

6.2 Time Complexity

The runtime of the algorithm can be simply be computed as $O(mkI/C)$ where m and k are the same as before, and I is the number of iterations and C is the number of cores available.

6.3 Memory Complexity

The memory complexity is $O(mk)$ since in each iteration we map mk states over the cluster.

6.4 Communication Analysis

The communication involved is one-to-all and all-to-one. In the map phase we send out data of size mk from the driver to all the worker nodes, and in the reduce phase we send data on the order of mk back to the driver.

7 Conclusion

In conclusion we propose a scheme for distributing the Monte Carlo tree search that does not scale as well as the regular MCTS. However, given enough cores, it is possible to configure the parameters in order to improve over the regular MCTS.

- $m(< N)$ is a parameter that is left to be tuned by the programmer, its how complex the game is, larger m should be
- Increasing mk doesnt increase overall runtime by much, so choosing a higher mk gives a better $computation/time = performance$.
- Runtime strongly dependent on the number of mapreduces, approximate 90% of time is spent in the mapreduce.
- Main advantage of distributed MCTS is that its scale invariant to game complexity, i.e. to mk, where m is possible number of moves, and k is the variance of the outcome of a random game