Jose A. Ramos
jar2333
Parallel Functional Programming

Proposal: MCTS.hs

Introduction

For the final project, I wish to implement the Monte Carlo Tree Search[1] general game playing algorithm in Haskell, with support for parallelizing it using Haskell's parallelism constructs/API. I have actually partially implemented this[2] (though not yet working or tested). Nevertheless, I will discuss both features that are already implemented and those that are planned.

Algorithm

Monte Carlo Tree Search (MCTS) is an algorithm for playing any two player game. It searches the game tree, where each node is a possible state of the game. The children of a node are those game states which can be reached by a move from the current player. A node with no children (a tree leaf) is a terminal state, where the outcome of the game is decided (who won and who lost). Unlike other tree-based algorithms which prune the search space like Minimax[3] variants, MCTS does not require any heuristics or custom evaluation function for a given game. Instead, it "samples" the outcome of a game from a given game state, by simulating an entire game starting at said state—usually by randomly sampling moves until completion. This implies that the algorithm can be applied to *any* two-player game, given that the rules of the game are known. More precisely, one has to define a game state struct/object, which has any relevant game state information, along with a few functions:

- One which takes a game state and returns all possible children.
- One which takes a terminal game state and returns the outcome of the game (or returns something indicating that the state is not terminal).
- One which takes a game state, simulates an entire game to completion, and returns the outcome of the game.

Implementation: Typeclass

The above description can be formalized in Haskell using typeclasses. The MCTS module can define a GameState typeclass (given a Player data type):

```
data Player = One | Two | Tie deriving (Eq)
class GameState g where
    next :: g -> [g]            -- gets node children
    eval :: g -> Maybe Player   -- determines leaf node winner
```

[1] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
[2] https://github.com/jar2333/MCTS.hs
[3] https://en.wikipedia.org/wiki/Minimax

```
    sim  :: g -> Player          -- determines winner (simulation)
```

Any instance of this type class can be used in the MCTS algorithm, making the implementation as generic as possible. For the purposes of the final project, concrete implementations of game playing agents using the generic MCTS algorithm can also be provided, such as for Tic-Tac-Toe, Dominoes, etc…

Implementation: Monads

In the implementation of the algorithm as presented in the MCTS wikipedia page (or more authoritatively, this Swarthmore page[4]), the algorithm is characterized as a sequence of N transformations applied to the game tree. Each transformation can be described imperatively as 4 steps:

1. Selection: walk down the game tree and select a leaf node using UCB formula.
2. Expansion: expand the leaf node and add n of its children to the tree.
3. Simulation: simulate games starting from each of the n children.
4. Backpropagation: propagate the simulation results up the tree.

After N iterations of the above process, the root node's child with the highest score is chosen as the next state to be reached. In my current implementation, the selection and backpropagation occur in the same function *walk*, which recursively walks down the tree to a leaf node, uses the State monad to store the simulation results obtained at said leaf as a state, then reconstructs or "updates" the current node's information using the state. The *iterate N* function is simply an invocation of *evalState* on the results of *walk*, N times. The full generic *mcts* function will utilize this *iterate* function as its primitive. This is the most prominent use of a monad in the implementation, though the Random[5] and Par[6] monads will be used in simulation implementations and MCTS parallelization, respectively.

Implementation: Parallelism

According to the wikipedia article, there are 3 main ways to parallelize MCTS:

- Leaf parallelization, i.e. parallel execution of many playouts from one leaf of the game tree.
- Root parallelization, i.e. building independent game trees in parallel and making the move based on the root-level branches of all these trees.
- Tree parallelization, i.e. parallel building of the same game tree…

Of these, both leaf parallelization and root parallelization are straightforward to implement. In the case of leaf parallelization, one can run a parallelized*map sim $ next g* for a given game state *g*. For root parallelization, one can call *iterate N* in parallel on M initial game states, and then combine the resulting trees with a *foldr*, before doing the final root child selection. Given that randomness is involved in the simulation, this will yield more accurate results.

[4] https://www.cs.swarthmore.edu/~mitchell/classes/cs63/f20/reading/mcts.html
[5] https://hackage.haskell.org/package/MonadRandom-0.1.3/docs/Control-Monad-Random.html
[6] https://hackage.haskell.org/package/monad-par-0.3.5/docs/Control-Monad-Par.html