William Kenyon

A Monte Carlo Tree Search Library in Haskell

Computer Science Tripos

Emmanuel College

May 18, 2012

Proforma

Name:	**William Kenyon**
College:	**Emmanuel College**
Project Title:	**A Monte Carlo Tree Search Library in Haskell**
Examination:	**Computer Science Tripos 2012**
Word Count:	**11,000**
Project Originator:	**Dr Sean Holden**
Supervisor:	**Dr Sean Holden**

Original Aims of the Project

There were two aims for this project. The first was to write a library allowing programmers to use *Monte Carlo Tree Search* (*MCTS*) in the context of game playing in *artificial intelligence* (*AI*). The second was to determine if the success achieved by MCTS at playing *Go* could be translated into the *Connect* family of games.

Work Completed

Over the course of this project, a versatile MCTS library has been constructed. The library can be used to implement game playing *agents* for a large range of games, from single player puzzles to n player, imperfect information, non zero-sum games with simultaneous play. The library is fully documented and supports the implementation of many of the modifications to MCTS proposed in current research. This library is then used to make an agent which plays all games in the Connect family. This agent was able to beat a *minimax* based agent in a 100 game *Freestyle GoMoku* tournament.

Special Difficulties

None.

Declaration

I, William Kenyon of Emmanuel College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Contents

List of Figures

Acknowledgements

Thanks to Dr Sean Holden for supervising, Professor Neil Dodgson for proof reading, and Jestine Ang for offering suggestions to improve sentence structure.

Chapter 1

Introduction

This chapter introduces the two major topics that this project is based on. Section 1.1 explains the *Monte Carlo Tree Search* (*MCTS*) method and discusses the improvements it makes upon more traditional game playing techniques. Section 1.2 introduces the *Connect* family of games and examines why it would be an interesting endeavour to write a *game playing agent* for this family.

1.1 Monte Carlo Tree Search

MCTS is a state-of-the-art method for making optimum decisions in artificial intelligence. MCTS is the basis for the best computer players for *Go*, a highly-strategic *combinatorial game* with a large search space [7, 8]. The significance of MCTS is highlighted by the fact that Go is considered to have replaced chess as the '*drosophila of AI*' [8].

MCTS is an evolution of *Monte Carlo simulation* and *minimax* search. Monte Carlo methods have their roots in statistical physics where they have been used to obtain approximations to intractable integrals and have since been used in a wide array of domains including games research [3]. Monte Carlo simulation follows a simple non-deterministic policy to play out a large number of games from a given start position. Often this policy is to randomly pick from all of the legal moves with uniform probability. Each simulation continues until the game reaches a terminal position. The average utility of all the terminal states reached can be used as an estimate for the utility of the start position. Monte Carlo simulation is usually used for games with an element of chance [18, 19].

Minimax decides a move which minimizes loss in the worst case scenario. In order to do this perfectly, a full depth game tree must be built. Building an unlimited depth game tree is, in general, computationally unfeasible. An evaluation function is often used to estimate the utility of the leaves of a depth limited game tree. Minimax was the basis of the tree search algorithm in *Deep Blue*, a chess-playing computer which defeated world champion Garry Kasparov in 1997.

Both Monte Carlo simulation and minimax based agents have flaws which MCTS improves upon. Agents using Monte Carlo simulation model opponents as players following a random strategy. This is not representative of rational opponent strategy. Therefore, many simulations are required in order to converge upon a good estimate of utility. Minimax agents, on the other hand, assume that an opponent will always make the best move she can. However, the use of minimax is dependent upon the existence of a good evaluation function. Finding an evaluation function can prove difficult for games such as *Go* [8] and *Kriegspiel* [6]. In addition, minimax agents are not able to consider as many moves ahead as Monte Carlo simulation agents, and are thus considered less strategic.

An attempt at solving the deficiencies of both methods was to combine a depth-limited minimax search with averaged Monte Carlo simulations as a utility function. MCTS improves on this approach by not separating between a minimax phase and a Monte Carlo phase. The search progressively changes from averaging to minimax as the number of simulations grows. This provides a fine-grained control of tree growth at the level of individual simulations [7].

Figure 1.1 illustrates the four stages of MCTS: *Selection*; *Expansion*; *Simulation* and *Backpropagation*. It also highlights the iterative nature of the search. In current literature, the specification for the Expansion, Simulation and Backpropagation phases of the search vary considerably across different applications. The library produced in this project allows full configuration of each of these phases while providing sensible defaults.

Upper Confidence Bounds

The Selection phase of MCTS is almost always based on *Upper Confidence Bounds* (*UCB*) [3]. The UCB strategy chooses the child node, i, that maximises the UCB score, Q_i:

$$Q_i = v_i + c \times \sqrt{\frac{\ln N}{n_i}}$$

where v_i is the minimax score of node i, n_i is the number of times this node has been explored on previous iterations, N is the number of times the current node has been explored and c is an 'exploration' constant. The UCB strategy represents a compromise between favouring unexplored nodes and nodes that have shown promise so far. It is this guided, but exploratory nature of UCB which makes MCTS such a powerful method.

Multi Player Monte Carlo Tree Search

Minimax can only be used for zero-sum, two-player games. A single value can represent the utility of a position for two players. Minimax can be generalized to the *Max-n* strategy for non zero-sum games or games with multiple players. For this strategy, the utility, or estimated utility, of a position for player i is stored

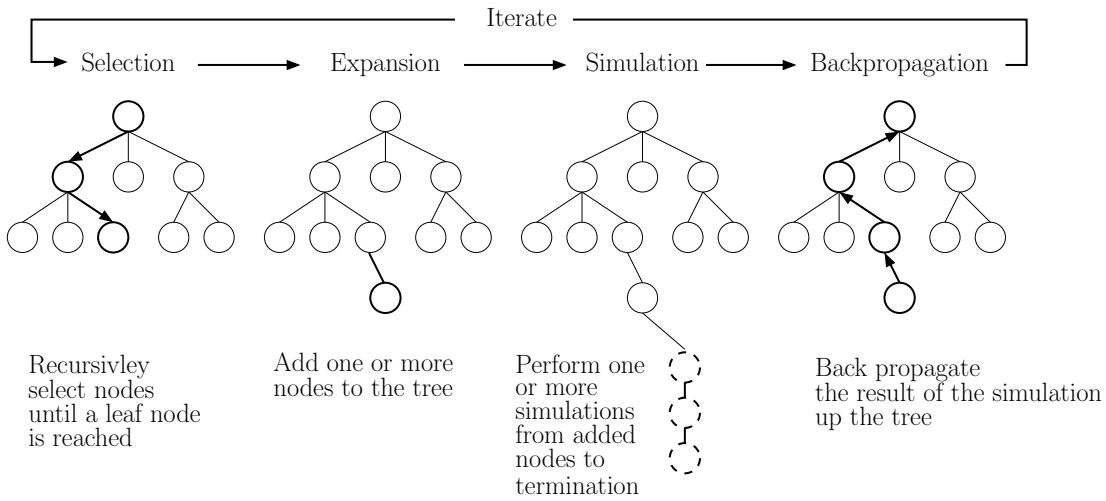


Figure 1.1: Demonstration of the phases of MCTS. (Figure adapted from [5])

in the ith element of a *score tuple*. By using a similar score tuple in the game tree, MCTS can be genralized to *Multi Player MCTS* [4].

1.2 Connect

Connect is a class of two player zero-sum games. It includes *Tic-Tac-Toe*, *Freestyle GoMoku* and *Connect6* [22]. The rules of Connect (n,m,k,p,q) are simple. The board is an $n \times m$ Go grid. Two players, Black and White, take turns to place stones of their color on the grid. The first player to get k consecutive stones on any diagonal, horizontal or vertical line is the winner. On her first turn, Black places q stones. On all subsequent turns for either player, p stones are placed. The following equivalences hold:

TicTacToe = Connect (3,3,3,1,1)

Freestyle GoMoku = Connect (15,15,5,1,1)[1]

Connect6 = Connect (19,19,6,2,1)

Connect (n,m,k,1,1) games are considered unfair. This is because black always has one more stone on the board than white at the end of her turn. White, on the other hand, has the same number of stones as black at the end of his turn. Intuitively, one additional piece on the board for a given player cannot be a disadvantage to that player. In fact, detailed analysis shows that there exists a winning strategy for black in Connect (15,15,5,1,1) [1, 20].

Connect (n,m,k,2,1) games were invented with the intention of restoring fairness. They seem intuitively fair because at the end of a player's turn, that player would have one more stone on the board than their opponent. However, there is no conclusive evidence that this indeed constitutes a fair game.

[1]Freestyle GoMoku is sometimes played on boards of other sizes

Herik [20] defines the *state-space complexity* as the number of legal game positions reachable from the initial position of the game. The state-space complexity and *branching factor* of Connect (19,19,k,1,1) is similar to that of Go. However Connect (19,19,k,2,1) has branching factor of $\binom{19\times19-1}{2} = 64620$ at the start of the game, about 180 times larger than the branching factor in Go.

Part of the reason MCTS performs so well in Go is that playing moves which seem strategically neutral at the time of play can be of critical importance 50 or 100 moves later [8]. The exploratory nature of UCB combined with with deep simulations allows MCTS to consider such moves. The strategy adopted by most computer and human players of Connect is to play tactically and maximise the short term advantage [12, 22]. It is hoped that a MCTS agent that considers a larger range of moves might uncover a hidden strategic element to Connect.

Chapter 2

Preparation

This chapter details the preparatory work done to enable the development of the *MCTS* library. Section 2.1 considers what requirements a user[1] might have the library. Section 2.2 explains why *Haskell* is a suitable choice of language for the library. The design of the library, and a discussion of how the design meets the requirements can be found in Section 2.4.

2.1 Requirements Analysis

2.1.1 Top Level Search Requirements

The library must allow the user to...

1. Perform a certain number of iterations of MCTS from a certain game state.
2. Perform as many iterations possible from a certain game state in a given time.

During a cursory literature review, it became apparent that MCTS is usually utilised via the second method [8, 6, 22]. This makes sense as most competitive games are played with a time limit on moves. The inclusion of the first method could be useful for playing games with no time limit, for example, some single player puzzle games. It could also provide a useful tool for evaluating agent performance.

In both items 1 and 2, it is implicit that MCTS is started from a singleton game tree, that is a tree with only a root node and no children. It is possible that a user may wish to perform MCTS on a partially populated tree. Figure 2.1 illustrates why this might be desirable. When performing MCTS from the start position of node 1, a large tree is built up. The agent then makes the most desirable move, in this case node 2. Since this is a single player game, the agent will then immediately perform MCTS with node 2 as the root node. It

[1] In this project, the 'user' refers to a programmer or developer using the MCTS library to write an agent for some arbitrary game.

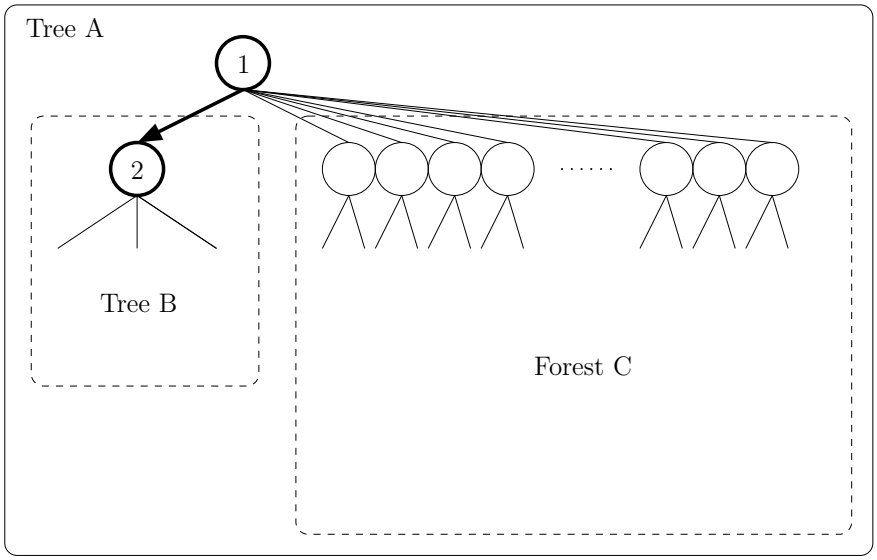


Figure 2.1: Tree produced after many iterations of MCTS for a single player game.

might seem advantageous for the agent to start MCTS on tree B rather than the singleton node 2. This would be able to take advantage of the work done by the search on the previous move and build a bigger tree than would have otherwise been possible in the same time. In fact, performing MCTS on a partial tree is not as useful as it may seem:

> MCTS will usually spend far more work building forest C than tree B (Figure 2.1). There are, however, two exceptions. When a greedy, non-exploratory Selection algorithm is used, or when the game being considered has a very low branching factor. These exceptions raise the question of why MCTS is being used for this problem.
>
> The example considered is a one player game. The advantages diminish further for games with more players. For example, if Figure 2.1 represented a 2 player game then node 2 would represent a move for the other player. Therefore, it would be a subtree of tree B that would be used to start the next execution of MCTS.

In addition, allowing users to perform MCTS on a partial tree would expose the user to a disproportionate increase in complexity, as the tree data structure would need to be exposed across the library interface. Hence, only items 1 and 2 are pursued in this project.

2.1.2 Search Implementation

This section examines how a user may wish to customize the various phases of MCTS. It also examines what might be considered suitable defaults for each of the phases.

The user must be able to specify how to carry out the...

1. Selection Phase

   Most basic Selection algorithm used is UCB. UCB was first used in the context of MCTS in Kocsis & Szepesvári [11]. This should be used as default because many of the enhancements upon UCB are not as general purpose and are suited to games of a particular type. The value of the exploration constant, c, should by default be set to 1.

2. Expansion Phase

   Typically, upon expanding a leaf node, either one child node, or all child nodes are added to the tree [3]. Which is chosen depends on the domain and computation budget. The MCTS library should allow the user to choose the maximum number of nodes to expand into the tree. By setting a value larger than the branching factor of the problem, all child nodes will be added to the tree. By default, a maximum of one node should be added to the tree.

3. Simulation Phase

   The default policy for MCTS is to select randomly amongst the available actions [3]. This simple strategy requires no domain knowledge, and repeated trials will most likely cover different areas of the search space. However, games simulated in this way are unlikely to be comparable to those played by rational players. The user should be able to define a bespoke Simulation strategy. They may do this to make Simulations more realistic, or for efficiency reasons.

4. Backpropagation Phase

   Users must be able to define how the score tuples in each node are altered when backpropagating a certain result. By default, the library should assume a 2 player, zero-sum game. Therefore, in the case of a win, the winning player should have her score increased by one in the score tuple, and the losing player have his score correspondingly decreased. Note that the true, game theory style score tuple is obtained by dividing each element by the number of times that the node has been simulated. Certain enhancements to Backpropagation have been found in the literature. For example, Xie and Liu [21] modify the Backpropagation algorithm to weight the results of Simulations performed later in the search higher.

2.1.3 Implementable Game Types

Traditional game AI research focuses on zero-sum games with two players, alternating turns, discrete action spaces, deterministic state transitions and perfect

information [3]. MCTS has been applied extensively to such games. Go, being the iconic example [8]. However, it has also been applied to other game types.

The user must be able to write agents for games with...

1. A single player

   SP-MCTS, a version of MCTS for single players, was introduced by Schadd [17, 16]. This was applied to *Bubble Shooter*.

2. Multiple players

   Cazenave investigated using MCTS to play *Multi-Player Go* [4]. Samothrakis et al. [15] used MCTS to play *Ms. Pacman* by modelling each ghost as an individual player.

3. Imperfect information

   In games with imperfect information, the game state is only partially observable to both players. Cianciani and Favini used MCTS to play Kriegspiel [6]. Kriegspiel has rules that are similar to chess, but the opponents pieces are obscured by a 'fog of war'.

4. Simultaneous play

   The *Ms. Pacman* agent, mentioned in item two, is also an example of a simultaneous play game.

2.2 Language Choice: Haskell

Haskell is a high level language and code can be written clearly and concisely. Until recently, no compilers or interpreters were able to run Haskell code as fast as low level languages such as C. It has been suggested that recent improvements in the *Glasgow Haskell Compiler* (*GHC*) may allow Haskell to be competitive at computational intensive problems such as *MCTS* [9].

2.3 Investigating Haskell Features & Libraries

Many features of *Haskell* used in this project have not been taught in the *Foundations of Functional Programming* course of this tripos. Potentially useful features & libraries were identified in *Real World Haskell* [13], *A Gentle Introduction to Haskell* [10] and the *GHC* documentation [14]. These features include *type classes*, *associated type families*, *monads*, *lazy evaluation*, *GHC* profiling, the *state transformer monad* (`Control.Monad.ST`), generating random behaviour (`System.Random`), *automated unit testing*, the *module* system, (`QuickCheck`) and documentation generation using *Haddock*.

2.3.1 Type Classes

Type classes implement *ad hoc polymorphism* or *overloading* [10]. They are similar to *interfaces* in the *Object Oriented Programming* (*OOP*) paradigm. Fragment 2.1 shows an example of a type class, `Game`, in the context of a library requiring users to define a type representing a game position. Block 1 expresses that for

Fragment 2.1 An example of a `Game` *type class* and an instance of that class, `TicTacToe`.

```
1 --Block 1 - In library code
2 class Game a
3   legalChildren :: a → [a]
4   currentState :: a → GameState
```

```
1 --Block 2 - In user code
2 instance Game TicTacToe
3   legalChildren = --insert implementation here
4   currentState = --insert implementation here
```

any type `a` to be an instance of the `Game` class, the functions `legalChildren` and `currentState` must be implemented. Block 2 gives the implementation of those functions for the game `TicTacToe`.

2.3.2 Associated Type Families

Associated Type Families is a bleeding-edge Haskell extension that is currently only specified in GHC documentation [14]. It is a generalisation of type classes to support ad hoc polymorphism or overloading of data types. It enables several different data types to have the same name, each one associated with a different instance of a type class. Fragment 2.2 expands upon the example given in Fragment 2.1 by specifying that a `Player` data type must be associated with any instance of the `Game` type class. It illustrates how type families could enable users of the MCTS library to define an arbitrary number of sensibly named players. `Player a` has a signature of `*`. This is a *kind signature*, not a type signature. It expresses that `Player a` is a concrete type. `Player` has kind `*->*`, which expresses that it takes one type as an argument.

2.3.3 Monads (`Control.Monad`)

A *monad* is a concept from a branch of mathematics known as *category theory*. In the context of Haskell, however, it is best to think of a monad as an *abstract datatype* of actions [14]. A convenient `do` notation for writing monadic expressions is provided in Haskell. This often makes monadic code look similar to code written in an imperative language. This code describes a computation sequence

Fragment 2.2 Definition of the `Game` type class with an encapsulated data type. An instance of the `Game` type class for `Pacman`.

```
1 --Block 1
2 class Game a
3   data Player a :: *
4   allPlayers :: [Player a]
```

```
1 --Block 2
2 instance Game Pacman
3   data Player Pacman = Pacman
4                      | Ghost Int
5
6   allPlayers = Pacman : (map [1..4] Ghost)
7   -- (Pacman : (map [1..4] Ghost)) evaluates to
8   -- [Pacman, Ghost 1, Ghost 2, Ghost 3, Ghost 4]
```

where multiple values are implicitly passed through each step of the sequence. The monad wraps up this sequence of computations for execution at some other time. This lends monads to combining pure calculations with impure features like *Input/Output* (*I/O*).

In section 3.2.3, the need for a `choose'` function (fragment 2.3) arises. This function is an good example of a particular monad, the *list monad*. `choose'` takes a list of lists of `b`s, it returns all of the possible ways of choosing one `b` from each of the input lists (see example output for clarity). It is not immediately clear how to solve this problem in a conventional recursive way (although it is possible). The key to the list monad approach lies on line 4. At this point, `a` has type `[b]` and `choose' mss` has type `[[b]]`. The ← breaks away this outer list and simultaneously assigns every list in `choose' mss` to `a`. This myriad of values of `a` are then implicitly passed to the next line. This gives rise to a myriad of `map (:a) mss`. These values are then wrapped up once again in a list, forming the result of the `do` block.

Aside from the this and the I/O applications, monads come in useful in many ways. Section 2.3.6 explains how *state transformer monads* can be used to make code more efficient. Section 3.1.1 shows how the generation of arbitrary values of a given type can be wrapped up in a *generator monad*.

2.3.4 Lazy Evaluation

Haskell is a *Call by Need* language; it uses a lazy evaluation strategy. Expressions are not computed in the order that they are written. Instead, a *Thunk* is stored which can be expanded later, if required. This feature of the language allows the Expansion phase of MCTS to be implemented easily. A data structure that is a conceptual representation of the entire game tree can be created. This would

Fragment 2.3 Illustration of the list monad using the `choose'` function and example output.

```
1  choose' :: [[b]] → [[b]]
2  choose' [] = [[]]
3  choose' (ms:mss) = do
4                       a ← choose' mss
5                       map (:a) ms
6
7  --note that (:a) is syntactic sugar for λx→(x:a)
8
9  -- choose' [] = [[]]
10 -- choose' [[]] = []
11 -- choose' [[1],[2]] = [[1,2]]
12 -- choose' [[1,2],[3,4]] = [[1,3],[1,4],[2,3],[2,4]]
13 -- choose' [[1,2],[0],[3,4]] = [[1,0,3],[1,0,4],[2,0,3],[2,0,4]]
```

internally be represented by a Thunk. When the root node of this tree is required, the Thunk is internally expanded into the root node where all of its children are represented as Thunks. In a strict language, attempting to build the entire game tree would be computationally unfeasible. An implementation of Expansion would therefore be needed which expands nodes of the game tree as they are required. This is essentially a less general implementation of Haskell's Thunks.

2.3.5 GHC Profiling

Despite improvements in GHC, performance is initially likely to be poor compared to *C* code. It is possible to more efficient fragments of code, but making arbitrary functions more efficient is not regarded as good practice. The largest increase in performance from a program can be extracted by making the most frequently run function more efficient. Profiling in GHC is performed by setting a flag at compile time. It gives a full breakdown of which functions consume most CPU time. This can be used to identify the functions that need to be made more efficient.

2.3.6 The State Transformer Monad (`Control.Monad.ST`)

The `ST` monad allows programmers to work safely with mutable state [13]. This enables the use of mutable data structures in Haskell. An immutable array, for example, can be *thawed* to a mutable array. This mutable array can be modified in place and then *frozen* into a an immutable array once all mutations have been performed. The critical difference between the `IO` and `ST` monads is that there is no (safe) function of type `IO a->a`. Once a value is encapsulated in the `IO` monad, it cannot be used in pure code again. There is, however,

a `runST::(∀s.ST s a -> a)` function. As a result, functions which compute values inside the `ST` monad can be broken out and used by pure functions, i.e. functions not inside the `ST` monad.

2.3.7 Unboxed Arrays

In Haskell, almost everything is a value, one might have an array of complex data structures, functions, computations wrapped up in monads. In addition, since Haskell is call-by-need, each of these things may only be partially evaluated. Therefore, arrays of simple types, such as `Int`s have this unnecessary overhead supporting unneeded features. For this reason, Haskell suports *unboxed arrays* and *unboxed mutable arrays*. These are implemented more like a C array, as a sequential block of memory containing sequential concrete values as opposed to pointers to values.

2.3.8 Random Behaviour using `System.Random`

It is essential that certain functions have an element of chance to their behaviour in this project. Functions in Haskell have to be mathematically pure so this is not possible implicitly. Fragment 2.4 shows how functions with pseudo-random behaviour can be implemented. Both functions simulate a game through to completion and return the result of the game when a terminal state is reached. `doSimulation` is unable to perform a statistically random Simulation since it can only pick each move based on some deterministic Selection. `doSimulation'` can be statistically random as it can base its Selection on the random seed. `doSimulation'` also returns a seed so that it can be passed into other functions. The initial seed of this chain of psuedo-random functions comes from the `main` function of the agent the user implements. When inside the `IO` monad, the user has access to the operating system's entropy pool.

Fragment 2.4 Random seed example using `doSimulation`

```
1 doSimulation :: Game a ⇒ a → GameState a
2 doSimulation game = ...
3
4 --the StdGen type represents a random seed
5 doSimulation' :: Game a ⇒ a → StdGen → (GameState a,StdGen)
6 doSimulation' game gen = ...
```

2.3.9 Testing using `QuickCheck`

QuickCheck is set apart from tools like *JUnit* (a unit testing framework for *Java*) in that there is no need to write any test cases, hundreds are automatically generated. Properties are written to check that certain statements about

functions hold. The conventional name for a function which conjoins all properties of a function, `fun`, is `prop_fun`. Fragment 2.5 shows the test code for `choose :: Int -> [a] -> [[a]]`, `choose k xs` returns all of the combinations of `k` elements from list `xs`. The two properties tested are:

1. `choose k xs` should evaluate to a list of length $\binom{n}{k}$, where n is the length of list `xs`
2. `choose k xs` is a list of lists, where each of the inner lists has length `k`

Testing Functions Taking User Defined Data Types

Whenever QuickCheck tests a `prop_*` function it needs to generate arbitrary values for all the arguments based on the type signature. This is predefined for embedded data types such as `Int` and `Bool`. However, more complex, user defined types must be made an instance of the `Arbitrary` type class. An example of this can be found in fragment 3.2.

Fragment 2.5 `prop_choose`

```
1  prop_choose :: Int → [Int] → Property
2  prop_choose k xs = (k' ≤ length xs') ⟹ (prop_len k' xs')  .&&.
3                                          (prop_memb k' xs')
4    --Skip test cases where k' is less than the length of xs' because
5    --evaluating choose with these arguments would cause an exception
6    --Test the remaining test cases with prop_len and prop_memb
7    where
8     prop_len :: Int → [a] → Property
9     prop_len k xs = printTestCase
10    "Checking outer list has length ‘n choose k‘ where n
11    is length of input list and k is the number of elements to be chosen"
12    $ (chooseN (length xs)  k) == (length $ choose k xs)
13    --QuickCheck will output this text if the property fails
14
15    prop_memb :: Int → [a] → Property
16    prop_memb k xs = printTestCase
17    "Checking inner lists all have length equal to the
18    number of elements being chosen"
19    $ all ((==k) . length) $ choose k xs
```

2.3.10 Module system

Modules provide a way to group related functions. The modules are usually not entirely independent and rely on data structures and functions from other modules. To enable this, modules can `import` functions from other modules.

However, module `A` can only import function `fun` from module `B` if `B` exports `fun`. This is called information hiding. Information hiding is especially important when implementing a library since it is not desirable to pollute the user name space or provide access to confusing internal library functions.

In the *Object Oriented Programming* (*OOP*) paradigm, information hiding is provided by access level modifiers (`private x`, `public y`, `protected z` etc).

Documentation Generation using Haddock

Haddock is a tool similar to *Javadoc*, a documentation generator for Java. Documentation for arbitrary Haskell code can be generated automatically. Annotations, in the form of comments can be used to add explanation to the automatically generated documentation.

Clear documentation is especially important for a library interface. Without good documentation, it is difficult to establish how to use a library. See Appendix B for the documentation generated for the MCTS library interface.

2.4 Designing the Library

Universal Modelling Language (*UML*) is commonly used in the OOP paradigm. To the best of my knowledge, no UML-like modelling techniques exist for the functional paradigm. This makes sense because just writing out type signatures and data types is a good way to model programs, that's the approach taken here. Although the code fragments are valid Haskell, they should be read as if written in a modelling language.

2.4.1 Modular structure

Figure 2.2 shows the modules in the library. The user can import from any module. If the user wishes to get only basic `MCTS` functionality they import `MCTS`. `MCTS` defines nothing itself, but collects functions from other modules and groups them together for convenience. To configure the Selection, Backpropagation and Expansion phases of the search the user would import the `MCTS.Config` module. This design may seem somewhat convoluted since it seems that `MCTS` exports almost all of the functions exported by `MCTS.Core`, `MCTS.Game` and `MCTS.Config`. The usefulness of this design is demonstrated in the implementation chapter as more functions are exported from these modules. The `MCTS.Sample.*` modules are included for testing purposes, to save space, they would not be distributed with the library.

The `MCTS.Game`, `MCTS.Config` and `MCTS.Core` modules are detailed in fragments 2.6, 2.7 and 2.8 respectively. These modules are also thoroughly documented in appendix B. The `MCTS.Sample.*` modules are defined in the imple-

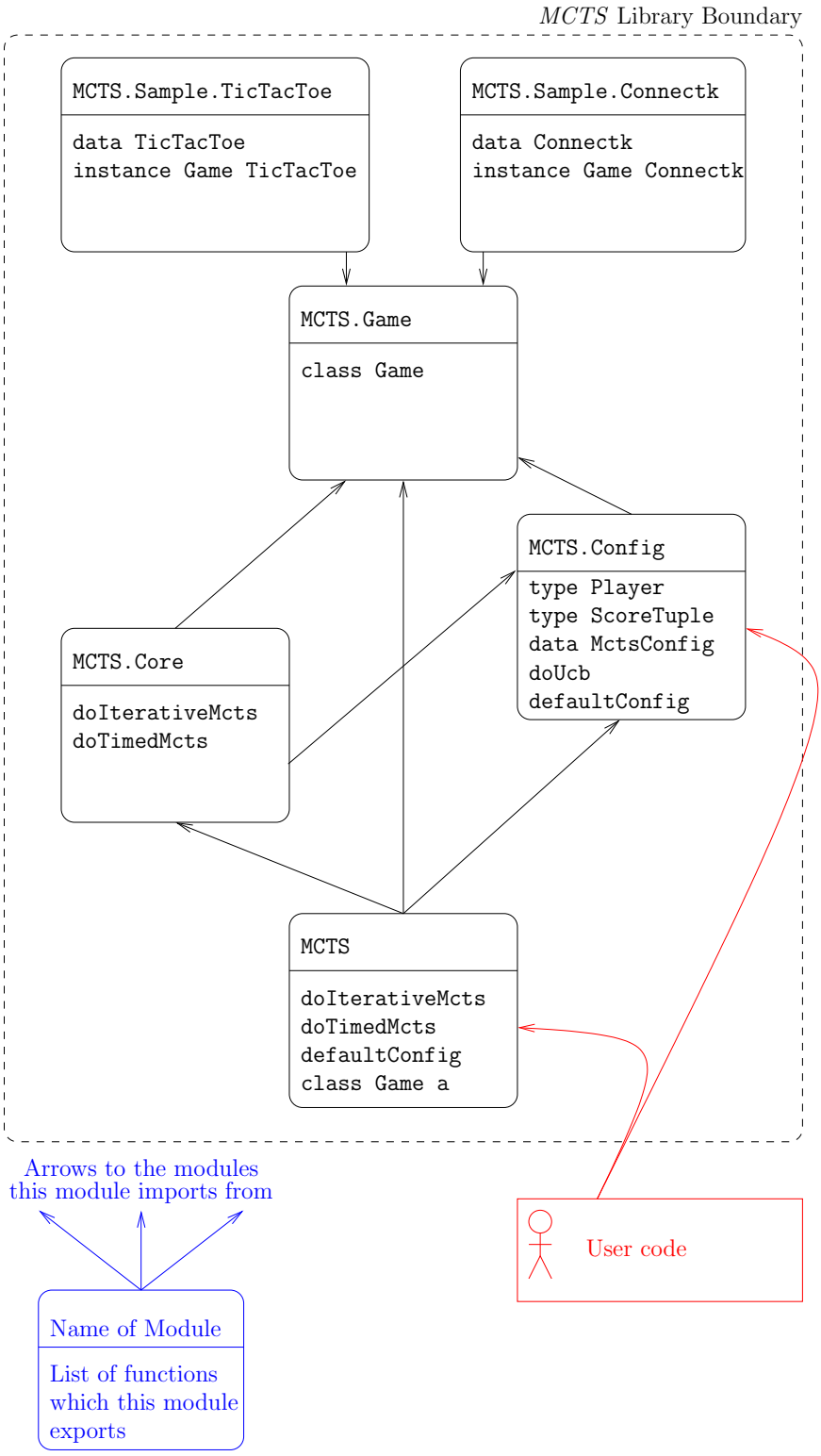


Figure 2.2: Module import/export graph

Fragment 2.6 Detail of the `MCTS.Game` module

```
1  module MCTS.Game where
2  import System.Random
3
4  class Game a
5    data Player a :: *
6    data GameState a ::*
7    allPlayers :: [Player a]
8    legalChildren :: a → [a]
9    currentPlayer :: a → Player a
10   currentState :: a → GameState a
11   doSimulation :: a → StdGen → GameState a
12
13   -- Defining the default implementations
14   data Player a = X | O
15
16   data GameState a = InProgress
17                    | Stale
18                    | Win (Player a)
19
20   allPlayers = [X,O]
21   doSimulation = --see Implementation section
```

Fragment 2.7 Detail of the `MCTS.Config` module

```
1  module MCTS.Config where
2  import MCTS.Game
3
4  type Plays = Int
5  type ScoreTuple a = [(Player a,Double)]
6  --the score tuple is represented as a list of (player, score) pairs.
7
8  data MctsConfig = MctsConfig {
9    expandConst :: Int
10   doSelection :: Game a ⇒ [(ScoreTuple a,Plays)]
11                         → Player a → Int
12   --Int in return type is the index of the selected tuple in the list
13
14   doBackpropagation :: Game a ⇒ GameState a
15                               → (ScoreTuple a, Plays)
16                               → (ScoreTuple a, Plays)
17 }
18
19 defaultConfig :: MctsConfig
20 --Provides default implementations of the functions in MctsConfig
21
22 doUcb :: Game a ⇒ Int → [(ScoreTuple a, Plays)]
23                 → Player a → Int
24 --Lets user specify what constant of exploration, c,
25 --should be used for UCB if not the default, 1.
```

Fragment 2.8 Detail of the `MCTS.Core` module.

```
1  module MCTS.Core where
2
3  import MCTS.Game
4  import MCTS.Config
5  import System.Random
6
7  doIterativeMcts :: Game a ⇒ a → MctsConfig → StdGen → Int
8                             → (a,StdGen)
9  --The Int is the number of iterations
10
11 doTimedMcts :: Game a ⇒ a → MctsConfig → StdGen → Int
12                       → IO (a,StdGen)
13 --The Int is the number of milliseconds to iterate for
14 --The IO in the result shows the result is wrapped up in the IO monad
```

mentation section. The `MCTS` module merely groups together functions from other modules, as such, there is nothing to detail.

2.4.2 Revisiting requirements analysis (cf section 2.1)

Library must allow the user to...

1. Perform a certain number of iterations of MCTS from a certain game state.
2. Perform as many iterations possible from a certain game state in a given time.

The user will be able to do either by calling one of the top level functions `doIterativeMcts` or `doTimedMcts`.

The user must be able to specify how to carry out the...

1. Selection Phase

   The user can use the expression:

   ```
   defaultConfig{doSelection = userDoSelection}
   ```

   for the `MctsConfig` argument of one of the `do*Mcts` functions to express that the search should be done as default except for selection which should use the users `userDoSelection` function. The type signature of `doSelection` is now explained. `[(ScoreTuple a,Plays)]` represents list of score tuple, number of plays pairs to select from. `Player` represents the `Player` doing the Selection and the result is the index of the input list which has been selected. The default implementation for this function should be UCB with $c = 1$. Changing the value of the exploration constant c is likely to be such a common requirement that the `doUcb` function is also exposed through the `MCTS.Config` module and can be changed thus:

   ```
   defaultConfig{doSelection = doUcb 0.25}
   ```

2. Expansion Phase

   The user can use a similar expression to modify `expandConst` to n, where $n > 0$. This specifies that for each Simulation, a maximum of n nodes are expanded in to the tree. This value will be set to 1 in the `defaultConfig`.

3. Backpropagation Phase

   The user can use a similar expression to modify `doBackpropagation` to their own function. The following explains the type signature of `doBackpropagation`. `GameState a` represents the result of a simulated game which is now being propagated back up the tree. The `(ScoreTuple a, Plays)` types represent a game theory style n-tuple, where n is the number of players. The function updates the n-tuple based on the result of the

Simulation. For example, a result of `Win X` might lead to an increase in the score for `X` in the tuple by 1 and a decrease by 1 for all other players, while a result of `Stale` might leave the tuple exactly the same.

4. Simulation Phase

   Since Simulation is tightly coupled with the actual implementation of the game used, the design decision was made to make the `doSimulation` function a member of the `Game` typeclass rather than a a member of the `MctsConfig` data structure. A default implementation will still be given, but this time the user can override it by giving an implementation for it in the instance of that particular game. The function takes two arguments: a game in a certain position and a random seed. The function should use the random seed to make pseudo-random moves based on some Simulation policy until a terminal position is reached. It results in the state of that terminal position and a new random seed. The default implementation for this function is to recursively pick one of the positions `legalChildren` with uniform distribution until a terminal position is reached.

The user must be able to write agents for games with...

1. A single player

   SP-MCTS is MCTS with a Selection strategy more suited to single player games. Therefore SP-MCTS can be implemented in the MCTS library by using a custom `doSelection` function.

2. Multiple players

   An investigation of the various ways of implementing multi player games in MCTS can be found in Cazenave [4]. All depend on a tuple containing scores for all players being stored in each node of the game tree. The `ScoreTuple` type meets these requirements. What is more, since users can define their own `Player` type, they can give the players sensible names to make their implementation clearer.

3. Imperfect information

   Cianciani & Favini [6] represent imperfect information by storing only the part of the state which is known in the tree. A probabilistic guess based on a database of previous games is then made about the unknown parts of the board at Simulation time. This is possible in the MCTS library by overriding the default `doSimulation` function.

4. Simultaneous play

   These types of games can be reduced to imperfect information games where the most recent move of all other players is not known.

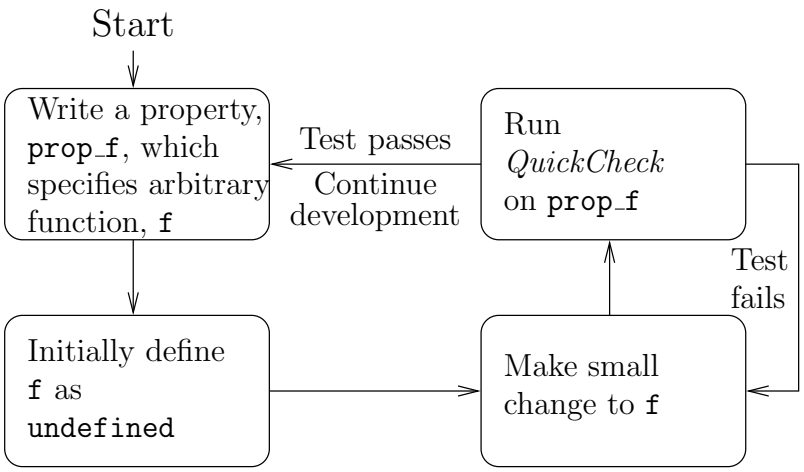


Figure 2.3: Flow diagram illustrating test driven development cycle for an arbitrary function `f`

2.5 Development Style

Functional programming projects are well suited to *test driven development* [2]. A test case or strict specification is written prior to implementing a functional. The functional unit is then developed incrementally, during which, repeated tests are conducted against the specification. Figure 2.3 shows this development cycle in the context of Haskell & QuickCheck. Note that functions are initially `undefined`. This is essentially the totally undefined function, $\perp$, from denotational semantics. It is used here since it has a completely unconstrained type signature and is therefore a valid value for any function. This acts as a base-line to incrementally develop from. Using `undefined` will allow code to be compiled and tests to run.

When an arbitrary function, `prop_f`, is found in code, the specified function, `f`, will be defined beneath it. Care must be taken never to export `prop_*` functions over the library interface, as the user may not necessarily have QuickCheck installed and it would be undesirable to have QuickCheck as a dependency to the library.

Chapter 3

Implementation

This chapter details the implementation of the MCTS library and the Connect agent created. In order to build the library in a test-driven fashion, the various elements had to constructed in a certain order. First, a concrete instance of the `Game` type class was written. *Tic-Tac-Toe* (or Connect (3,3,3,1,1)) was chosen for its simplicity. Secondly, default implementations for the Expansion, Selection, Simulation and Backpropagation phases of the search were implemented. Next, the library was completed by writing the library core. Finally, the Connect agent was written using the completed library.

3.1 Building the Library

3.1.1 Tic-Tac-Toe

Tic-Tac-Toe is isomorphic with the following game:

1. Two players take turns to choose tiles numbered 1 to 9.
2. Once a tile is picked up, the player must keep it; it cannot be picked again.
3. If a player, after his go, can add the numbers on 3 of his tiles to make 15, then he has won the game.

This isomorphism is related to the fact that all rows, columns and diagonals of a 3×3 magic square add up to 15 (and no other groups of 3 add to 15). Player X obtaining a line in the top row would be equivalent to choosing the tiles numbered 2, 7 and 6 (Figure 3.1). 2, 7 and 6 sum up to 15. This tiled number game is easier to implement in code than TicTacToe.

To implement this game such that it can be used in the library, a `TicTacToe` data type must be defined to encapsulate the current position. `TicTacToe` must be an instance of the `Arbitrary` type class so that functions which take it can be automatically tested by QuickCheck. `TicTacToe` must also be an instance of the `Game` type class so that it can be used by the MCTS library. To do this, the functions in fragment 3.1 must be implemented.

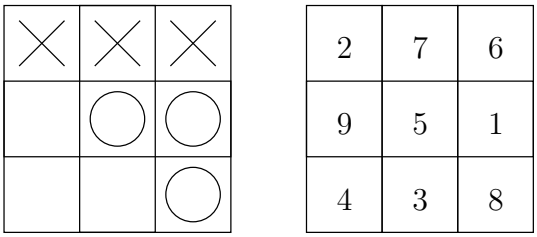


Figure 3.1: A comparison between the Tic-Tac-Toe board and the 3×3 magic square.

Fragment 3.1 The minimal set of functions needed to implement TicTacToe

```
1  module MCTS.Sample.TicTacToe where
2  data TicTacToe
3  instance Arbitrary TicTacToe
4    where
5      arbitrary :: Gen TicTacToe
6
7  instance Game TicTacToe
8    where
9      legalChildren :: TicTacToe → [TicTacToe]
10     currentPlayer :: TicTacToe → Player TicTacToe
11     currentState  :: TicTacToe → GameState TicTacToe
```

The rest of this section details the implementation of these data types and functions. `legalChildren`, `currentPlayer` and `currentState` were implemented using test-driven development; the design of the QuickCheck properties which should hold of them are discussed.

In the rest of this chapter, the following notation is used to make explanations clearer: if `a` is an expression in Haskell, then ⟦`a`⟧ is the representation of that expression in the real world.

The `TicTacToe` data type

The `TicTacToe` data type represents the current position of the game. It encapsulates three `Sets`, `sX`, `sO` and `sC`. For an arbitrary `g` ∈ `TicTacToe`, ⟦`sX g`⟧ represents the tiles X has chosen, ⟦`sO g`⟧ the tiles O has chosen and ⟦`sC g`⟧ the tiles still to be chosen.

The `arbitrary` function

To produce an arbitrary position, ⟦`sX`⟧, ⟦`sO`⟧ and ⟦`sC`⟧ can be treated as initially empty buckets. Each of the tiles from 1 to 9 can then be thrown into a random bucket. This raises the need for a function which randomly distributes a list of numbers over 3 lists. Implementing the 'randomly' part of this in Haskell

is non-trivial (see Section 2.3.8). This random behaviour could be introduced by passing a random seed in to the `arbitrary` function. However, it is more convenient to wrap up the generation of arbitrary types in a Generator monad. `split3` (fragment 3.2) wraps up the computation of non-deterministically putting each element of the input list into each of the output lists. The actual random choices are made later inside the QuickCheck library.

Fragment 3.2 Definition of the `arbitrary` function for `TicTacToe`

```
1  instance Arbitrary TicTacToe where
2           arbitrary = do (a,b,c) ← split3 [1,2,3,4,5,6,7,8,9]
3                          return $ TicTacToe (Set.fromList a)
4                                             (Set.fromList b)
5                                             (Set.fromList c)
6  split3 :: [a] → Gen ([a],[a],[a])
7  split3 [] = return ([],[],[])
8  split3 (x:xs) = do (a,b,c) ← split3 xs
9                     oneof [return (x:a,b,c),
10                           return (a,x:b,c),
11                           return (a,b,x:c)]
```

The alert reader will note that the `arbitrary` function does not always produce legal `TicTacToe` positions. This is desirable; it may be advantageous for property functions to check how a function behaves when given an illegal position. If illegal positions are disallowed, they can be filtered out.

The `currentPlayer` function

For any legal Tic-Tac-Toe position, it is always possible to tell which player's turn it is. There are two possible scenarios:

1. There are the same number of X's and O's on the board.
2. There are $n + 1$ X's and n O's on the board, where $n \in \mathbb{N}$.

In the case of item 1, it would be X's turn. In the case of item, 2 it would be O's turn.

Similarly, in the isomorphic version of Tic-Tac-Toe, if X and O had the same number of tiles, it would be X's turn. If X had one more tile than O, it would be O's turn. The following property must therefore hold of the `currentPlayer` function:

$$\forall \texttt{g} \in \texttt{TicTacToe.legal g} \implies$$
$$\texttt{currentPlayer g} = \begin{cases} \texttt{X} & \text{if } (\texttt{size \$ sX g}) = (\texttt{size \$ sO g}) \\ \texttt{O} & \text{if } (\texttt{size \$ sX g}) = (\texttt{size \$ sO g}) + 1 \end{cases}$$

where `legal::TicTacToe->Bool` decides whether a `TicTacToe` position is legal, or not, and `size::Set a->Int` gives the size of a set. This property is encoded in the `prop_currentPlayer` function. The definition of `currentPlayer`:

$$\texttt{currentPlayer g} =_{\text{def}} \begin{cases} \texttt{X} & \text{if } (\texttt{size \$ sX g}) = (\texttt{size \$ sO g}) \\ \texttt{O} & \text{if } (\texttt{size \$ sX g}) = (\texttt{size \$ sO g})+1 \end{cases}$$

is very similar to this property.

The above equations are not entirely mathematically correct. An implicit conversion needs to be made from Haskell data types to mathematical sets. For example, Haskell expressions of type `Bool` and `Int` should be thought of as members of the $\mathbb{B}$ and $\mathbb{Z}$ sets.

The `legalChildren` function

For arbitrary $\texttt{g} \in \texttt{TicTacToe}$ where `g` is a legal position, there are $[\![\texttt{size \$ sC g}]\!]$ different possible moves in $[\![\texttt{g}]\!]$. Each move represents moving a tile from $[\![\texttt{sC g}]\!]$ to $[\![\texttt{sX g}]\!]$ if it is X's turn, or $[\![\texttt{sO g}]\!]$ if it is O's turn.

A `prop_legalChildren` function checks that for all $\texttt{g} \in \texttt{TicTacToe}$, if $[\![\texttt{g}]\!]$ is a legal position, $[\![\texttt{legalChildren g}]\!]$ are all legal positions. It also checks that the list `legalChildren g` has length `size $ sC g`.

The `legalChildren g` function maps a *lambda* function (also known as an *anonymous* function) onto the `sC g` set. This lambda function produces a position identical to $[\![\texttt{g}]\!]$ but with a tile moved from $[\![\texttt{sC g}]\!]$ into $[\![\texttt{sX g}]\!]$ or $[\![\texttt{sO g}]\!]$ depending on the value of $[\![\texttt{currentPlayer g}]\!]$.

The `currentState` function

This function is specified by the following:

$$\forall \texttt{g} \in \texttt{TicTacToe.legal g} \implies$$

$$\texttt{currentState g} = \begin{cases} \texttt{Win X} & \text{if any 3 tiles from } [\![\texttt{sX g}]\!] \text{ sum to 15} \\ \texttt{Win O} & \text{if any 3 tiles from } [\![\texttt{sO g}]\!] \text{ sum to 15} \\ \texttt{Stale} & \text{if } [\![\texttt{sC g}]\!] \text{ contains no tiles} \\ \texttt{InProgress} & \text{otherwise} \end{cases}$$

Having the 'if any 3 tiles...' case raises the need for a `choose` function. This must be able to find all of the different 3 elements can be chosen from a list. This function is specified in Section 2.3.9.

3.1.2 Implementing Default MCTS Phases

This section outlines the implementation of `doSimulation`, `doBackpropagation`, `doSelection` and `expandConst` (fragment 3.3). Several helper functions were

also written. It is probable that users wishing to define their own configuration functions might find these functions useful. These functions were added to the export list of the `MCTS.Config` module (fragment 3.4).

Fragment 3.3 Reminder of parts of the `MCTS.Game` and `MCTS.Config` modules

```
module MCTS.Game where
class Game a
  where
    -- ...
    doSimulation :: a → StdGen → (GameState a, StdGen)
```

```
module MCTS.Config where

type ScoreTuple a = [(a,Score)]
type Score = Double
type Plays = Int

defaultConfig :: MctsConfig

doUcb :: Game a ⇒ Double → [(ScoreTuple a, Plays)]
                         → Player a
                         → Int

data MctsConfig = MctsConfig {
  expandConst :: Int
  doSelection :: Game a ⇒ [(ScoreTuple a,Plays)]
                        → Player a → Int

  doBackpropagation :: Game a ⇒ GameState a
                               → (ScoreTuple a, Plays)
                               → (ScoreTuple a, Plays)
}
```

Simulation

The default implementation of `doSimulation` is to recursively pick a random, legal child of the position being simulated. When a terminal position is reached the `currentState` of this position is returned. As part of the implementation, `pick` (see fragment 3.4) was written. `pick` generates a random number i, where $0 < i < n - 1$ and n is the length of the input list. It returns the i^{th} element of the input list. `pick` is exported by the `Game` module, it could be useful to a user implementing their own `doSimulation` function.

Fragment 3.4 Functions added to the export list of the `MCTS.Config` module.

```
1  module MCTS.Config where
2
3  -- ... functions defined earlier ... --
4
5  pick :: [a] → StdGen → (a, StdGen)
6
7  readTuple :: Eq a ⇒ a → ScoreTuple a → Score
8
9  update ::
10 (a → Bool) → --boolean property of a
11 (Score → Score) → --mapped onto scores of players satisfying property
12 (Score → Score) → -- ‘‘    ‘‘      ‘‘    ‘‘   not satisfying property
13 ScoreTuple a → --the input score tuple
14 ScoreTuple a --the output score tuple
```

Selection

Recall the UCB equation:

$$Q_i = v_i + c \times \sqrt{\frac{\ln N}{n_i}}$$

where v_i is the minimax score of node i, n_i is the number of times this node has been explored on previous iterations, N is the number of times the current node has been explored and c is an ‘exploration’ constant.

The default implementation of `doSelection` is to choose the node i which maximises the UCB score Q_i when $c = 1$. `doSelection` is defined in terms of a more general function `doUcb`, `doSelection = doUcb 1`. `doUcb` is exposed to the user through the `Config` module since modifying the exploration constant is likely to be a common use case.

Since `doUcb` is given no information about the current node, the value of N is unobtainable directly. Instead, the $N = \sum_i v_i$ equivalence is exploited. This could be avoided by passing in the details of the parent node to the function, however, this would make the public interface more complex. Computing $N = \sum_i v_i$ in this manner does have a small overhead. However, all values of v_i are already considered to select the most desirable node. Therefore complexity is increased only by a small constant factor.

Backpropagation

The default implementation of `doBackpropagation` performs simple case analysis on the `GameState` passed in as the first argument:

> `InProgress` - result in the identity function. In other words, leave the score tuple and play count alone.

One might question why it is necessary to backpropagate an `InProgress` result. Backpropagation should only occur once a Simulation has reached a terminal state. Although the default implementation of `doSimulation` will never return `InProgress`, a user's implementation is free to do so. Therefore, this case must be handled.

`Stale` - result in the function which increments the play count by 1 and leaves the score tuple alone.

`Win x` - result in the function which increments the play count by 1, increases the score of player `x` by 1 in the score tuple, and decreases the score of all other players by 1.

An `update` helper function was written (see fragment 3.4) to assist with the `Win x` case. This function is a conditional map. It maps a certain function (e.g. $+1$) on to the scores of players that satisfy a certain property (e.g they were on the winning team) and a different function (e.g. -1) onto the scores of all other players. The `prop_update` specifies that for an arbitrary `ScoreTuple (Player TicTacToe)` applying arbitrary $+/-$ `Int->Int` functions produces the expected results.

Expansion

The default implementation of the Expansion phase should be to add a single node to the tree. Therefore, by default, `expandConst = 1`.

3.1.3 Library Core

Fragment 3.5 shows an updated version of the `Core` module. The additional functions are revealed to the user should they wish to perform advanced search techniques requiring direct access to the tree structure. `data MCT a` is the data type representing a *Monte Carlo Tree*, the internal tree generated during MCTS. `selectBestMove` is the function used at the root node, after all iterations have been performed, to decide which move is most desirable. `expand` lazily builds a full game tree from a given start position. `mcts` performs one iteration of MCTS on a partial tree.

The `doIterativeMcts` and `doTimedMcts` functions

These functions are very similar. They both:

Use `expand` on the game position they are given to build an empty game tree.

Perform a number of MCTS iterations by repeatedly running `mcts`.

Fragment 3.5 The updated textttMCTS.Core module.

```
1  module MCTS.Core where
2  doIterativeMcts :: Game a ⇒
3                     a → MctsConfig →
4                     Int → StdGen →
5                     (a,StdGen)
6
7  doTimedMcts :: Game a ⇒ a → MctsConfig →
8                           StdGen → Int →
9                           IO (a,StdGen)
10
11 data MCT a = MCT {rGame::a
12                  , rPlayed::Int
13                  , rT::ScoreTuple
14                  , rChildList::[MCT a]}
15
16 selectBestMove :: Game a ⇒ MCT a → a
17
18 expand :: Game a ⇒ a → MCT a
19
20 mcts :: Game a ⇒ MCT a →
21                   MctsConfig →
22                   StdGen →
23                   (MCT a, GameState a, StdGen)
24
25 --all functions below this line are private
26
27 instance Arbitrary (MCT TicTacToe)
28   where
29     arbitrary :: Gen TicTacToe
30
31 (~!!~) :: [a] → Int→ (a,[a])
```

Use `selectBestMove` on the `MCT` generated to decide the best move that the root node should pick.

`doTimedMcts` was interesting because use of strict evaluation was needed. The initial implementation checked the system clock to get a timestamp, t_0. An internal function recursively checked the system clock to get a timestamp, t and performed an iteration of MCTS. This recursion bottomed out when the agent had consumed all available thinking time: $t - t_0 >= t_{\text{thinking}}$. However, due to lazy evaluation, this function spent all thinking-time getting deeper in the stack. Only once all thinking-time had been consumed would it start performing MCTS iterations. This resulted in the function having a much longer run-time than the thinking-time allocated. Making one of the function applications strict solved this problem and made the function behave properly. This behaviour could not be detected by QuickCheck since QuickCheck properties can only be written for pure functions. `doTimedMcts` is not a pure function as it relies on obtaining a system timestamp.

The `MCT` data type

The definition of this follows from the fact that each node in the tree must store: the current game position; a count of how many times this node has been simulated; a score tuple representing the current estimate of utility at this node; a list of the child nodes to this node.

The `arbitrary` function

This function returns a Generator for an arbitrary `MCT TicTacToe`. An arbitrary `TicTacToe` start position is generated using the `arbitrary` function that was defined for `TicTacToe` in fragment 3.2. The `ScoreTuple` for each node contains `allPlayers` of `TicTacToe`, has zero sum, and is otherwise random. The tree itself is built recursively from the start position using the `legalChildren` for the game at each node. The maximum depth of the tree is related to the `size` parameter of QuickCheck. At each step of the recursion, the maximum depth of each child tree is a random value between 1 and the maximum depth of the current tree. This generates asymmetric trees which are likely to be generated by MCTS.

The `selectBestMove` function

`selectBestMove` differs from `doUcb` in that `doUcb` will weight unexplored nodes higher. This is done in the hope of visiting unexplored areas of the tree. `selectBestMove` should greedily select which child of the root node represents the most promising move. Counter intuitively, this should be the node with the largest play count, rather than the node with the largest average score for that player [7].

The `expand` function

As discussed earlier, `expand` simply needs to build the whole game tree. Lazy evaluation ensures that nodes are actually expanded only as they are needed.

The `mcts` function

`prop_mcts` is the conjunction of several properties for trees `A` and `B`, where `B` results from running `mcts` on `A`:

1. `B` has exactly one more `node` where $[\![$`rPlayed node`$]\!] \neq 0$ than `A`.
2. The sum of all `GameTuple`s are conserved from `A` to `B`.
3. In `A`, there is a path from the root node to a leaf, for which each node has a `rPlayed` value exactly 1 greater than the equivalent path in B.
4. `A` and `B` are otherwise identical.

Item 1 checks that the Expansion phase of MCTS behaves as expected. Items 2 and 3 check that results are backpropagated correctly. Item 4 is a general sanity check.

`mcts`, given an `MCT` game tree, recursively selects one of its children based on the policy provided by `doSelection`. This recursion bottoms out when either:

1. A node which is in a terminal state is selected.
2. A node which has never been selected before is selected.

In the case of item 1, this result is backpropagated up the tree. This is performed by modifying the `rPlayed` and `rT` records for each node as the stack is unwound. The `doBackpropagation` function specifies the way that `rPlayed` and `rT` are updated. In the case of item 2, this new node is simulated using `doSimulation`. Backpropagation then occurs as specified in item 1. Since `doSelection` simply returns the index of the child node which should be selected, a function was required to split the child list into the selected node and the rest of the list. For this purpose, the `~!!~` function was defined. It is defined using the built-in Haskell functions `!!` and `delete`.

3.2 Connect agent

This section describes the implementation of the Connect agent using the MCTS library. The intention is to implement the agent using as little domain knowledge as possible. Defaults are used for all stages of MCTS. For the subclass of Connect(n,m,k,p,q) where $n = m = k$, an $n \times n$ magic square isomorphism can be used. However, in general, this technique cannot be applied. For the Connect agent, a different implementation is used.

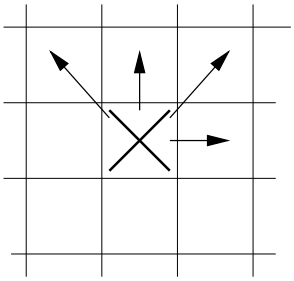


Figure 3.2: The directions which must be checked for all squares on the board to identify a connect-k.

This section describes Connect in the way it was implemented, slightly different from the description in the introduction. For example, the players are X and O as opposed to Black and White, and the data structure representing a Connect game is called `Connectk`. The differences are purely cosmetic.

3.2.1 Naive Implementation

Finding k-in-a-row

1. For each cell containing X on the board:
   (a) Check the $k-1$ cells in the directions shown in Figure 3.2.
   (b) If any of the directions have $n-1$ consecutive Xs then X has a k-in-a-row and has won the game.
2. Repeat step 1 replacing X for O.

To implement this in an imperative language, a two-dimensional array could be used represent the board. A k-in-a-row could be detected by scanning $k-1$ steps in each direction by iterating over the indexes of the array. However, since lists are usually preferred in functional programming, a list based representation is used (Figure 3.3). This is a slight simplification, _ cannot be used here since Haskell reserves the _ keyword for a different purpose. There are two possibilities for representing an unoccupied square: add an `Empty` player to the `Player Connectk` type; or make the list of lists have type `[[Maybe (Player Connectk)]]`. The latter option is used in this project since the type signature makes it clear that each square on the board may have a player in it, or may be empty(see Fragment 3.6).

Checking for a k-in-a-row is harder using this list representation than it would be using an array. It is relatively easy to check the east direction by `fold`ing each list checking for k consecutive pieces for one player. It is less obvious how to check the other directions in figure 3.2. The problem can be solved by checking the easterly direction for k-in-a-row on 6 different transformations of an initial board, A (figure 3.4). S,T and R represent:

Fragment 3.6 List representation of board in Figure 3.3 and definition of the `Maybe` data type (`Maybe` is a built in type [14]).

```
1 data Maybe a = Just a
2              | Nothing
3
4 connectkBoard =
5 [[Just X, Nothing,Nothing,Just O],
6  [Just X, Just X, Just O, Nothing],
7  [Nothing,Just O,Nothing,Nothing],
8  [Nothing,Nothing,Nothing,Nothing]]
```

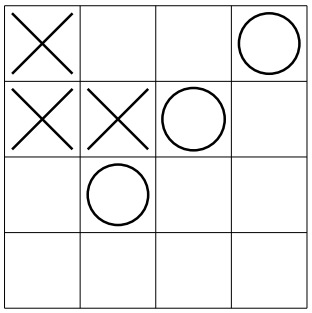


Figure 3.3: `[[X,_,_,O],[X,X,O,_],[_,O,_,_],[_,_,_,_]]` represented graphically

S - Shift all of the rows left by their row index, where the first row has index 0.

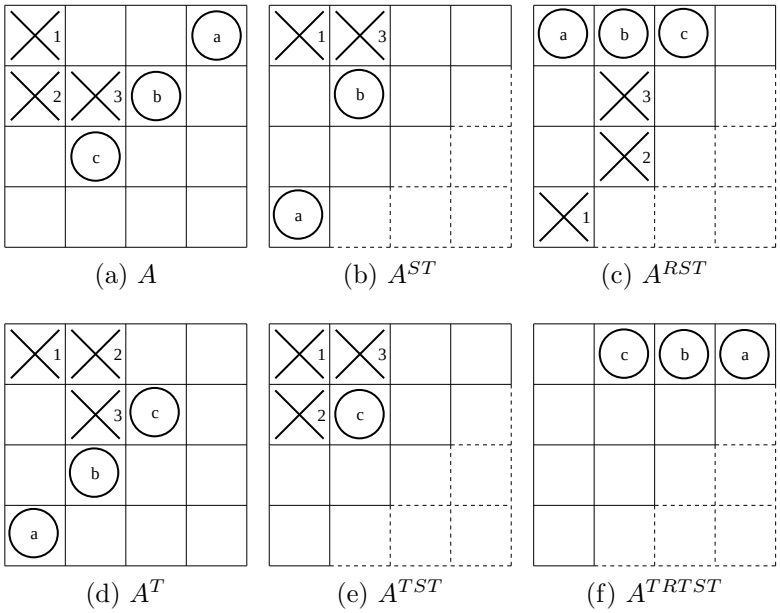
T - Standard matrix transposition.

R - Reverse each of the rows.

Relating this to figure 3.2: (a) and (d) represent searching the east and north directions respectively; (b) and (e) represent searching the north-west diagonals; (c) and (f) represent searching the south-west diagonals. Note how the abc 3-in-a-row comes up twice for O because both of the long diagonals are checked by two different transformations.

Representing the current player

For Tic-Tac-Toe, the argument that it is O's turn if both players have the same number of pieces on the board was used to decide the current player. For Connect, this does not hold because boards with arbitrary values of p and q must be considered. In this implementation, the problem is solved by exploiting lazy evaluation. The move sequences for games are represented by infinite player-lists (Fragment 3.7). The head of the player-list represents the current player (`currentPlayer g = head $ playerList g`) and the tail of the player-list is the player-list for all of the legal children of the position.



(a) A (b) A^{ST} (c) A^{RST}

(d) A^{T} (e) A^{TST} (f) A^{TRTST}

Figure 3.4: A depiction of the transformations required to find k-in-a-row in board A

Fragment 3.7 Demonstration of how to produce different `playerLists` for various values of the Connect parameters p and q

```
1 --p=1;q=1 ... [X,O,X,O...]
2 playerList1 = X:O:playerList1
3
4 --p=2;q=2 ... [X,X,O,O,X,X,O,O...]
5 playerList2 = X:X:O:O:playerList2
6
7 --p=2;q=1 ... [X,O,O,X,X,O,O,X,X...]
8 playerList3 = tail playerList2
```

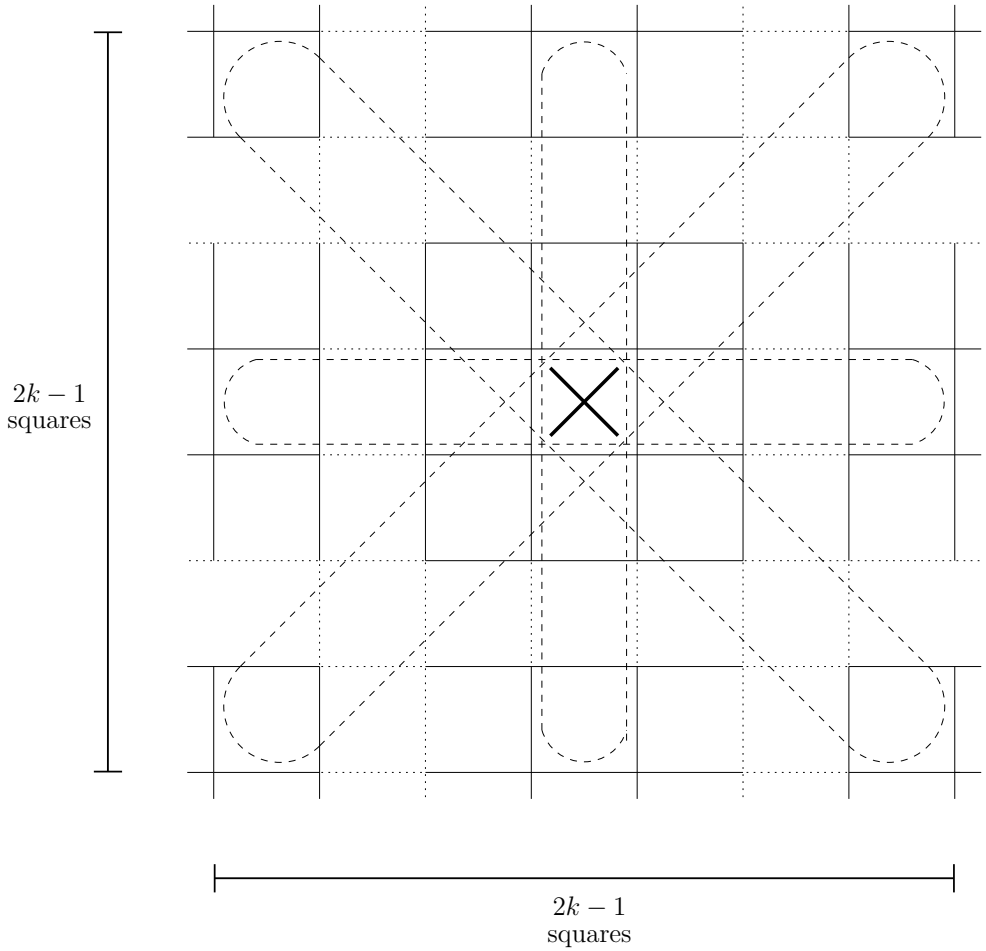


Figure 3.5: Illustration of the lines which need to be checked for a k in-a-row after each move.

3.2.2 Improving Simulation

Analysis performed in the Evaluation chapter shows that the agent detailed above, labelled *Slow*, does not perform competitively. Profiling data (Appendix A) for an empty 6×6 board suggested the Simulation code should be improved. This makes sense for two reasons:

1. Every time a move is made during Simulation, copies of linked lists are made. There is no conceptual need to take these copies since there is no need to store any of the mid-Simulation positions reached. In an imperative language, this would not be a problem, as there would just be a single mutable array and one element would be updated every time a move was made.

2. Every time a move is made, the entire board is checked for k in a row for both players. This involves checking $O(n^2)$ squares.

Item 1 can be solved by using unboxed mutable arrays inside the `ST` monad. Item 2 can be solved by being more intelligent about which squares are checked. It

is possible to find all k-in-a-rows by checking only $O(k)$ squares on each move[1]. Figure 3.5 shows which lines need to be checked, assuming, without loss of generality, that X has just moved into the centre square of figure 3.5. The following paragraph justifies this statement.

Assume arbitrary, legal, $\texttt{g}, \texttt{g'} \in \texttt{Connectk}$ where $[\![\texttt{g}]\!]$ is $[\![\texttt{InProgress}]\!]$, it is X's turn in $[\![\texttt{g}]\!]$ (without loss of generality), and for arbitrary $\texttt{m} \in \texttt{Move}$, where $[\![\texttt{m}]\!]$ is an empty square in $[\![\texttt{g}]\!]$, $[\![\texttt{g'}]\!]$ is $[\![\texttt{g}]\!]$ after move $[\![\texttt{m}]\!]$. Checking the lines in figure 3.5 centred around move $[\![\texttt{m}]\!]$ in position $[\![\texttt{g'}]\!]$ to decide weather or not X wins the game, is equivalent to using the `currentState` function. *Informal Proof:*

1. Neither player has a k-in-a-row in $[\![\texttt{g}]\!]$ since if they did it would contradict the assumption that $[\![\texttt{g}]\!]$ is $[\![\texttt{InProgress}]\!]$.

2. If, after X moves, a k-in-a-row is detected in one of the lines in figure 3.5 then $[\![\texttt{currentState g}]\!]$ is $[\![\texttt{Win X}]\!]$ since that k-in-a-row also represents a k-in-a-row in $[\![\texttt{g}]\!]$. In this case, using `currentState` and checking figure 3.5 produce consistent results.

3. If, after X moves, no k-in-a-row is detected in one of the lines in figure 3.5 then $[\![\texttt{currentState g}]\!]$ is either $[\![\texttt{InProgress}]\!]$ or $[\![\texttt{Stale}]\!]$. It cannot be $[\![\texttt{Win O}]\!]$ since O did not have a k-in-a-row before X moved (item 1) and the state of O's pieces have not changed since. It cannot be $[\![\texttt{Win X}]\!]$ since X did not have a k-in-a-row before X moved (item 2) and none of the lines involving this most recently added piece contain a k-in-a-row (item 2). In this case, using `currentState` and checking figure 3.5 produce consistent results.

The assumption that the arbitrary game positions must be in progress is valid for two reasons: the core of the library only ever simulates games which are in progress; as soon as the `doSimulation` function reaches a terminal state, Simulation terminates.

Arrays and Bounds

One of the problems of programming with arrays is that it is necessary to consider that arrays are not infinite and have bounds. This can be a source of off-by-one errors and other bugs. In this project, rather than considering bounds, all functions access arrays using custom array accessors. Whenever a function requests a value from an out of bounds location, a `Dummy` player is returned. Conceptually, the board is infinite and consists of a small legal playing area surrounded by an infinite field of `Dummy` values.

[1] O notation may not be the best to use here since boards usually are quite small. For real world existing games, not larger than $n = 19$, $k = 6$. However if we look at the exact number of squares checked: $6n^2$ and $8k - 4$, the later is still clearly better.

`Dummy` is a player which...

Does not have an entry in the `ScoreTuple` in the game tree.

Is unable to win a game.

Is able to block other players from winning.

A function attempting to search for a k-in-a-row which strays off the board is immediately blocked by the `Dummy` values.

Unboxed Mutable Arrays

The type `Maybe (Player Connectk)` was used to represent the value of a square in the naive implementation. This is not suitable for an unboxed array since it is a complex data type. To solve this problem, `Maybe (Player Connectk)` was made an instance of the `Enum` class. This provides bijection from `Maybe (Player Connectk)` to `Int`. In particular:

$$\texttt{Nothing} \to -1$$
$$\texttt{Just X} \to 0$$
$$\texttt{Just O} \to 1$$
$$\texttt{Just Dummy} \to 2$$

The modified array accessors introduced in the previous section deal with this conversion. Functions never deal with the arrays directly. This is important since storing the board as an array of `Int`s could lead to type safety issues. For example, imagine a function which sums all the elements in the array. This function would type and compile correctly in Haskell, even though it does not make sense to sum a list of `Maybe (Player Connectk)`s.

Improved Copde

Several monadic functions were written to implement the improved `doSimulation` code: `checkNewAddition` checks all the lines as per figure 3.5; `doMove` places a piece for a particular player into a mutable array; `emptySquares` lists all of the empty squares in a mutable array. The following QuickCheck property specifies the functions based on their equivalence to the existing function, `currentState`:

$$\forall \texttt{g} \in \texttt{Connectk}, \texttt{m} \in \texttt{Move.currentState g} = \texttt{InProgress} \implies$$
$$\texttt{m} \in \texttt{emptySquares g} \implies$$
$$\texttt{checkNewAddition m \$ doMove g m p} =$$
$$\begin{cases} True & \texttt{currentState \$ doMove g m p} = \texttt{Win \$ currentPlayer g} \\ False & \texttt{currentState \$ doMove g m p} = \texttt{InProgress} \\ False & \texttt{currentState \$ doMove g m p} = \texttt{Stale} \end{cases}$$

3.2.3 WinSave Heuristic

Recall that the rules of Connect specify that a parameter p specifies the number of moves per turn. A 'turn' is a sequence of consecutive moves for one player. A 'move' is the action of a player marking a single square with their symbol (O or X).

As will be shown in the Evaluation chapter, the agent as described so far, *Medium*, does not perform well against more conventional agents. Profiling showed execution time was distributed fairly uniformly over many functions. This suggested that there was no major bottleneck and further optimization would be a major task. A common, alternative approach to improving efficiency, and therefore number of iterations per unit time, is to improve the quality of the search by altering the Simulation policy [3].

In the agent as described so far, the Simulation phase chooses completely random moves. This does not represent rational opponents well. A more realistic way of simulating games might be to use the policy:

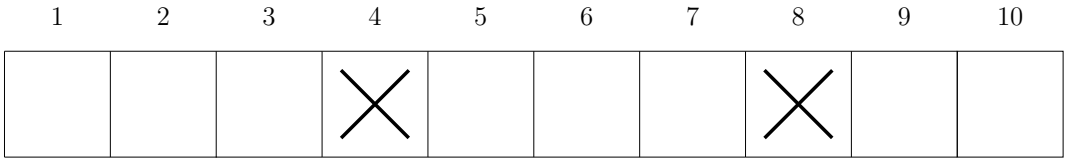
1. If current player can win this turn then she should play the winning moves.
2. If both
   (a) The opponent can win when it becomes his turn.
   (b) The current player is able to block the opponent's win.

   then the current player should play the blocking moves.
3. Otherwise, play a random move.

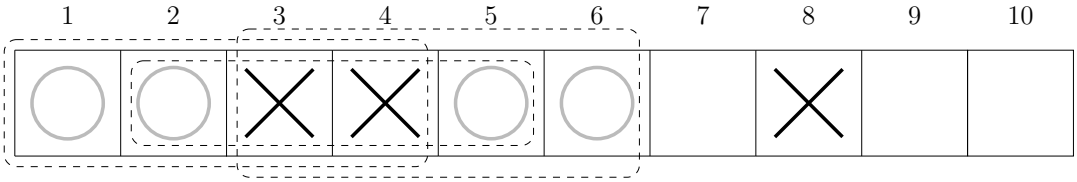
One way to implement this new Simulation policy is to override the default `doSimulation` function to use this policy. However, the steps described would also perform well as part of the tree policy. Since it is never worth considering moves where the opponent can force a win in the next turn, the policy would serve to prune undesirable nodes from the tree. This policy is referred to as the WinSave heuristic throughout the rest of this report.

Immutable Unboxed Arrays

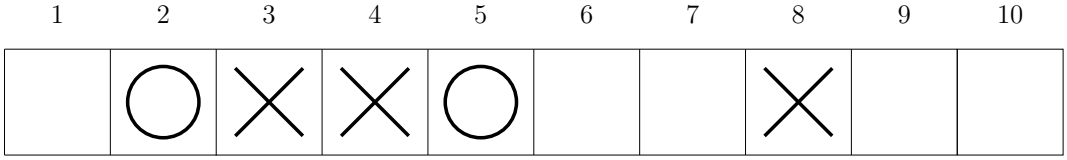
In order to use this modified policy in the tree, it became clear that the `Connectk` data structure needed to be updated from the list of lists representation (figure 3.3) to an immutable unboxed array representation. Not only did this improve speed performance (see *Fast* agent in Evaluation chapter), but also allowed for a cleaner transition from tree policy to Simulation policy. When using the list of lists representation in combination with mutable unboxed arrays a custom conversion function was used. When using mutable and immutable unboxed arrays, the built-in `freeze` and `thaw` functions were used. This new data structure for Connect games based on immutable arrays is called `Connectk'`.



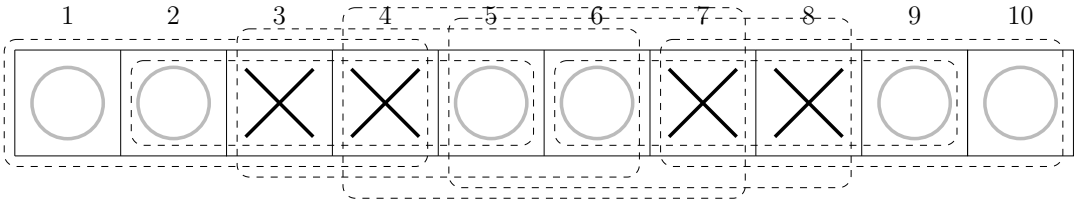
(a) X unable to win in the next turn. O to play, 2 moves left.



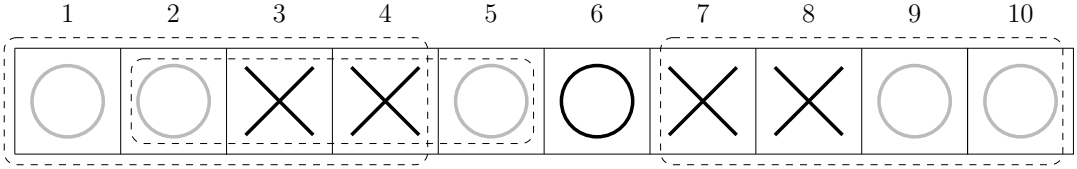
(b) X with 3 potential ways she can win in the next turn. O to play, 2 moves left.



(c) Figure 3.6b after O has played his 2 moves which block all 3 of X's potential wins.



(d) X with 7 potential ways she can win in the next turn. O to play, 2 moves left.



(e) Figure 3.6d after O has chosen move 6. X has 3 potential ways she can win in the next turn. O to play, 1 move left.

Figure 3.6: Example of WinSave heuristic for $k = 4$, $p = 2$.

Connectk'

The data structure is essentially the same as `Connectk` except that the board is represented as an immutable array and that it carries two lists, `noteToSelf` and `noteToNext`. `noteToSelf` is a list of move chains which will save the current player from loosing when it becomes the opponents turn. If `noteToSelf` is empty, the current player cannot save himself and has effectively already lost the game. If `noteToSelf` contains a single, empty move chain, he is safe for at least the next turn and may safely play randomly. `noteToNext` is a list of move chains allows the current player to win when it next becomes his turn. At the end of a players turn, this list is used to generate the `noteToSelf` list which the opponent will use for his first move. In a way, this can be seen as the players cooperating, telling each other how to block each other's attacks.

The `noteToNext` list is populated through a generalization of figure 3.5. When a new piece is added, instead of searching for k-in-a-row in all of the lines specified, the search aims to find all k-in-a-row-with-p-gaps (figure 3.6). This represents all the ways the current player can win when it next becomes his turn, the definition of `noteToNext`. Similar reasoning to the informal proof in section 3.2.2 can be used to show that it is not possible to get into a situation where there are untracked k-in-a-row-with-p-gaps on the board since they are either always blocked or the game reaches a terminal state.

Some examples

Figure 3.6a represents a transition from X's turn to O's turn. At the end of X's turn she has `noteToNext=[]`, that is, there is no way she can win on her next turn. This is converted into O's `noteToSelf=[[]]`, that is, O does not need to block X and may play randomly.

Figure 3.6b represents a transition from X's turn to O's turn. At the end of X's turn she has `noteToNext=[[1,2],[2,5],[5,6]]`, these represent the move chains which she can use to win on her next turn. This is converted into O's `noteToSelf=[[1,5],[2,5],[2,6]]`, these represent the move chains which which O can use to block X. In figure 3.6c he has chosen `[2,5]`, but any of the others would have been equally valid.

Figure 3.6d represents a transition from X's turn to O's turn. At the end of X's turn she has `noteToNext=[[1,2],[2,5],[5,6],[6,9],[9,10]]`, these represent the move chains which she can use to win on her next turn. This is converted into O's `noteToSelf=[]`, this effectively says O is unable to block X and has already lost. Figure 3.6e does not represent a legal position in `Connectk'` since, by this policy, O had already lost the game before moving to 6.

List Conversion

Fragment 3.8 shows some of the functions which were required to perform the switching behaviour to convert between the `noteToNext` to `noteToSelf` lists as it changes from one players turn to another. Since the functions are general and could be helpful to library users they were added to the public export list of `MCTS.Game`. `choose'` generates all the ways of selecting one element from every list in a list of lists. It is detailed in fragment 2.3. `(~!!~)` selects one element of a list by index, and returns that element and a list of all the other elements. It is moved here from being a private function in the `MCTS.Core` module as it became apparent it was useful for more general purposes. `settify` turns a list into a set by deleting duplicate elements. It is interesting because it was the cause of very confusing bug in this agent. Although `prop_settify` specified the function correctly, QuickCheck never generated any arbitrary lists with duplicate elements and `prop_settify` always passed. It was discovered that for a significant amount of time, `settify` would only delete one instance of a duplicate element, rather than all-but-one of the duplicate elements.

Fragment 3.8 Functions added to `MCTS.Game` public interface.

```
1 module MCTS.Game where
2
3 --... previous definition ...--
4
5 choose' :: [[a]] → [[a]]
6 (~!!~) :: [a] → Int→ (a,[a])
7 settify :: Eq a ⇒ [a]→[a]
```

`Connectk'` as an Instance of `Game`

The only way this differs from the `Connectk` instance of `Game` is through the definition of the `currentState` and `legalMoves` functions:

`currentState g` =

$$\begin{cases} \texttt{Win p} & \text{if the player who isn't p's turn in } [\![\texttt{g}]\!] \text{ and } \texttt{noteToSelf g} = \texttt{[]} \\ \texttt{Stale} & \text{if } \texttt{emptySquares g} = \texttt{[]} \\ \texttt{InProgress} & \text{otherwise} \end{cases}$$

`legalMoves g` =

$$\begin{cases} \texttt{[]} & \text{if } \texttt{currentState g} = \texttt{Win \_} \\ \texttt{emptySquares g} & \texttt{noteToSelf g} = \texttt{[[]]} \\ \texttt{map head \$ noteToSelf g} & \text{otherwise} \end{cases}$$

Chapter 4

Evaluation

Evaluation of the project is split into two parts: the suitability of the MCTS library for purpose; and the effectiveness of the Connect agent. The first part briefly explains why the library meets the requirements. The second part shows that the agent plays well given that enough MCTS iterations are performed.

4.1 MCTS Library

Section 2.4.2 shows that the design meets the requirements set out in Section 2.1. Since the finished library follows the design closely, the library also meets these requirements.

The library was developed in a test-driven fashion which gives a reasonable level of assurance that the library functions properly. In addition, experiments involving multi-player variations of Connect, and bespoke Selection, Simulation and Backpropagation phases showed that the internal tree structure behaved as expected and the variations on the agents play rationally. The following section gives a more detailed insight into the performance which was extracted from the MCTS library by the Connect agent.

4.2 Connect Agent

The test bed used is the open source program *Connectk* [12]. It is designed to test various Connect agents. Several good agents are provided with the software. It also provides the user with the ability to write their own agents in the C programming language. A dummy agent which uses named pipes to communicate with the MCTS agent was written.

Evaluation of the MCTS agent is performed by tournaments[1] against other agents. Care must be taken when running tournaments as two agents can sometimes repeatedly play the same game against each other. When this occurs, the

[1] A 'tournament' is the simulation of many games between two agents where the results of games are stored.

tournament results are unreliable. Games which are rotated or translated versions must also be considered 'repeated', these scenarios are harder to identify. One method of identification is to check whether games repeatedly finish after x moves.

Variables to be considered are...

1. The various opponent agents available.
2. The various versions of the agent.
3. The values for the parameters of Connect (n, m, k, p, q).
4. The length of time to allow each agent to move.

The opponent agents considered are...

1. Minimax search with *Threats* utility function. Threats favours moves which build multiple sequences, or build a sequence and block an opponent sequence.
2. Minimax search with *Sequences* utility function - Sequences gives disproportionate utility to shorter sequences. In the hope that many shorter sequences will lever the agent into advantageous situations later in the game.

The Sequences agent is considered the strongest by the writers of Connectk [12]. However, qualitative analysis of games played by this agent showed that although it plays well during the mid-game, there are several distinct pathological move sequences in the early-game which always beat Sequences.

Tournaments between Sequences and Threats were uninteresting since the agents are deterministic and every game is the same. Small changes were made to the search parameters of the agents for each game. The changes made were identical in both agents in an attempt to preserve fairness. As a result, each game was different. Threats was established as the best agent over a large range of game parameters. Indeed, Threats combined with modest search parameters provides a formidable opponent to casual human players. Threats is used as the opponent for the MCTS agent in all tournaments.

'Modest search parameters', refers to searching 9 moves ahead with a variable branching factor in the range 1-13. The branching factor varies for each position based on the level of certainty with which Threats evaluates each move from that position. High certainty leads to a low branching factor, and vice versa. The exact thinking-time taken is dependant upon board position, with an average of about 10 seconds.

The versions of the MCTS agent to be evaluated are ...

Slow - List representation of boards & default Simulation.

Medium - List representation of boards & mutable arrays used in Simulation.

Fast - Immutable array representation of boards in game tree & mutable arrays used in Simulation.

Clever - Immutable array representation of boards in game tree & mutable arrays used in Simulation. Tree policy and Simulation policy uses WinSave heuristic.

Lazy - Simulations which do not complete in 10 moves are terminated early. For early terminations, `Stale` is Backpropagated up the tree. Otherwise, this agent behaves the same as Clever.

These discrete agents represent the evolution of the agent well. Using these, it should be possible to show several things: to what extent the individual efficiency improvements were successful; to what extent the WinSave heuristic affects both tournament performance and execution time; how useful in terms of tournament performance it is to simulate games all the way to termination; and how much speed performance can be improved by terminating Simulations early.

Discussion of game parameters

As detailed in Section 1.2, Connect (n,m,k,p,q) is played on an $n \times m$ board where two players aim to connect k stones in a row. On Black's first turn she places q stones. On subsequent turns, p stones are placed. Care must be taken when choosing the values for these parameters since many choices lead to very unfair or uninteresting games. Connect6 = Connect (19,19,6,2,1) and GoMoku = Connect (15,15,5,1,1) are real games with carefully chosen parameters. Only Connect (n,n,5,1,1) and Connect (n,n,6,2,1) games will be considered. In the interest of performing as many tournaments as possible, a small value of n should be chosen. It is not, however, advantageous to consider very small boards. Connect (5,5,5,1,1) and Connect (6,6,6,2,1) for example are very easy draws for both players. Experimenting with self-play of Threats reveal that Connect (12,12,5,1,1) and Connect (13,13,6,1,1) are the smallest interesting boards. They are the first cases where Threats with a modest minimax search is able to consistently beat the Threats utility function alone.

Choice of Thinking Time

Gomocup, a AI tournament for GoMoku allows up to 300 seconds thinking-time per move. Such long thinking-times lead to very long tournament times. A 100-game tournament with an average of 30 moves per game would take 250 hours. It

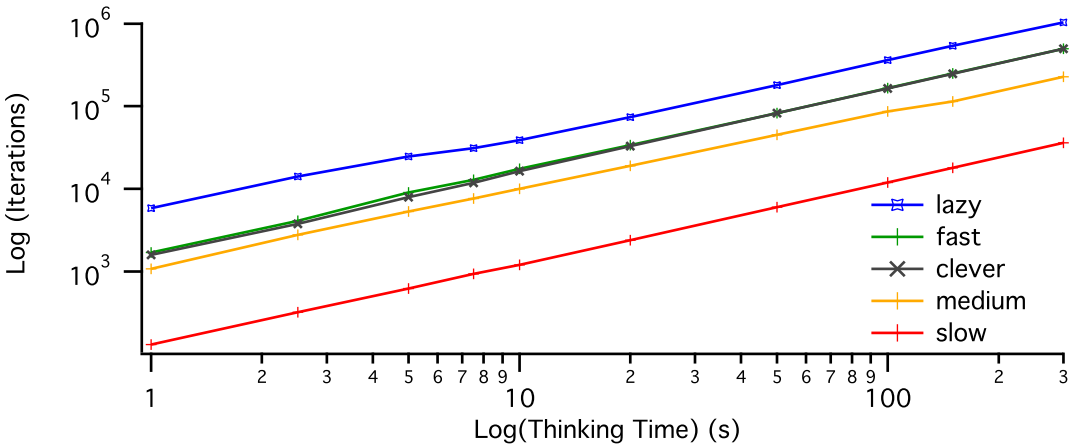


Figure 4.1: Number of MCTS iterations performed to decide first move when starting from an empty 12×12 board. Plotted against thinking time.

is important to run large tournaments of around 100 games to ensure reliability in data acquisition. Ideally, even longer tournaments would be performed, but a compromise must be made due to time constraints. After the strongest version of the MCTS agent has been identified, two tournaments with the full 300 seconds thinking-time will be played. One will play Freestyle GoMoku; the other, Connect6. Both will last 100 games, using Threats with a modest minimax search as the opponent.

4.2.1 Runtime performance

In this section, the performance of various boards is tested based on the number of iterations performed on an empty board. It is difficult to consider iterations performed in the mid-game, as the number of iterations varies considerably based on the board position. As figure 4.1 shows, the number of iterations performed on the first move is directly proportional to the thinking time allowed. This holds for all agents on 12×12 boards, it is assumed that it holds for other board sizes. Therefore, the number of iterations performed on the first move can be used interchangeably with thinking-time.

Analysis

Figure 4.1 shows that the number of iterations performed on the first move is directly proportional to the thinking time allocated for the first move. This is unsurpring for the Slow, Medium, Fast, and Clever agents. For these agents, every iteration represents performing a full Simulation from a start node to termination. It is a surprising result for the Lazy agent. In the Lazy agent, as more Simulations are performed, the tree grows, but only a maximum of 10 nodes are simulated beyond the fringe of the tree. From this, one would expect that the number of iterations performed per unit thinking time would decrease as the tree grows in

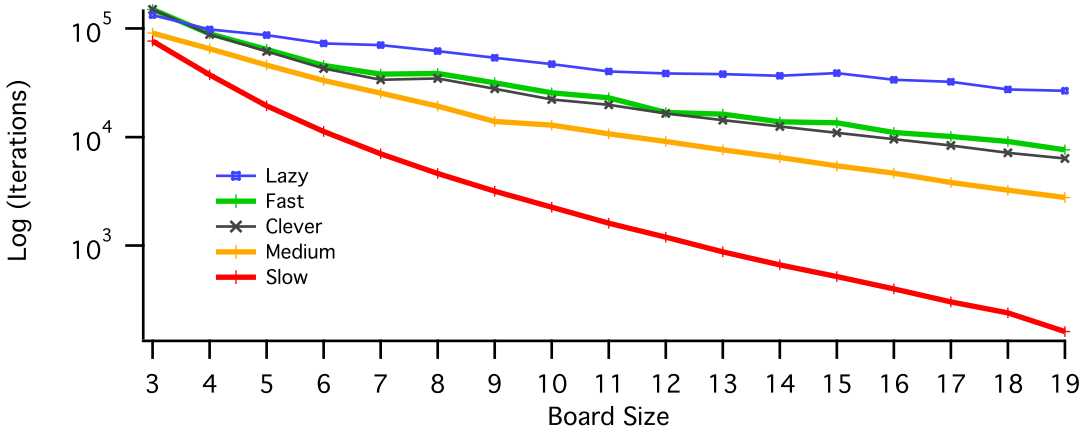


Figure 4.2: Number of MCTS iterations performed in 10 seconds to decide first move when starting from an empty board. Plotted against board size.

size. Figure 4.1 shows that this effect does not occur, at least for trees containing up to a million nodes. This suggests the Selection and Backpropagation phases are not a significant contribution to overhead, when compared to the Simulation and Expansion phases. Profiling data generated by the Lazy agent supported these findings.

Figure 4.2 shows that all agents are roughly the same for small board sizes. This is unsuprising since MCTS quickly builds the entire game tree when the state space is small. For example, in the case of 3×3 boards, after about 1000 iterations the full game tree is built, and the rest of the work performed by the agent is repeated Selection. The Selection phase is implemented in the same way for all of the agents.

The number of iterations performed decreases dramatically for all agents as the board size increases. This is to be expected since increasing board size increases both the branching factor at each move and the average depth at which Simulations terminate. The Lazy agent is affected least by the board size since the average depth at which Simulations terminate is always 10, regardless of board size.

The points in Figure 4.2 for the Lazy, Fast, Cleverand Medium agents look unreliable since they don't appear to fit a curve well. Repeated trials showed that the standard deviations for all points were at most 0.1% of their corresponding mean values. The slightly erratic behaviour may be due to the idiosyncrasies of different board-sizes. In Figure 4.2, on a board size of 12 the Clever and Fast agents perform almost the exact same number of iterations as each other when given 10 seconds thinking time. Figure 4.1 shows that this observation holds across a range of thinking times for 12×12 boards.

In general, the Clever agent performs almost as well as the Fast agent. This strongly suggests that the WinSave heuristic contributes very little to the overall overhead of the search.

4.2.2 Connect(12,12,5,1,1) Tournament Performance

Figure 4.3 shows that, when the Fast, Medium and Slow agents are able to perform the same number of iterations on their first move, the three agents perform similarly in tournaments. This is consistent with the fact that Fast, Medium and Slow perform MCTS in exactly the same way, the only difference being their efficiency (see informal proof in Section 3.2.2). The Slow and Medium agents are not plotted for the full x axis due to Simulation time constraints. For example, a single 100 game tournament for the Slow agent for an x value of 40,000, would take approximately 33 hours.

Figure 4.3 also shows three approximate phases of tournament performance: for 0-15 thousand iterations, most games are lost; for 15-27 thousand iterations, most games are drawn; and for more than 27 thousand iterations, most games are won. Qualitive analysis of games played show two sub phases in the 0-15 thousand range: for 0-8 thousand iterations, most moves seem irrational and essentially random; for 8-15 thousand iterations, most moves are sensible but the agent is outplayed by the stronger Threats agent.

Figure 4.4 shows that, for any given thinking time, the most successful agents are Lazy, Clever and Fast in that order. Slow and Medium are not considered as they perform MCTS in the same fashion as Fast, but at a slower speed. The fact that Clever performs better than Fast is to be expected; the WinSave heuristic has runtime performance almost as good as Fast, but has a more rational tree and Simulation policy. The fact that Lazy performs better than Clever is interesting: for 12×12 boards, the boost in speed performance from terminating Simulations early outweighs the loss in tournament performance per iteration. Figure 4.2 suggests that as boards get larger, the advantages of using Lazy over Clever increase.

4.2.3 Connect(13,13,6,2,1) Tournament Performance

No agents are able to play Connect(13,13,6,2,1) well against the Threats utility function, even when given up to 300 seconds thinking-time for each turn. The Clever agent is able to perform reasonably well against a random agent. However qualitive analysis of these games show that the Clever agent will play randomly until a win that turn is possible, or it is forced to block that turn. These results could be due to the high branching factor of Connect(13,13,6,2,1), $\binom{13\times13-1}{2} = 20856$. This is about 145 times higher than the branching factor for Connect(12,12,5,1,1). The Lazy agent, which performs one million iterations in 300 seconds, would only be able to dedicate an average of 50 iterations to each of the possible moves within this time. This problem was foreseen during development, and as such, each ply of the game tree does not represent a single turn, but a single move. This reduces the branching factor in the tree to the 'move' branching factor, 169. This result shows that this was not sufficient to solve the problem.

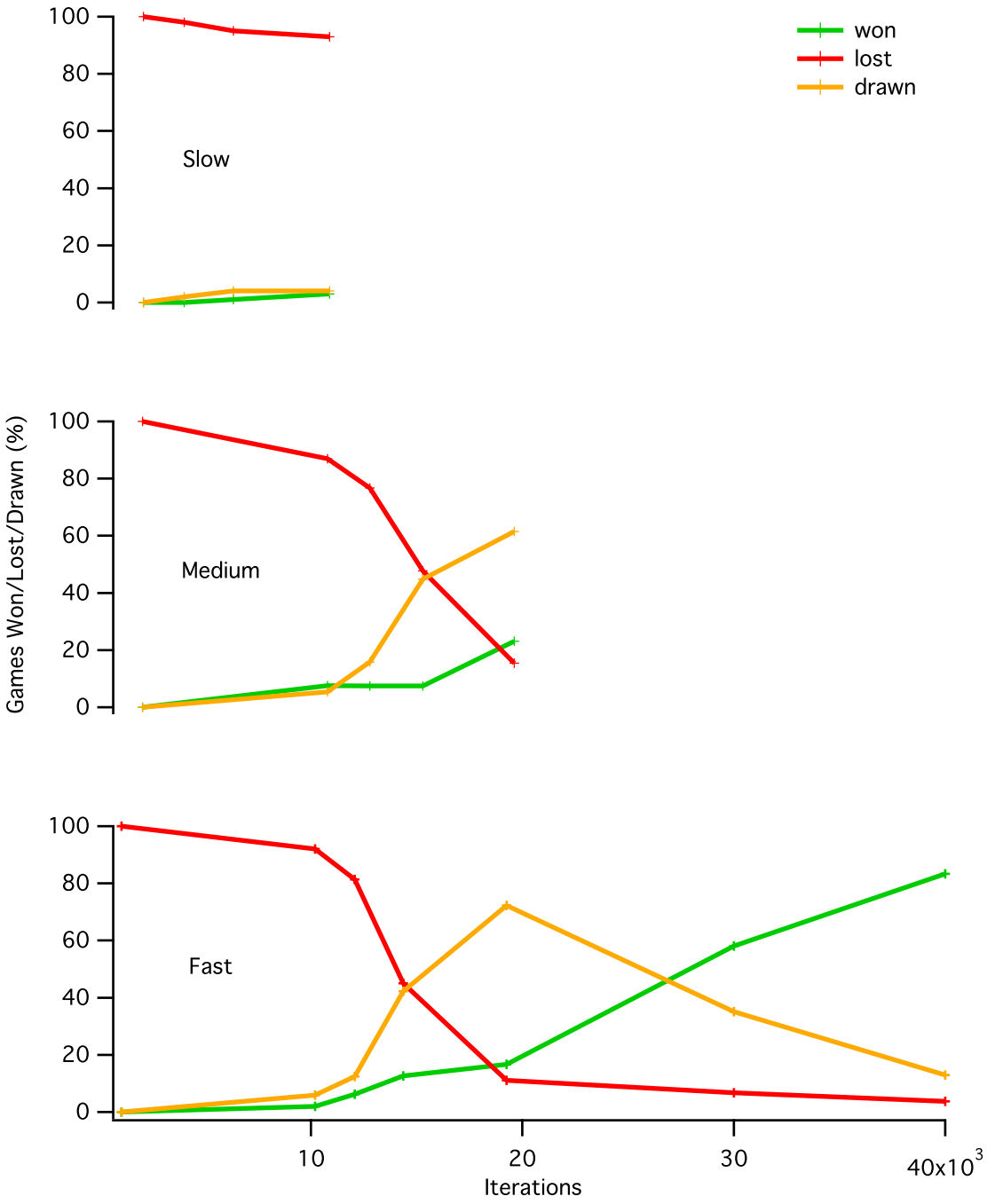


Figure 4.3: Tournament results for the Slow, Medium, and Fast agents. Each plays against the Threats utility function with modest minimax search. The tournament results are plotted against the average number of iterations performed for the first move in each game.

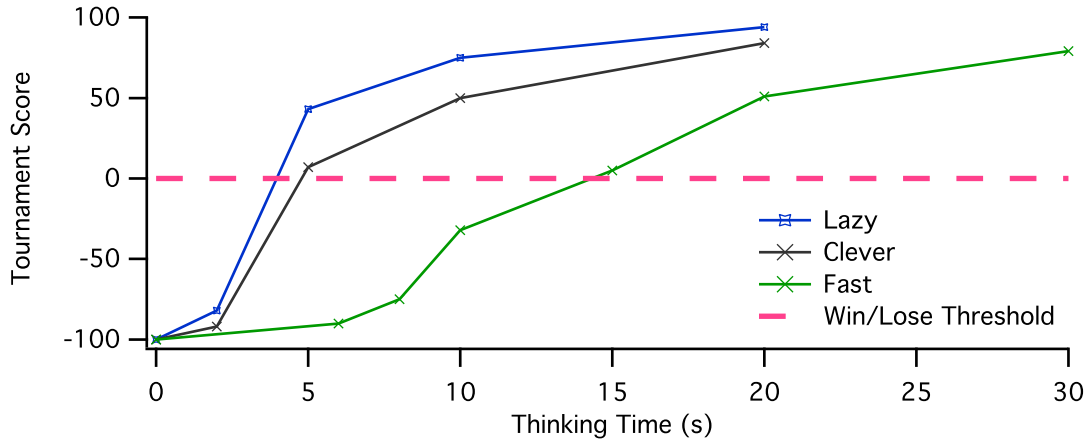


Figure 4.4: The tournament results plotted against thinking time.

4.2.4 Freestyle GoMoku and Connect6 Tournaments

The Lazy agent was chosen for the Freestyle GoMoku tournament since it outplays Clever for 12×12 boards. The number of iterations performed by Lazy on 15×15 boards is a slight increase on 12×12 boards. In contrast, the number of iterations performed by Clever from 12×12 to 15×15 boards drops by 40%.

The tournament was played against the Threats utility function with modest search parameters. The Lazy agent won the tournament, winning 31% of the games. The narrow margin of this win highlights the scalability issues for the MCTS agent. While the tournament performance of Threats with minimax search is fairly constant as board size increases, the performance of the MCTS agent deteriorates rapidly. Not only can fewer iterations be performed on larger boards, but more iterations are required to play well. It was not considered worthwhile running a Connect6 tournament.

Chapter 5

Conclusions

In this project, an MCTS library and Connect agent were successfully implemented in Haskell.

The library...

Can be used to implement a large range of games, from single-player puzzles to n-player, imperfect information, non-zero-sum, simultaneous play games.

Allows bespoke customization of the phases of MCTS described in the report while providing sensible defaults.

Supports the implementation of many of the modifications to MCTS proposed in current research.

Is user-friendly and fully documented (see Appendix B).

The Connect agent...

Can play any game in the Connect family.

Plays in a tactical fashion, akin to humans and other computer players, rather than strategically.

Beat a minimax-based agent in a 100-game Freestyle GoMoku tournament.

Is unable to play Connect6 games to a good standard.

The use of Haskell in this project has been a pleasure. Despite the considerable investment in increasing code efficiency, there are significant advantages to using the language:

Functional languages strongly resemble pure mathematics. This often makes the step of converting a mathematical definition of a function into an implementation in code straight-forward.

Functional languages are well-suited to test-driven development. It is a clean practice which allows function specifications to be embedded into the code.

Haskell supports features such as ad-hoc polymorphism and monadic programming. This makes it better suited to software engineering projects than other functional languages.

Very little time was required to debug code; most bugs were caught by the type-checker during compilation. Bugs that slipped the net usually led to failed QuickCheck properties.

The use of MCTS in this project has been interesting. It is fascinating that a search based upon random simulations and limited domain knowledge can perform well for such a diverse range of problems. It also posed a range of challenging problems during implementation.

Connect was a poor choice for an example agent for the MCTS library. The game was chosen for the large state space and high branching factor, properties which are shared with Go. However, while it is difficult to write an effective utility function for Go, this is not the case for what Connect. Although Go has simple rules, the emergent complexity is profound. Gelly & Silver [8] noted that a move made now may not have an effect until 50 or 100 moves later. It was my hope that I might have revealed some of this complexity in Connect, but I suspect that most of the time spent simulating these distant scenarios was wasted. This is supported by the fact that the Lazy agent performed well, even though it did not run Simulations through to completion.

A successful MCTS agent for Connect6 has been constructed [22]. However, it employs a large amount of domain specific knowledge. I wanted this example agent to remain agnostic to the specific human tactics of the game. An attempt to implement an agent for a more complex game, such as Kriegspiel, may have been a more suitable endeavour.

Bibliography

[1] L. V. Alus, H. J. van den Herik, 1, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12(1):7–23, 1996.

[2] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 1 edition, November 2002.

[3] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[4] Tristan Cazenave. Multi-player go. In *Proceedings of the 6th international conference on Computers and Games*, CG '08, pages 50–59, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Guillaume Chaslot, Mark Winands, Jaap H. van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. In *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session*, 2007.

[6] Paolo Ciancarini and Gian Piero Favini. Monte Carlo tree search in kriegspiel. *Artificial Intelligence*, 174(11):670 – 684, 2010.

[7] Remi Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. In *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin / Heidelberg, 2007.

[8] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856 – 1875, 2011.

[9] Sean Holden. Projects for part II students. `http://web.archive.org/web/20100111001125/http://www.cl.cam.ac.uk/~sbh11/projects.html`.

[10] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98, 1999.

[11] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning Machine Learning: ECML 2006. volume 4212 of *Lecture Notes in Computer Science*, chapter 29, pages 282–293. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[12] Michael Levin. Connect-k. `http://web.archive.org/web/20100714155017/http://risujin.org/connectk/`.

[13] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly, 2009.

[14] Simon Peyton Jones. Ghc documentation. `http://www.haskell.org/ghc/docs/latest/html/libraries/index.html`.

[15] S. Samothrakis, D. Robles, and S. Lucas. Fast approximate max-n Monte-Carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[16] M. P. D Schadd. *Selective Search in Games of Different Complexity*. PhD thesis, Maastricht University, Netherlands, 2011.

[17] Maarten P.D. Schadd, Mark H.M. Winands, Mandy J.W. Tak, and Jos W.H.M. Uiterwijk. Single-player Monte-Carlo tree search for samegame. *Knowledge-Based Systems*, 2011.

[18] Ronald David Schwarz. Blackjack :–an application of Monte Carlo simulation to gaming strategies involving risk, 1982.

[19] Gerald Tesauro and Gregory R. Galperin. On-line policy improvement using Monte-Carlo search. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9 (NIPS-96)*, pages 1068–1074. The MIT Press, 1997.

[20] H.Jaap van den Herik, Jos W.H.M. Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(12):277 – 311, 2002.

[21] Fan Xie and Zhiqing Liu. Backpropagation modification in Monte-Carlo game tree search. In *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, volume 2, pages 125 –128, nov. 2009.

[22] Shi-Jim Yen and Jung-Kuei Yang. Two-stage Monte Carlo tree search for connect6. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(2):100–118, 2011.

Appendix A

Profiling Data

The following is raw profiling data generated by *GHC*. The conclusion was made that the `doSimulation` function should be replaced with a more efficient implementation.

```
Fri Mar  2 13:25 2012 Time and Allocation Profiling Report  (Final)

Main +RTS -p -RTS
[[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
 [Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
 [Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
 [Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
 [Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
 [Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]]

total time  =       13.74 secs   (687 ticks @ 20 ms)
total alloc = 6,535,097,376 bytes  (excludes profiling overheads)

COST CENTRE                    MODULE               %time %alloc

doSimulation                   MCTS.Game             34.8   53.7
row                            MCTS.Sample.Connectk  14.8    3.7
doSelection                    MCTS.Config           12.2   14.1
kDiag                          MCTS.Sample.Connectk   9.0    7.9
funs                           MCTS.Sample.Connectk   6.8    3.7
kInARow                        MCTS.Sample.Connectk   5.2    5.9
pick                           MCTS.Game              5.1    1.8
mcts                           MCTS                   3.8    4.8
firstMatch                     MCTS.Sample.Connectk   2.8    0.0
averageScore                   MCTS                   1.5    1.3
topMcts                        MCTS                   1.0    1.8


                                                                      individual    inherited
COST CENTRE             MODULE                                no.     entries  %time %alloc   %time %alloc

MAIN                    MAIN                                    1           0   0.0    0.0   100.0  100.0
 main                   Main                                  568           3   0.0    0.0   100.0  100.0
  cmpBoard              Main                                  654           3   0.0    0.0     0.0    0.0
   cmpRow               Main                                  655           4   0.0    0.0     0.0    0.0
  iterativeMcts         MCTS.Mcts                             574      369964   0.6    0.2    99.6   99.3
   topMcts              MCTS.Mcts                             581       20000   1.0    1.8    99.0   99.1
    doBackpropagation   MCTS.Mcts                             614       10000   0.0    0.0     0.0    0.0
    rChildList          MCTS.Mcts                             596      730000   0.1    0.0     0.1    0.0
    doSelection         MCTS.Mcts                             595      370000   3.6    4.1     4.9    4.8
     averageScore       MCTS.Mcts                             604           0   0.9    0.5     1.0    0.5
      rQ                MCTS.Mcts                             627      359299   0.1    0.0     0.1    0.0
      rPlayed           MCTS.Mcts                             605      359999   0.0    0.0     0.0    0.0
     rPlayed            MCTS.Mcts                             603           0   0.3    0.1     0.3    0.1
    mcts                MCTS.Mcts                             591       33834   3.8    4.8    92.1   91.8
     doBackpropagation  MCTS.Mcts                             634       23521   0.3    0.0     0.3    0.0
      rQ                MCTS.Mcts                             636       23360   0.0    0.0     0.0    0.0
      rPlayed           MCTS.Mcts                             635       23521   0.0    0.0     0.0    0.0
     doSelection        MCTS.Mcts                             630      838466   8.6   10.0     9.9   11.0
      rPlayed           MCTS.Mcts                             633           0   0.4    0.3     0.4    0.3
      averageScore      MCTS.Mcts                             631           0   0.6    0.8     0.9    0.8
       rQ               MCTS.Mcts                             637      511062   0.3    0.0     0.3    0.0
       rPlayed          MCTS.Mcts                             632      814632   0.0    0.0     0.0    0.0
     rChildList         MCTS.Mcts                             629           0   0.1    0.0     0.1    0.0
     simulateNode       MCTS.Mcts                             613       10000   0.0    0.0    73.1   73.4
      doBackpropagation MCTS.Mcts                             625        7577   0.0    0.0     0.0    0.0
       rQ               MCTS.Mcts                             628        7531   0.0    0.0     0.0    0.0
       rPlayed          MCTS.Mcts                             626        7577   0.0    0.0     0.0    0.0
      simulate          MCTS.Mcts                             616      207640  34.8   53.7    73.1   73.4
```

pick	Game.Game	623	197640	5.1	1.8	5.1	1.8
kDiag	Game.Connectk	621	1609188	7.7	6.4	13.2	9.5
funs	Game.Connectk	622	804594	5.5	3.1	5.5	3.1
kInARow	Game.Connectk	618	1214111	4.4	4.9	19.2	8.5
row	Game.Connectk	620	38748590	13.1	3.6	13.1	3.6
firstMatch	Game.Connectk	619	8457967	1.7	0.0	1.7	0.0
firstMatch	Game.Connectk	617	1411816	0.7	0.0	0.7	0.0
rGame	MCTS.Mcts	615	0	0.0	0.0	0.0	0.0
rPlayed	MCTS.Mcts	612	33834	0.1	0.0	0.1	0.0
kDiag	Game.Connectk	610	270672	1.2	1.1	2.2	1.6
funs	Game.Connectk	611	135336	1.0	0.5	1.0	0.5
kInARow	Game.Connectk	594	203004	0.9	0.8	2.6	0.9
row	Game.Connectk	607	6496128	1.5	0.1	1.5	0.1
firstMatch	Game.Connectk	606	1421028	0.3	0.0	0.3	0.0
firstMatch	Game.Connectk	593	236838	0.0	0.0	0.0	0.0
rGame	MCTS.Mcts	592	0	0.0	0.0	0.0	0.0
kDiag	Game.Connectk	587	80000	0.1	0.3	0.4	0.5
funs	Game.Connectk	588	40000	0.3	0.2	0.3	0.2
kInARow	Game.Connectk	584	60000	0.0	0.2	0.3	0.2
row	Game.Connectk	586	1920000	0.3	0.0	0.3	0.0
firstMatch	Game.Connectk	585	420000	0.0	0.0	0.0	0.0
firstMatch	Game.Connectk	583	70000	0.0	0.0	0.0	0.0
rGame	MCTS.Mcts	582	0	0.0	0.0	0.0	0.0
rChildList	MCTS.Mcts	575	20000	0.0	0.0	0.0	0.0
expand	MCTS.Mcts	573	90602	0.4	0.6	0.4	0.6
rGame	MCTS.Mcts	576	1	0.0	0.0	0.0	0.0
gameToLeaf	MCTS.Mcts	572	1	0.0	0.0	0.0	0.0
playerList	Game.Connectk	571	1	0.0	0.0	0.0	0.0
selectBestMove	MCTS.Mcts	569	1	0.0	0.0	0.0	0.0
rGame	MCTS.Mcts	649	1	0.0	0.0	0.0	0.0
averageScore	MCTS.Mcts	641	0	0.0	0.0	0.0	0.0
rQ	MCTS.Mcts	643	70	0.0	0.0	0.0	0.0
rPlayed	MCTS.Mcts	642	70	0.0	0.0	0.0	0.0
rChildList	MCTS.Mcts	570	2	0.0	0.0	0.0	0.0
CAF:lvl14_ro80	Main	562	1	0.0	0.0	0.0	0.0
CAF:lvl13_ro7Y	Main	561	1	0.0	0.0	0.0	0.0
main	Main	578	0	0.0	0.0	0.0	0.0
CAF:lvl12_ro7W	Main	560	1	0.0	0.0	0.0	0.0
main	Main	579	0	0.0	0.0	0.0	0.0
CAF:lvl11_ro7U	Main	559	1	0.0	0.0	0.0	0.0
main	Main	580	0	0.0	0.0	0.0	0.0
CAF:lvl10_ro7S	Main	558	1	0.0	0.0	0.0	0.0
CAF:lvl9_ro7Q	Main	557	1	0.0	0.0	0.0	0.0
main	Main	651	0	0.0	0.0	0.0	0.0
CAF:lvl8_ro7O	Main	556	1	0.0	0.0	0.0	0.0
main	Main	652	0	0.0	0.0	0.0	0.0
CAF:lvl7_ro7M	Main	555	1	0.0	0.0	0.0	0.0
main	Main	653	0	0.0	0.0	0.0	0.0
CAF:$dEq_ro7I	Main	554	1	0.0	0.0	0.0	0.0
CAF	GHC.Read	532	2	0.0	0.0	0.0	0.0
CAF	GHC.Float	531	1	0.0	0.0	0.0	0.0
CAF	Text.Read.Lex	520	5	0.0	0.0	0.0	0.0
CAF	GHC.Int	516	2	0.0	0.0	0.0	0.0
CAF	GHC.IO.Handle.FD	490	2	0.0	0.0	0.0	0.0
CAF	GHC.IO.Encoding.Iconv	448	2	0.0	0.0	0.0	0.0
CAF	GHC.Conc.Signal	445	1	0.0	0.0	0.0	0.0
CAF:lvl42_r9vn	MCTS.Mcts	437	1	0.0	0.0	0.0	0.0
selectBestMove	MCTS.Mcts	638	0	0.0	0.0	0.0	0.0
averageScore	MCTS.Mcts	639	1	0.0	0.0	0.0	0.0
rQ	MCTS.Mcts	648	1	0.0	0.0	0.0	0.0
rPlayed	MCTS.Mcts	640	1	0.0	0.0	0.0	0.0
CAF:lvl41_r9vl	MCTS.Mcts	436	1	0.0	0.0	0.0	0.0
selectBestMove	MCTS.Mcts	644	0	0.0	0.0	0.0	0.0
averageScore	MCTS.Mcts	645	1	0.0	0.0	0.0	0.0
rQ	MCTS.Mcts	647	1	0.0	0.0	0.0	0.0
rPlayed	MCTS.Mcts	646	1	0.0	0.0	0.0	0.0
CAF:lvl40_r9ve	MCTS.Mcts	435	1	0.0	0.0	0.0	0.0
doSelection	MCTS.Mcts	600	0	0.0	0.0	0.0	0.0
averageScore	MCTS.Mcts	601	1	0.0	0.0	0.0	0.0
rPlayed	MCTS.Mcts	602	1	0.0	0.0	0.0	0.0
CAF:$dNum_r9uU	MCTS.Mcts	431	1	0.0	0.0	0.0	0.0
CAF:lvl35_r9uk	MCTS.Mcts	428	1	0.0	0.0	0.0	0.0
CAF:lvl34_r9ui	MCTS.Mcts	427	1	0.0	0.0	0.0	0.0
CAF	System.Random	393	1	0.0	0.0	0.0	0.0
CAF	Data.Fixed	388	3	0.0	0.0	0.0	0.0
CAF	Data.Time.Clock.POSIX	385	2	0.0	0.0	0.0	0.0
CAF:lvl6_rkVC	Game.Connectk	301	1	0.0	0.0	0.0	0.0
row	Game.Connectk	624	0	0.0	0.0	0.0	0.0
CAF:lvl5_rkVA	Game.Connectk	300	1	0.0	0.0	0.0	0.0
row	Game.Connectk	608	0	0.0	0.0	0.0	0.0
CAF:k	Game.Connectk	299	1	0.0	0.0	0.0	0.0
k	Game.Connectk	609	1	0.0	0.0	0.0	0.0
CAF:funs2	Game.Connectk	297	1	0.0	0.0	0.0	0.0
funs	Game.Connectk	589	0	0.0	0.0	0.0	0.0
CAF:lvl4_rkVu	Game.Connectk	296	1	0.0	0.0	0.0	0.0
funs	Game.Connectk	590	0	0.0	0.0	0.0	0.0

Appendix B

Documentation

The following pages contain the user documentation for the MCTS library. It was produced using *Haddock*, a Haskell documentation tool.

MCTS.Config

Documentation

defaultConfig :: MctsConfig

This is the default configuration object. Use this if you want to use *UCB* with *c*=1 for *Selection* and standard *Expansion*, *Simulation* and *Backpropagation*. To make changes to the *c* used for *UCB* you should use the following:

```
defaultConfig{'doSelection'='doUcb' c}
```

To make other changes you probably want to import `MCTS.Config` and do a similar trick with the helper functions there.

doUcb :: (Game a, RandomGen g) => Double -> [(ScoreTuple (Player a), Plays)] -> Player a -> g -> (Int, g)

This does *UCB* with a specific value of *c*.

data **MctsConfig**

Dictionary type containing configuration functions and constants for *Expansion*, *Selection* and *Backpropagation* phases of *MCTS*

Constructors

MctsConfig

expandConst :: Int

This constant specifies how many nodes to add to the tree on each simulation. If you can spare the increased memory usage, it may be advantagous to increase this number. It is not possible to only add nodes to the tree which have been simulated at least *n* times due to the way the search is implemented. It might be desirable to do this to save memory usage

doSelection :: (Game a, RandomGen g) => [(ScoreTuple (Player a), Plays)] -> Player a -> g -> (Int, g)

Must select randomly among equal options

doBackpropagation :: Game a => GameState (Player a) -> (ScoreTuple (Player a), Plays) -> (ScoreTuple (Player a), Plays)

Not recursive

pick' :: RandomGen g => g -> [a] -> ((a, [a]), g)

(~!!~) :: [a] -> Int -> (a, [a])

```
update :: (p -> Bool) -> (Score -> Score) -> (Score -> Score) -> ScoreTuple p ->
ScoreTuple p
```

```
readTuple :: Eq p => p -> ScoreTuple p -> Score
```

```
type ScoreTuple a = [(a, Score)]
```

```
type Score = Double
```

```
type Plays = Int
```

MCTS.Core

Documentation

doIterativeMcts :: (Game a, RandomGen g) => a -> MctsConfig -> Int -> g -> (a, g)

Do a number of iterations of *MCTS* and return the most desirable game position. For example:

```
let (myGame',myRandomSeed') = doIterativeMcts myGame defaultConfig myRandomSee
```

where n is the number of iterations to do.

doTimedMcts :: Game a => a -> MctsConfig -> StdGen -> Int -> IO (a, StdGen)

Do as many possible iterations of *MCTS* possible in the availiable time and return the most desirable game position wrapped up in IO monad. Most common use would be directly in the main method, for example:

```
main = do
          ...
          (myGame',myRandomSeed') <- doTimedMcts myGame defaultConfig myRandomS
          ...
```

where n is the number of milliseconds to search for. Note, the search will take a few milliseconds longer than this value, so you will have to play with this value if you are under strict time constraints.

mcts :: (Game a, RandomGen g) => MCT a -> MctsConfig -> g -> (MCT a, GameState (Player a), g)

expand :: Game a => MCT a -> MCT a

gameToLeaf :: Game a => a -> MCT a

MCTS.Game

Documentation

```
class (Eq a, Eq (Player a)) => Game a where
```

Associated Types

```
data Player a :: *
```

You must define your own player type. For a simple two player game you might use

```
data 'Player' a = Black | White | Dud
```

where Dud is a commonly used trick to help in implementation. For example, in chess, your `readPlayerFromBoard` function might return Dud if an out of range position is requested. This allows a simple way to prevent pieces moving off the board without needing to do messy bounds checking everywhere.

Methods

```
legalChildren :: a -> [a]
```

This function is your implementation of the game rules It should specify all of the legal moves which can be made from any legal position. Behavior for illegal positions may be undefined.

```
currentState :: a -> GameState (Player a)
```

This should not result in a representation of the current game position but the state in the sense of weather or not the game has terminated, and if it has, in what state. See `GameState` for more details.

```
allPlayers :: [Player a]
```

This is the function which always returns the list of players which will be represented in the game tree. Following from the above example you would use

```
allPlayers _ = [Black, White]
```

since you don't want to maintain a score for Dud in the game tree.

```
currentPlayer :: a -> Player a
```

This should return the player whose move it is. If you are implementing a simultanious play game, then you should still consider one player to move at a time, but treat the game as having imperfect information.

```
doSimulation :: RandomGen g => a -> g -> (GameState (Player a), g)
```

A default implementation is provided for this function. Don't override it unless you want to modify the *Simulation* phase of *MCTS* You may want to override it if you can provide a more

start of simulation or want to implement a different simulation policy to uniform random.

Instances

Game TicTacToe

---

data **GameState** a

This is not your representation of the current game position, it is what the current position means as far as the search is concerned. The values are fairly self explanatory.

Constructors

InProgress

Stale

Win a

Instances

Eq a => Eq (GameState a)

Show a => Show (GameState a)

---

pick :: RandomGen g => g -> [a] -> (a, g)

Appendix C

Project Proposal

The following pages contain the proposal for this project.

William Kenyon
Emmanuel College
wk249

Part II Computer Science Project Proposal

A Monte Carlo Tree Search Library in Haskell

October 20, 2011

Introduction and Description of the Work

Monte Carlo Tree Search (MCTS) has been used in recent papers to produce good agents for two player perfect information games like Go as an alternative to using complex expert knowledge and pattern recognition. I would like to build a library which implements the MCTS in Haskell allowing people to use it as the core of agents for other games. I would then like to try out my library by writing an agent for one or more sample games.

Starting Point

The Foundations of Functional Programming course taught in part IA will help me think about problems from a functional point of view. I have also spent some time in the summer reading up on the differences[1] between ML and Haskell.

Substance and Structure of the Project

A good description of how Monte Carlo Tree Search works can be found in [3]. I'd like to keep the tree search as general and versatile as possible so that many different games & modifications on the search can be implemented. I shall also provide good defaults (e.g. use UTC as described in [3] as default but still allow the user to write their own selection function) so that you can very quickly code an implementation for a game and have the agent work (even though some games will have terrible performance without modifying things a bit).

In parallel with building the library, I will build a tic-tac-toe[2] agent which will use the search. I will also use slight variants of tic-tac-toe (e.g. blind tic-tac-toe[3]) to ensure that the library is versatile enough to support different kinds of games.

Libraries Used

I intend to build the library from scratch using only general purpose Haskell libraries such as `System.Random` (generating random numbers) and `Data.List` (list manipulation tools).

[1] Simple differences in syntax and semantics, but I have yet to learn about some of the more complex language features of Haskell.

[2] I'm aware that tic-tac-toe is trivial, but I just want a very simple game to use while building the library. It is all that I need to show my library works and is simple enough that I can check to see the algorithm produced really is doing a MCTS. I will consider more complex games as extension problems.

[3] A game of my invention. Neither player knows the others moves. If a player plays in an already occupied square, the move is rejected and the player must try again.

I considered using `Data.Tree.Game_tree`[4] as a data structure to use for algorithm but it seems to be too $\alpha\beta$ search oriented. Also, since it isn't part of an official Haskell library I don't know how trustworthy it is.

Success Criteria

These criteria are based on what sort of games MCTS has been applied to in recent research. I will test each of the following criteria by implementing an agent for a suitable variant of tic-tac-toe and testing it against a random agent (and perfect play agent when that makes sense in the context) with the intention of analysing the trace of the algorithm and checking that it behaves correctly.

Must be possible to use the library to write agents for the following categories of games:

1. Two player, perfect information games.

   MCTS is used in [3] to play Go.

2. Games with imperfect information.

   MCTS has been used in an agent for Kriegspiel ([1]), where each player does not explicitly know anything about the opponents piece layout and the layout must be inferred.

3. Games with more than two players.

   In [4] (an agent for Ms. Pacman using MCTS) the ghosts are represented as other players meaning this is represented as a six player game.

4. Games with simultaneous play.

   A general game player ([2]) uses MCTS to implement agents for simultaneous games. Ms. Pacman, from above, is also an example of where a simultaneous play game.

Researchers have also made modifications to the search that I would like to remain possible. I am thinking from the point of view of 'I want to make change x to the standard MCTS algorithm. Will I be able to do this if I use your library?'.

1. Must allow user to write their own selection function.

   [6] for example, introduces a particular selection function which proves effective for playing the game Lines of Action.

[4] http://hackage.haskell.org/package/game-tree

2. Must support terminating simulations.

   This involves stopping the simulation phase of the algorithm before it has completed and using an evaluation function to evaluate the game state.

   Several ([7, 4]) have found this useful for various reasons. For example, sometimes it is obvious which player has won the game by evaluation function and further simulation would be a waste of time.

Extensions

Implement an AI for k in a row, $n \times n$ tic-tac-toe. Play against connectk[5] for various k and n to test how well a naive agent scales. Connectk seems to be a GUI only agent, but it does support adding custom agents written in C so I would write a dummy C agent which would sit in my test harness whilst also being plugged in to the connectk gui.

I am considering writing a chess agent. However, I would expect very poor performance, even against a random agent, as nobody so far seems to have had very much luck using Monte Carlo methods to play chess. If I were to implement an agent I would test it against against the GNU Chess engine on various difficulty settings and a random agent.

Implement checkers and play it against a random agent, checkersland[6] and the online version of the 1994 world champion chinook[7]. I would expect checkers to perform better than chess would. It may seem pointless to write a checkers AI when the game has been solved ([5]) but I think it would be interesting to try and gather some intuition as to why different games are suited to the search and others aren't.

Finally, I would attempt to modify the agents produced so far in an attempt to get them to perform better.

Timetable and Milestones

In some of the work packages I intend to write up the work package to a fairly complete standard in parallel with whatever the current work package is. This is so that I can fill in details while it is still fairly fresh in my mind. Of course I will still keep notes of what I did as I do it a notebook, but I would like to fill that out in dissertation style sooner rather than later.

[5] http://risujin.org/connectk/
[6] http://www.checkersland.com/download/pc.jsp
[7] http://webdocs.cs.ualberta.ca/~chinook/play/player

21st October - 4th November

Intensity: Medium - Initial research phase.

As the worked examples I have done in Haskell have been very basic toy examples, I've not needed some of the more advanced features of the language. As the project becomes more complex it may be beneficial to grasp these. I know that IO and random number generation requires use of Monads and I'd rather understand them properly rather than copy code fragments from textbooks or the internet.

It may be useful in this time to examine the code structure of large projects which have been implemented in haskell, such as darcs and xmonad, which may give me pointers on how to structure a large haskell project. I also would like to look into test utilities such as HUnit to see what would be most suitable for my needs.

Milestones:

1. Decide on a source version tool.
2. Know advanced features of the language Haskell (and have freshened up my Haskell knowledge by doing some exercises over these two weeks).
3. Have good idea of how to structure a large project.

4th November - 25th November

Intensity: Heavy - Main MCTS implementation coding phase

Implement MCTS library as described along with a tic-tac-toe agent. Write up preparation section of dissertation.

Milestones:

1. Satisfy the first success criterion of the project.
2. Have a fairly complete preparation section.

25th November - 2nd December

Intensity: Light - Preparing to leave Cambridge & finishing off supervisions.

Begin writing up the implementation section of the dissertation. Use the library itself & Graphviz to produce vector graphics renditions of the MCTS in action. Outline clever bits of code that I'm proud of and explain the outward facing interface to the library.

Milestones:

1. Get Graphviz producing graphs of executions of the search.
2. Have sections of the Implementation section written.

2nd December - 9th December

Intensity: No Work - On Varsity Ski Trip

Milestones: None

9th December - 16th December

Intensity: Light - CULRC[8] Henley Trial 8's camp

Make modifications to the library so that it satisfies all other success criteria. And modify tic-tac-toe agent to test the different game types.

Milestones: Satisfy all success criteria.

16th December - 6th January

Intensity: Heavy - Christmas holidays - aim to spend most of it in Cambridge. Lightening off for Christmas Day and Boxing Day.

Implement test harness, $n \times n$ tic-tac-toe agent and dummy AI agent in connectk.

Milestones: Have games playing against connectk for various board sizes and have the test harness log results of these games

6th January - 13th January

Intensity: No Work - CULRC training in Soustons, France

Milestones: None

13th January - 3rd February

Intensity: Medium - Getting back into Lectures and supervision work.

Implement the game checkers and (possibly) chess and get them working in the test harness against checkersland, chinook and GNU Chess. Evaluate the results from the $n \times n$ tic-tac-toe agent. Work on progress report for the hand in date of 3rd of February.

[8] Cambridge University Lightweight Rowing Club

Milestones:

1. Be in a position to evaluate how well the naive MCTS plays chess and checkers.
2. Hand in progress report and aim to show results from agents at the peer presentation.

3rd February - 24th February

Intensity: Hard.

Evaluate the performance of MCTS on chess and checkers and then improve on the naive MCTS playing of chess and checkers and aim to see how much it is possible to improve this.

Milestones: Have the chess + checkers AI perform better than a random agent after improvements.

24th February - 16th March

Intensity: Medium

Gather lots of statistics on games produced so far. This will consume a large amount of time since many games need to be played out and the agents will be allowed several seconds to pick a move. Evaluate what improvements I made with the chess and checkers agents.

Write up the implementation of the extensions made.

Milestones: Have a near complete evaluation section.

16th March - 23rd March

Intensity: No Work - Varsity Lightweight Boat Race

Milestones: None

23rd March - 13th April

Intensity: Medium - Need to be doing an awful lot of revision over the Easter break in addition to work.

If the project has so far gone to plan then I will implement an imperfect information game (Kriegspiel) with the intention of replicating the results in [1]. However, if there have been problems with the project so far then I will use this time as a buffer to solve those problems.

Milestones: Fix problems with project / Implement Kriegspiel agent.

13th April - 27th April

Intensity: Hard - Lead up to imaginary deadline before getting into solid revision

Dissertation and code polishing time. Write things like the introduction section now that I know a lot more about the subject. Also, write up the conclusions I have made given the work I have done on the project.

Milestones: Complete (almost) final draft of dissertation, distribute around friends and academics, ensure everything is checked.

27th April - 18th May

Intensity: Extra Light - Spend most of time revising & attending lectures.

I will spend this time incorporating feedback from others and improving parts of the dissertation which seem insufficient.

Milestones: Hand in Dissertation

References

[1] Paolo Ciancarini and Gian Piero Favini. Monte carlo tree search in kriegspiel. *Artificial Intelligence*, 174(11):670 – 684, 2010.

[2] Hilmar Finnsson and Yngvi Bjornsson. Simulation-based approach to general game playing. In *AAAI'08 Proceedings of the 23rd national conference on Artificial intelligence*, volume 1, 2008.

[3] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856 – 1875, 2011.

[4] S. Samothrakis, D. Robles, and S. Lucas. Fast approximate max-n monte-carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[5] Jonathan Schaeffer, Yngvi Bjornsson, Neil Burch, Akihiro Kishimoto, Martin Muller, Rob Lake, Paul Lu, and Steve Sutphen. Solving checkers. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 292–297, 2005.

[6] Mark H. M. Winands and Yngvi Bjrnsson. Monte-carlo tree search solver. In *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*, Lecture Notes in Computer Science.

[7] Mark H.M. Winands and Yngvi Bjrnsson. Evaluation function based monte-carlo loa. *Advances in Computer Games*, 2009.