Twist: a trivially simple draughts engine + extensible microframework for game-playing agents

Tobia Tesan

Abstract

In this document we give an overview of adversarial search techniques and introduce the game of draughts and its role in AI research in section 1.

We detail the requirements and implementation of a general framework capable of hosting game-playing agents in section 2 and the implementation of a minimally simplified draughtsboard in subsection 2.1.

We quickly review the theoretical foundations and discuss the implementation of an Alpha-Beta agent equipped with quiescence search and the killer heuristic in subsection 3.1 and of a UCT-based agent in subsubsection 3.2.1.

Finally, in section 4 we draw conclusions and propose future extensions.

Contents

List of Tables

List of Figures

List of Listings

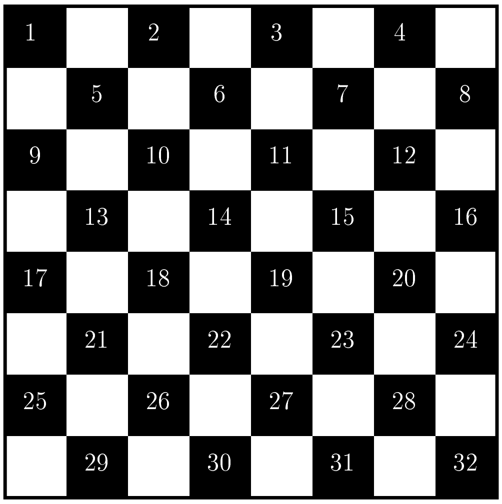

Figure 1: Draughtsboard with PDN numbering. Red starts on $1 \div 12$, White starts on $21 \div 32$; $1 \div 4$ and $29 \div 32$ are called "King's Row"; for a man to reach King's Row on the opposite side of the board results in a promotion to king.

1 Introduction

Draughts is a well-known family of two-player games, the most played of which is English draughts (or American checkers).

The rules issued by the World Checkers Draughts Federation [Fed12], which claims for itself the title of "official world governing body for the game of Checkers", define draughts as "a board game of skill played between two players who, following a fixed set of rules, attempt to win the game by either removing all of their opponent's playing pieces from the draughts board, or by rendering their opponent's pieces immobile."

The draughtsboard for English draughts with its 32 squares is reproduced in Figure 1, with squares numbered according to PDN (Portable Draughts Notation); from here on, with "draughts" we will be referring to the game of English draughts as defined by the WCDF rules, which will be assumed to be known. PDN numbering will also be liberally used in code fragments.

With a branching factor estimated as 2.5 by Lu [Lu93] and 6.4 by Guerra [Gue11] draughts is a somewhat less complex game than the "drosophila of artificial intelligence", chess, with its branching factor estimated to be around 40; this hasn't prevented several high-profile attempts at devising draughts-playing engines that include early efforts by Samuel [Sam67] and, most famously, Chinook, written by the team of Jonathan Schaeffer, which proved a worthy opponent for world champion Marion Tinsley in 1996. [Sch+07]

Draughts is a weakly solved game, in other words perfect play by both sides leads to a draw [Sch+07]; moreover, in the classical framework of game theory Draughts is a two-player, deterministic, perfect-information game: for this class of games variations on the minimax algorithm have been standard since Shannon's seminal 1950 work [Sha50] but are currently being challenged by Monte Carlo methods, particularly since Sylvain Gelly et al's successful implementation [GW06] of Kocsis and Szepesvari's UCT algorithm [KS06] in the Go engine MoGo.

2 A micro-framework for two-player games

Despite the popularity of computer draughts, when we set out to extend an existing program with MTCS or advanced evaluation functions and set up a framework to carry out benchmarks we were faced with a lack of appropriate programs in source code form in the public domain or under a sufficiently permissive license.

More precisely, nearly all programs we found found fell into one of the following categories:

- "Industrial-strength" programs, generally hard to extend and modify, and either
  - architecturally complex
  - highly optimized at the expense of simplicity
  - low-level (e.g. written in C with extensive manual menory handling)
- Programs too tightly coupled with a specific agent or class of agent (typically Minimax)
- Programs too tightly coupled with their UI and/or with the human vs IA mode of play, hard to extend in order to allow for IA vs. IA play and statistics collection.

Requirements We eventually resolved to write our own, with the aim that it be

- Architecturally simple
- Strongly decoupled, in particular regarding agents and games, and versatile
- Written in a high-level language

Choice of programming language Scala [Ode] was chosen as a programming language; amongs its benefits are its high portability thanks to the JVM (and, indeed, its ability to go beyond what the JVM affords – for example the ability of its dialect Scala.js to run in a browser [Doe13]), a reasonable type system that can make debugging and writing correct programs easier and its ability to mix stateless, expressive functional programming with imperative programming with side effects, which is ideal for writing high-level functional code while retaining the ability enter algorithms found in literature in ALGOL-like languages verbatim.

Architecture The code is available at [Twist].

The architecture is almost entirely specified by files `Game.scala` and `Player.scala`, shown in Listing 1 and Listing 2.

In `Game.scala` positions are partitioned into `TerminalPositions` and their complement, `LivePositions`[1].

[1] `LivePosition` has nothing to do with the notion of "dead position", related to that of quiescence, introduced by Turing in 1950.

```scala
trait Game[G <: Game[G]] {
  def startingPosition(): LivePosition[G]
}

trait Move[G <: Game[G]] {}

sealed abstract class Side {
  def opposite(): Side
}

case object Min extends Side {
  def opposite() = Max
}

case object Max extends Side {
  def opposite() = Min
}

trait TerminalPosition[G <: Game[G]] {

  /**
   * @return 0 if draw, -1 if Min wins, +1 if Max wins
   */
  def utility: Integer
}

/**
 * A position that is not terminal
 */
trait LivePosition[G <: Game[G]] {

  /**
   * @return the side to move, /if/ there are available moves
   */
  def sideToMove: Side

  /**
   * @return a non-empty Move -> Position map
   *
   * Unless the search space is trivial it's advisable to lazily evaluate positions
   */
  def successor(): Map[Move[G], Either[LivePosition[G], TerminalPosition[G]]]
}
```

Listing 1: `Game.scala`

```scala
trait DebugStats[+A <: AI[_]] {
  def getNodes: Int
  def toString: String
}

case class MoveWithStats[G <: Game[G], +M <: Move[G], +S <: DebugStats[AI[G]]](
    val move: M,
    val stats: S
)

sealed trait Player[G <: Game[G]] {
  def apply(p: LivePosition[G]): Move[G]
}
trait Human[G <: Game[G]] extends Player[G]
trait AI[G <: Game[G]] extends Player[G] {
  def debug(p: LivePosition[G]): MoveWithStats[G, Move[G], DebugStats[AI[G]]]
  def apply(p: LivePosition[G]): Move[G] = debug(p).move
}
```

Listing 2: `Player.scala`; in order to implement an agent it is sufficient to extend `AI` with an appropriate implementation of `debug`, which returns a move and a `DebugStats` object containing statistics about the search that yielded said move (e.g. how many nodes expanded, how many cuts...)

The method `successor`[2] acts as the authoritative generator of legal moves and returns a map of moves to successor states; lazy evaluation, afforded by Scala, is necessary to make this simple interface viable for games with non-trivial state space.

2.1 Implementation of the game of Draughts

An implementation of the game of Draughts is to be found in `Draughts.scala`, accompained by a test suite living under `test/`.

The implementation of the game will not be discussed at length nor reproduced here, but for the details of practical relevance that follow.

Firstly, our board implements faithfully the WCDF rules with the exception of the definition of a draw given in §1.32.

For simplicity **a draw is reached after a fixed number of plys have been played without a winner**. The default number of plys is set to 100; Guerra gives the average game length as 60 ply [Gue11].

Moreover, we work under the implicit assumption that utility is in the range $[-1, 1]$, where 1 and -1 are the utility of a win for either side, as suggested in [Sha50].

[2] Modeled after the definitions presented in the second edition of [AIMA], rather than those in the third ed. that include an explicit `result` function

3 Agents

3.1 Minimax

The Minimax algorithm, first introduced by Shannon in [Sha50] and illustrated at length in [AIMA], performs a complete depth-first exploration of the game tree in order to derive a *minimax value* for each child node and play the optimal move.

Its naive implementation has time complexity $O(b^m)$ and space complexity $O(bm)$ [AIMA] and is thus impractical; its most popular refinement is α/β pruning, likely independently invented in the 1950s by various authors.

Alpha-Beta pruning, also illustrated in great detail in [AIMA], propagates heuristic bounds on the value of a position while traversing the game tree, corresponding to the minimum (resp. maximum) value that the agent about to move (resp. agent's opponent) can achieve.

Subtrees that are outside this range are cut off, resulting, in the best case, in a $O(b^{m/2})$ complexity.

In our family of Minimax implementations we will use the equivalent Negamax formulation, described in [KM75], for ease of implementation, in which the minimax value F of a position p is defined as

$$F(p) = \begin{cases} f(p) & d = 0 \\ \max(-F(p_1), \ldots, -F(p_d)) & d > 0 \end{cases}$$

Implementation The skeleton of our Alpha-Beta implementation, found in the class `AbstractAlphaBeta` inside `Minimax.scala`, is shown in Listing 3.

The most basic concretization of `AbstractAlphaBeta` – the classical Alpha-Beta algorithm – is implemented in class `BasicAlphaBeta`, reproduced in appendix A.2.

Notice how `AbstractAlphaBeta[G]` expects that an `Evaluation` object, appropriate for the game `G` we want to apply our agent to, be passed as parameter.

A basic evaluation function is provided for the game of draughts in `Draughts.scala` and shown in figure Listing 4, along with further evaluation functions to be discussed in ??.

A cutoff is implemented by throwing an exception `Cut`, which will forcibly abandon the evaluation of current subtree.

Move ordering It is a well known fact that the efficiency of Alpha-Beta in terms of expanded nodes over naive Minimax is fully realized only through an appropriate ordering of moves.

We have provided a simple ordering function in `BasicDraughtsMoveOrdering` (not reproduced) that privileges

1. among capture moves, the ones that maximize the amount of captured pieces
2. among ordinary, non-capturing moves, those that reach a farther place on the board or travel a longer distance.

```scala
abstract class AbstractAlphaBeta[G <: Game[G]](e: MinimaxEvaluation[G],
                                               o: AlphaBetaOrdering[G],
                                               depth: Int,
                                               maximize: Boolean = false)
    extends AI[G] {
  def onTerminal(t: TerminalPosition[G],
                 plyLeft: Int,
                 nega: Int): (Double, AlphaBetaStats[G])

  def onStatic(l: LivePosition[G],
               plyLeft: Int,
               nega: Int): (Double, AlphaBetaStats[G])

  def otherwise(l: LivePosition[G],
                plyLeft: Int,
                alpha: Double,
                beta: Double,
                nega: Int): (Double, AlphaBetaStats[G])

  case class Cut(val v: Double, val stats: AlphaBetaStats[G]) extends Exception

  def iter(p: Either[LivePosition[G], TerminalPosition[G]],
           plyLeft: Int,
           alpha: Double = Double.NegativeInfinity,
           beta: Double = Double.PositiveInfinity,
           nega: Int = 1): (Double, AlphaBetaStats[G]) =
    p match {
      case Right(t: TerminalPosition[G]) =>
        onTerminal(t, plyLeft, nega)
      case Left(l: LivePosition[G]) =>
        if (plyLeft <= 0)
          onStatic(l, plyLeft, nega)
        else
          otherwise(l, plyLeft, alpha, beta, nega)
    }

  def debug(
      p: LivePosition[G]): MoveWithStats[G, Move[G], AlphaBetaStats[G]] = {
    val evaluatedMoves = p.successor.map(mp => mp._1 -> iter(mp._2, depth))
    val cumulativeStats = evaluatedMoves
      .map(_._2._2)
      .fold(new AlphaBetaStats[G](0, 0, 0, 0))(_ + _)
    val rankedMoves = evaluatedMoves.toList
      .map((t: ((Move[G]), (Double, AlphaBetaStats[G]))) => (t._1, t._2._1)) // Discard stats
      .sortWith(_._2 < _._2)
    if (maximize)
      MoveWithStats[G, Move[G], AlphaBetaStats[G]](rankedMoves.last._1,
                                                   cumulativeStats)
    else
      MoveWithStats[G, Move[G], AlphaBetaStats[G]](rankedMoves.head._1,
                                                   cumulativeStats)
  }
}
```

Listing 3: AbstractAlphaBeta

The assumption that motivates the second point is that the ordering is particularly critical in the early phases of the game, when the search space is larger and deeper[3], and in opening and middle game players typically will try to advance, in order to ultimately reach King's Row and obtain a promotion.

Subsection 3.1.2 discusses the killer heuristic as an improvement.

```
object NaiveDraughtsEvaluation extends MinimaxEvaluation[Draughts] {
  def apply(p: LivePosition[Draughts]): Double =
    p match {
      case (p: LiveDraughtsPosition) =>
        (p.board.filter(_ == Some(Man(Max))).size * 1 +
         p.board.filter(_ == Some(Man(Min))).size * -1 +
         p.board.filter(_ == Some(King(Max))).size * 2 +
         p.board.filter(_ == Some(King(Min))).size * -2).toDouble / 24

    }
}
```

Listing 4: `NaiveDraughtsEvaluation`

3.1.1 Quiescence Search Minimax

A key improvement in minimax-like algorithms is quiescence search, used to contrast the "horizon effect", which is the tendency of naive Minimax to be blind to developments that human players would call "obvious" in consequence of a given move when they occur one or more moves after the cutoff depth.

A typical example is found in Figure 2: assuming White is to move, if Red were to take a material-based static evaluation for the position the resulting estimate might be exceedingly optimistic.

The horizon effect is acknowledged as early as [GEC67], where it is countered with the "secondary search" approach.

The evaluation function is therefore trustworthy only for positions that are *quiescent*.

The idea of quiescence search is to equip the agent with some heuristic function to determine quiescence and, if necessary, carry out a further search for nonquiescent positions that extends beyond cutoff depth.

Implementation We have equipped the Alpha-Beta algorithm with quiescence search by integrating a further instance of `BasicAlphaBeta`, as seen in Listing 5.

Our augmented agent launches a simple minimax search of a given depth `extraPlys` if at cutoff depth the position is non-quiescent.

We have chosen to use a naive implementation that inevitably takes the static evaluation after `extraPlys` further plys, even if the position reached then is nonquiescent, as this is the simplest way to guarantee termination irrespective of the choice of `q`.

[3]This is trivially true under the assumption that a draw is forced after the n-th ply without a winner, as discussed in subsection 2.1

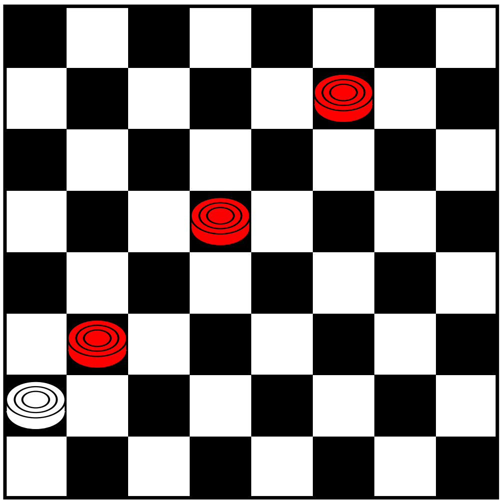

Figure 2: A nonquiescent position. White moves.

```scala
/**
 * Simple AlphaBeta enhanced with quiescence search
 *
 * @param extraPlys fixed depth of extra search if "ordinary" search depth is exhausted
 *  on a non-quiescent position
 */
class QuiescenceAlphaBeta[G <: Game[G]](e: MinimaxEvaluation[G],
                                        o: AlphaBetaOrdering[G],
                                        depth: Int,
                                        q: QuiescenceCheck[G],
                                        extraPlys: Int,
                                        maximize: Boolean = false)
    extends BasicAlphaBeta[G](e, o, depth, maximize) {
  override def iter(l: Either[LivePosition[G], TerminalPosition[G]],
                    plyLeft: Int,
                    alpha: Double = Double.NegativeInfinity,
                    beta: Double = Double.PositiveInfinity,
                    nega: Int = 1): (Double, AlphaBetaStats[G]) =
    l match {
      case Right(t: TerminalPosition[G]) => onTerminal(t, plyLeft, nega)
      case Left(l: LivePosition[G]) =>
        if (plyLeft <= 0)
          if (q(l))
            onStatic(l, plyLeft, nega)
          else
            // Position is non-quiescent, we do a further local, small
            // minimax search
            (new BasicAlphaBeta[G](e, o, extraPlys, maximize))
              .iter(Left(l), extraPlys, alpha, beta, nega)
        else
          otherwise(l, plyLeft, alpha, beta, nega)
    }
}
```

Listing 5: `QuiescenceAlphaBeta`

Notice how the quiescence search contributes to the collected statistics, which will allow us to estimate if and how it is competitive with a conventional minimax search search of depth `ply + extraPly`.

3.1.2 Killer Heuristic

Literature describes several variations on the “killer heuristic”; Gillogly, who used it in the TECH program, gives the common underlying inutition as follows: “a move which generates a prune in one set of moves may also generate a prune in the adjacent set (first cousin positions) so that this «killer» move should be tried first” [Gil72].

Therefore, programs using some form of killer heuristic will save moves that are refutations on a killer list or buffer, and examine the moves at each node as they are generated to see whether one of them matches a move on the killer list [Gil72].

According to [AN77], implementations can differ, among other things, on whether a separate list is kept for each ply.

We have therefore provided two separate implementations, `SimpleKillerAlphaBeta` (reproduced in appendix A.3) and `KillerAlphaBeta` (not reproduced): the former uses a single killer list, whereas the latter uses a separate killer list for each ply.

We have chosen for simplicity to make the killer list not persistent between different searches; this has been experimentally shown to suffice in yielding an appreciable gain in **??**.

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search, shortened in MTCS, is a kind of simulation-based search [GS11]; it belongs to a family of search algorithms that evaluate nodes by repeatedly carrying out *simulations* – i.e. descending a single path in a game tree according to a randomized *simulation policy* – in order to estimate their utility.

Abramson first introduced the expected outcome model in [Abr90] and shown that the game-theoretic value of a game-tree node is approximated by the expected value of the game’s outcome given random play from that node on.

An important difference with Minimax-based approaches is that the expected outcome approach considers the relative merit of game-tree nodes rather than board positions, and is thus, in its simplest formulation, independent of domain knowledge.

“Flat Monte Carlo” is the simplest simulation search algorithm and proceeds by uniform selection [Bro+12]; its drawback is that it makes an ineffective use of computational resources.

Intuitively, uniform selection implies that an equal number of simulations are spent on “interesting” and “uninteresting” moves alike. In the extreme case, if a position allows for exactly three moves, one of which directly leads to a loss and the other two are root to a large and “interesting” subtree that may or may not lead to wins, losses and draws, at the limit $1/3$ of simulations will be spent on the former move, when they could be spent on gathering more information for moves two and three.

We would like to include features of a Shannon Type B strategy (one that focuses on “plausible moves”) in flat Monte Carlo, which in the limit is a Shannon type A strategy, by focusing our efforts on promising moves – however, we wouldn't want to run the risk of discarding a good move after a few unlucky simulations.

The Monte Carlo Tree Search algorithm thus incorporates the following steps [Bro+12] in every iteration, until a pre-allocated budget is exhausted:

- Starting from the root node, the tree is descended according to the *simulation policy*, until a non-terminal state with unexpanded children is reached and a node is then chosen to be expanded.
- A simulation is carried out according to the *default policy*.
- The result of the simulation is backpropagated through the parent nodes.

Figure 3, excerpted from [GS11], illustrates the progression of the algorithm on a game tree.

3.2.1 Bandit problems and UCT

As mentioned in the previous paragraph, we would like to focus on moves that appear to be "interesting", but we wouldn't want to run the risk of discarding a good move based on inconclusive data.

In this respect, our search problem can be framed as a "bandit problem", an element in a class of problems akin to that of repeatedly choosing to play one out of k slot machines[4] with distinct, unknown distributions of rewards in order to maximise the cumulative reward.

This is the nature of the *exploitation-exploration dilemma* studied in bandit problems [Bro+12], i.e. the need to balance exploitation the "arm" believed to be optimal at a given time with the need to gather more information in order to have adequate confidence on the underlying distribution of arms that appear suboptimal.

Kocsis and Szepesvári [KS06] have shown that in Monte-Carlo tree search it is possible to treat each state of the search tree as a multi-armed bandit, in which each action corresponds to an arm of the bandit and have proposed UCT, an extremely popular variant of MTCS.

In UCT, the tree policy selects actions by using the UCB1 algorithm, which maximises an upper confidence bound on the value of actions; UCT is proven to converge on the minimax action value function.

When applying the tree policy to a node s, a child node a is selected to maximise

$$\underbrace{Q(s,m)}_{\text{MC value}} + \underbrace{c\sqrt{\frac{\log N(s)}{N(s,m)}}}_{\text{exploration bonus}}$$

where $Q(s,m)$ is the "Monte-Carlo value" of taking move m from the position corresponding to node s – or the mean outcome of previous all simulations, $N(s,m)$ is the number of times move m has been taken from s previously and $N(s)$ is the number of times s has been expanded; the value is augmented by an exploration bonus that is highest for rarely visited state-action pairs [GS11].

[4]known as "one-armed bandits"

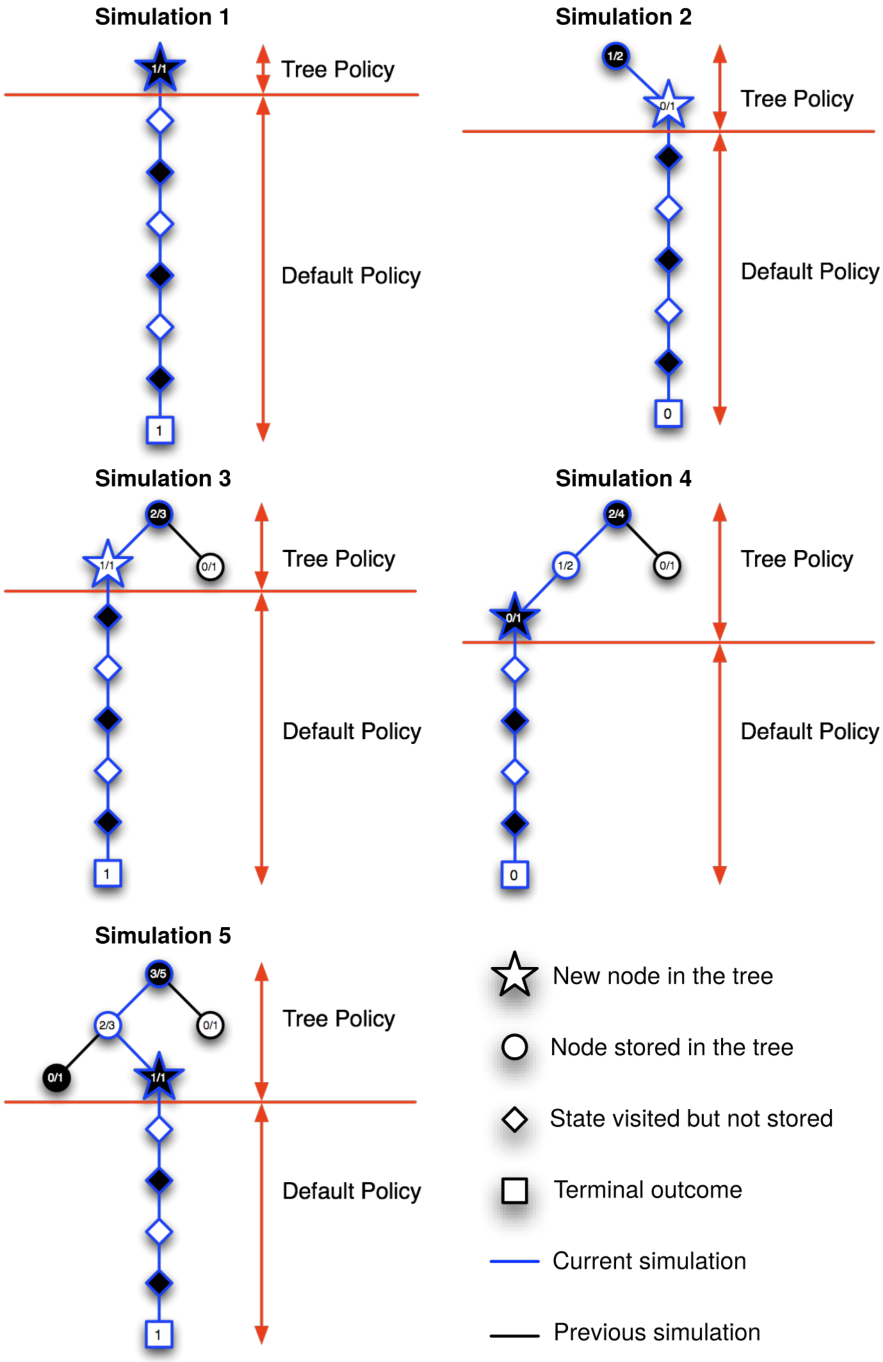

Figure 3: Five iterations of MTCS (from [GS11])

Implementation In `MTCS.scala`, reproduced in appendix A.1, we give a recursive implementation that follows closely the pseudocode for UCT reported in [GS11].

The budget is measured in total nodes expanded instead of wall clock time, in order to make it independent of optimizations in the implementation (or lack thereof).

Although domain knowledge can and often is incorporated in the tree policy – in fact, according to [Bro+12], the full benefit of MCTS is "typically not realised" until the algorithm is thus adapted – this basic implementation of MTCS is domain-agnostic.

4 Conclusion and future work

We have presented an architecture that can host different sorts of two-player, deterministic games and different sorts of agents that play such games.

We have presented an implementation of the game of draughts and two agents, one based on classical minimax and one based on MTCS.

A number of possible extensions and optimizations can be thought of; we give a necessarily non-exhaustive list.

The simplest possible extension probably is the addition of a new game using the framework specified in `Game.scala`.

Furthermore, the framework itself could be extended as to allow for games with more than two players, stochastic games such as Backgammon or games of imperfect information, for example through a subclassing of `Position` and a modification of class `Match` in order to serve an appropriately "obfuscated" version of each position to different players.

The UCT implementation could be extended with domain-specific optimizations and additional heuristics such as the history heuristic could be implemented in the Alpha-Beta agent.

Given enough CPU time, it would be interesting to run extensive benchmarks to characterize precisely the performance of agents and fit a function to predict the optimal value of c for UCT in relation to the allocated budget.

A further interesting nontrivial extension might be using a genetic, evolutionary or swarm-based approach to derive optimal evaluation functions.

While hardly an essential improvement, the usability of the framework also could be improved (thus facilitating the more fundamental additions discussed in the previous paragraphs) by providing an extended framework for benchmarks, extending class `Match` in order to realize a client-server model or, finally, adding a graphical user interface for play and visualization of statistics, perhaps leveraging Scala.js for in-browser execution.

References

[Abr90] Bruce Abramson. “Expected-outcome: A general model of static evaluation”. In: *IEEE transactions on pattern analysis and machine intelligence* 12.2 (1990), pp. 182–193.

[AIMA] Stuart Russell, Peter Norvig, and Artificial Intelligence. “Artificial Intelligence: A Modern Approach”. In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* (1995).

[AN77] Selim G Akl and Monroe M Newborn. “The principal continuation and the killer heuristic”. In: *Proceedings of the 1977 annual conference*. ACM. 1977, pp. 466–473.

[Bro+12] Cameron B Browne et al. “A survey of monte carlo tree search methods”. In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[Doe13] Sébastien Doeraene. *Scala. js: Type-directed interoperability with dynamically typed languages*. Tech. rep. 2013.

[Fed12] World Draughts Checkers Federation. *Rules of Checkers*. 2012.

[GEC67] Richard D Greenblatt, Donald E Eastlake III, and Stephen D Crocker. “The Greenblatt chess program”. In: *Proceedings of the November 14-16, 1967, fall joint computer conference*. ACM. 1967, pp. 801–810.

[Gil72] James J Gillogly. “The technology chess program”. In: *Artificial Intelligence* 3 (1972), pp. 145–163.

[GS11] Sylvain Gelly and David Silver. “Monte-Carlo tree search and rapid action value estimation in computer Go”. In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1875.

[Gue11] Joao Carlos Correia Guerra. “Classical Checkers”. PhD thesis. Master’s thesis, Instituto Superior Técnico, 2011.

[GW06] Sylvain Gelly and Yizao Wang. “Exploration exploitation in go: UCT for Monte-Carlo go”. In: *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*. 2006.

[KM75] Donald E Knuth and Ronald W Moore. “An analysis of alpha-beta pruning”. In: *Artificial intelligence* 6.4 (1975), pp. 293–326.

[KS06] Levente Kocsis and Csaba Szepesvári. “Bandit based Monte-Carlo planning”. In: *ECML*. Vol. 6. Springer. 2006, pp. 282–293.

[Lu93] Chien-Ping Paul Lu. “Parallel search of narrow game trees”. MA thesis. University of Alberta, 1993.

[Ode] Martin Odersky. *The Scala language specification*.

[Sam67] Arthur L Samuel. “Some studies in machine learning using the game of checkers. II—recent progress”. In: *IBM Journal of research and development* 11.6 (1967), pp. 601–617.

[Sch+07] Jonathan Schaeffer et al. “Checkers is solved”. In: *science* 317.5844 (2007), pp. 1518–1522.

[Sha50] Claude E Shannon. “XXII. Programming a computer for playing chess”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[Twist] Tobia Tesan. *Twist*. `https://github.com/tobiatesan/twist`. 2017.

A Long listings

A.1 UCT Agent

```scala
/**
 * Implements an (inefficient) UCT agent based on Gelly 2011
 *
 * @param budget the number of total nodes (including default policy) to explore before stopping
 * @param c the exploration constant; the larger it is, the more the algorithm favors exploration
 *  over exploitation.
 */
class UCTAgent[G <: Game[G]](val budget: Int,
                            val c: Double,
                            val maximize: Boolean = false,
                            val r: Random)
  extends AI[G] {
  type Utility = Int
  type NodeCount = Int

  def SimDefault[G <: Game[G]](
      p: Either[LivePosition[G], TerminalPosition[G]]): (Utility, NodeCount) =
    p match {
      case Right(t: TerminalPosition[G]) => (t.utility, 0)
      case Left(l: LivePosition[G]) => {
        val map = l.successor()
        val randomMove = r.shuffle(map.keys.toList).head
        val (util, nodes) = SimDefault((map.get(randomMove).get))
        (util, nodes + 1)
      }
    }

  def SelectMove[G <: Game[G]](p: LivePosition[G],
                              t: UCTNode[G],
                              c: Double): Move[G] = {
    def N_s: Double = t.den.toDouble
    def N_s_a(a: Move[G]): Double =
      t.children.get(a).map(_.den).getOrElse(0).toDouble
    def Q_s_a(a: Move[G]): Double =
      t.children.get(a).map((a) => a.num / a.den).getOrElse(0).toDouble
    def argmax[A](a: Seq[A], f: A => Double) = a.sortBy[Double](f).last
    def argmin[A](a: Seq[A], f: A => Double) = a.sortBy[Double](f).head
    if (t.maxNode)
      argmax(p.successor.keys.toSeq,
             (a: Move[G]) => Q_s_a(a) + c * Math.sqrt(Math.log(N_s) / N_s_a(a)))
    else
      argmin(p.successor.keys.toSeq,
             (a: Move[G]) => Q_s_a(a) - c * Math.sqrt(Math.log(N_s) / N_s_a(a)))
  }

  /**
   * @return Updated tree, spent budget and value of last simulation to be propagated
   */
  def Simulate(p: LivePosition[G],
```

```
            node: UCTNode[G]): (UCTNode[G], NodeCount, Utility) = {
  val move = SelectMove(p, node, c)
  p.successor.get(move).get match {
    case (Right(t)) =>
      // Terminal game position, return utility
      (UCTNode(p,
              node.num + t.utility,
              node.den + 1,
              node.maxNode,
              node.children),
       1,
       t.utility)
    case (Left(l)) =>
      if (node.children contains move) {
        // Already in tree and not terminal, continue with tree policy
        ((rec: (UCTNode[G], NodeCount, Utility)) => {
          (UCTNode(p,
                  node.num + rec._3, // Backup
                  node.den + 1,
                  node.maxNode,
                  node.children updated (move, rec._1)),
           rec._2 + 1,
           rec._3)
        })(Simulate(l, node.children.get(move).get))
      } else {
        // Not in tree, not terminal: expland
        ((sim: (Utility, NodeCount)) => {
          (UCTNode[G](p,
                     node.num + sim._1,
                     node.den + 1,
                     node.maxNode,
                     (node.children updated (move,
                     UCTNode(l, sim._1, 1, !(node.maxNode), Map.empty)))),
           sim._2 + 1,
           sim._1)
        })(SimDefault(Left(l)))
      }
  }
}

def UCTSearch(p: LivePosition[G],
             budget: NodeCount): (Move[G], NodeCount, UCTNode[G]) = {
  var tree = new UCTNode[G](p, 0, 0, maximize, Map.empty)
  var budget_* = budget
  while (budget_* > 0) {
    val (newtree, spent, _) = Simulate(p, tree)
    tree = newtree
    budget_* = budget_* - spent
  }
  (SelectMove(p, tree, 0), budget - budget_*, tree)
}

def debug(
    p: LivePosition[G]): MoveWithStats[G, Move[G], DebugStats[UCTAgent[G]]] =
  ((x: (Move[G], Int, UCTNode[G])) =>
```

```
    MoveWithStats[G, Move[G], DebugStats[UCTAgent[G]]](
      x._1,
      new UCTStats(x._2, x._3)))(UCTSearch(p, budget))
}
```

A.2 BasicAlphaBeta

```
class BasicAlphaBeta[G <: Game[G]](e: MinimaxEvaluation[G],
                                   o: AlphaBetaOrdering[G],
                                   depth: Int,
                                   maximize: Boolean = false)
    extends AbstractAlphaBeta[G](e, o, depth, maximize) {
  def onTerminal(t: TerminalPosition[G],
                 plyLeft: Int,
                 nega: Int): (Double, AlphaBetaStats[G]) = {
    (nega * t.utility.toDouble, new AlphaBetaStats(0, 1, 0, 0))
  }

  def onStatic(l: LivePosition[G],
               plyLeft: Int,
               nega: Int): (Double, AlphaBetaStats[G]) = {
    (nega * e(l), new AlphaBetaStats(0, 0, 1, 0))
  }

  def otherwise(p: LivePosition[G],
                plyLeft: Int,
                alpha: Double,
                beta: Double,
                nega: Int): (Double, AlphaBetaStats[G]) = {
    var alpha_* = alpha
    var runningStats = new AlphaBetaStats[G](1, 0, 0, 0)
    try {
      p.successor.toSeq
        .sortWith((x: (Move[G], _), y: (Move[G], _)) => o.lt(x._1, y._1))
        .foreach { m =>
          {
            val (negv, stats) =
              this.iter(m._2, plyLeft - 1, -beta, -alpha_*, -nega)
            val v = -negv
            alpha_* = max(alpha_*, v)
            runningStats = runningStats + stats
            if (alpha_* >= beta)
              throw new Cut(alpha_*, runningStats)
          }
        }
      (alpha_*, runningStats)
    } catch {
      case (p: Cut) => (p.v, p.stats + new AlphaBetaStats[G](0, 0, 0, 1))
    }
  }
}
```

A.3 SimpleKillerAlphaBeta

```scala
/**
 * Simple extension of BasicAlphawBeta with killer heuristic
 * implemented with a _single_ list for all plys.
 *
 * According to Akl77, "programs differ in the number of killer moves
 * saved, the number of matches looked for, and on whether a separate
 * list is kept for each ply", so there is at least precedent.
 *
 * @param killerSize *total* size of the killer list
 */
class SimpleKillerAlphaBeta[G <: Game[G]](e: MinimaxEvaluation[G],
                                o: AlphaBetaOrdering[G],
                                depth: Int,
                                killerSize: Int = 10,
                                maximize: Boolean = false)
  extends BasicAlphaBeta[G](e, o, depth, maximize) {

 var killerList: FiniteQueue[Move[G]] = new FiniteQueue(Queue(), killerSize)

 object killerOrdering extends Ordering[Move[G]] {
  /*
   * if x is on the killer list and y is not x < y (comes first)
   * if y   "    "   then y < x
   * defer to usual ordering otherwise
   */
  def compare(x: Move[G], y: Move[G]): Int = {
   if (killerList.contains(x) && !killerList.contains(y))
    -1
   else if (!killerList.contains(x) && killerList.contains(y))
    +1
   else
    o.compare(x, y)
  }
 }

 override def debug(
    p: LivePosition[G]): MoveWithStats[G, Move[G], AlphaBetaStats[G]] = {
  // Wipe killer list at each new search
  killerList = new FiniteQueue(Queue(), killerSize)
  super.debug(p)
 }

 override def otherwise(l: LivePosition[G],
                 plyLeft: Int,
                 alpha: Double,
                 beta: Double,
                 nega: Int = 1): (Double, AlphaBetaStats[G]) = {
  var alpha_* = alpha
  var runningStats = new AlphaBetaStats[G](1, 0, 0, 0)

  try {
   l.successor()
    .toSeq
```

```
      .sortWith((x: (Move[G], _), y: (Move[G], _)) =>
        killerOrdering.lt(x._1, y._1))
      .foreach { m =>
        {
          val (negv, stats: AlphaBetaStats[G]) =
            this.iter(m._2, plyLeft - 1, -beta, -alpha_*, -nega)
          val v = -negv
          alpha_* = max(alpha_*, v)
          runningStats += (stats)
          if (alpha_* >= beta) {
            killerList = killerList.enqueue(m._1)
            throw new Cut(alpha_*, runningStats)
          }
        }
      }
    (alpha_*, runningStats)
  } catch {
    case (c: Cut) => {
      (c.v, c.stats + new AlphaBetaStats[G](0, 0, 0, 1))
    }
  }
 }
}
```