A Comparison of Monte-Carlo Approaches for General Game Playing

Abstract-

Context/Background

Traditionally, Artificial Intelligence for games playing has focused on developing a player for a particular game, using pre-defined knowledge that is relevant only to the game they are designed to play. Performance in other games is typically poor (if they work at all). General Game Playing involves the design of agents that are able to successfully play more than one game. To accomplish this, the agent must be able to transform knowledge about the game into an evaluation function and modify strategy based on feedback. Reinforcement Learning techniques are concerned with how an agent should select actions in an environment in order to maximise a reward. Monte-Carlo Search methods can offer good performance while with domain-independent evaluation and this is where the project will focus.

Aims

This project aims to design and implement some general game playing agents that use Monte-Carlo search techniques and evaluate their performance in a number of different games. By analysing how successful the agents are in different games, the project will establish whether one approach gives the best general performance or different approaches are required depending on the characteristics of the game. Given their dominance in the area of general game playing, the focus will be on Monte Carlo Search-based approaches.

Method

Agents will extend a common base implementation of a player and interact via a game server which will manage the games and communication. Testing will consist of the agents playing against one another to establish whether one agent outperforms the others in most games.

Proposed Solution

Implement a number of general game playing agents that each use a different reinforcement learning technique. These resulting programs will be built upon an existing framework for developing general game playing agents which includes a number of features that would otherwise be too time consuming to implement. Following development, algorithms will be tested using a predetermined set of parameters for turn time etc. The games used in testing will be Connect-Four, Checkers and Pentago.

Keywords

AI, general game playing, reinforcement learning, Monte Carlo Search

I. Introduction

The purpose of this project is to examine the performance of a different Monte-Carlo based reinforcement learning approaches to the problem of general game playing. A number of players will be implemented, each using a different learning algorithm to select moves and a number of enhancements will be discussed and implemented to attempt to create a player that is able to make good choices when playing against other general game playing agents.

Agents and the server through which they communicate will be set up using the GGP-Base package, which takes includes enough functionality that the focus of development can be the algorithms themselves rather than systems for managing games, communication between players etc.

The agents will be tested in a competition-like environment where they will play a number of games against one another in the games Checkers, Connect Four and Pentago to establish which algorithm gives the best general performance.

Reinforcement Learning

Monte-Carlo Search is a form of reinforcement learning. This is an area of machine learning that is concerned with the problem of designing agents that can act on an environment to maximise some cumulative reward. The reinforcement learning problem consists of:

- The agent - decision maker
- The environment - things the agent interacts with. Often, we consider parts of an entity to be part of the environment, rather than the agent. For example motors and actuators are typically considered this way.
- The reward computation - External to the agent, although the agent may have some knowledge of it.
- The agent-environment boundary - this represents the limit of the agent's absolute control, not of its knowledge.

The reinforcement learning agent interacts with the environment in discrete time steps, at which an action is chosen and a reward is determined based on the transition from the current state to the new state, given the chosen action. Reinforcement learning agents are not supplied with correct input/output pairs (as with supervised learning) and must balance exploration of the state space with exploitation of its current knowledge to find a good course of action.

Environments are formulated as *Markov Decision Processes (MDP)*. An MDP describes a set of transitions between states based on available actions at each step, receiving a corresponding award computed by a reward function. States in an MDP display the *Markov Property*, which is to say that each state summarises the sequence of actions/states that preceded it. The meaning of the state is independent from the history of how it was arrived at.

General Game Playing

Creating Artificial Intelligence for individual games has been a long-running area of interest for computer programmers. There are now programs that can compete (and win) at the highest levels for games such as chess, backgammon and checkers. These programs rely heavily upon specialist knowledge of the game, often having access to books of opening and endgame strategies for example, and on the expertise of the programmer. Much of the 'intelligent' work is carried

out by the system's designers before it ever comes online. Whilst exhibiting expert play in their particular game, though, these systems are very limited in their ability to play other games or be transferred to different applications.

General game playing is concerned with the design of AI systems that are able to take a description of any arbitrary game at runtime and learn to play the game based solely on the information contained in that description. The goal is to develop game playing systems that can handle games of arbitrary numbers of players, complete or incomplete information, solo or team play and so on.

For this project, the Game Description Language (GDL) developed for the Stanford Logic Group General Game Playing Project (Love *et al.,* 2008) will be used to describe games and the GGP-Base framework released by Sam Schreiber. With GDL, it is possible to describe any game that is finite, discrete, deterministic game of complete information. The focus will be on games with two players but the language can describe games with any number of players.

Game Description Language

In order for agents to be able to correctly perceive game states and parse game rules, there must be some consistent way of representing the games that will be played. The Game Description Language (GDL) developed as part of the General Game Playing project at Stanford [4] and is used in the GGP competition run by the Association for the Advancement of Artificial Intelligence (AAAI) provides a standard for defining games. GDL is a logic-based programming language that can represent deterministic, finite games of complete information as a set of logical propositions. Features such as board layout, legal moves and termination conditions are described as relations between entities. For example, a description of Tic-Tac-Toe could be formulated as:

The game is played by two people, with roles defined as:

```
role(white)
role(black)
```

Base propositions define potential states in the game. In this case, a cell (M,N) can be marked with x, o or b(lank) as long as M and N are valid indices.

```
base(cell(M, N, x) :- index(M) & index(N)
base(cell(M, N, o) :- index(M) & index(N)
base(cell(M, N, b) :- index(M) & index(N)
```

Also, both players alternately take control of the game.

```
base(control(white))
base(control(black))
```

Possible actions are described using the `input` relation. For example, a legal move in Tic-Tac-Toe is:

```
input(R,mark(M,N)) :- role(R) & index(M) & index(N)
```

Which states that an input of `(R, mark(M,N))` is valid id R is a role, and M, N are indices. In addition to being valid, a move must also be legal given the current state:

```
legal(W,mark(X,Y)) :- true(cell(X,Y,b)) & true(control(W))
```

Which tells the agent that move is legal if it the given player has control, X and Y are valid indices and the cell (X,Y) is currently blank.

For a complete description of the language see the language specification (Love *et al*, 2008).

GGP-Base

The publicly available GGP-Base software released by Sam Schreiber (available at https://github.com/ggp-org/ggp-base) provides a framework for developing general game playing agents and provides already implemented functionality such as a game server for starting and managing games, as well as the interface players need to follow to be compatible and sample implementations of simple players. The package also includes an interpreter that will convert the game description to a state machine, an implementation of the state machine and the methods for simulating moves and querying/updating the current state of the game. During game play, the game states are then converted back into GDL and passed to a prover to prove their truthfulness. Where possible the included functionality will be used rather than duplicating work that has already been done and is not core to focus of the project: the performance of the agents themselves.

Games

Checkers

Checkers (or Draughts) is played on an 8x8 board as shown in figure 1 with the starting configuration. Both players begin with 12 pieces, which are all placed on the light coloured squares. A turn consists of either moving a piece one square forward diagonally, or capturing an opponent's piece by moving two consecutive squares, where the opponent's piece occupies the first square and the second is empty. In this manner, multiple pieces may be captured in a single turn. If a piece reaches the opposite end of the board it is 'crowned', gaining the ability to move and capture backwards. Figure 2 shows a game of checkers in progress.

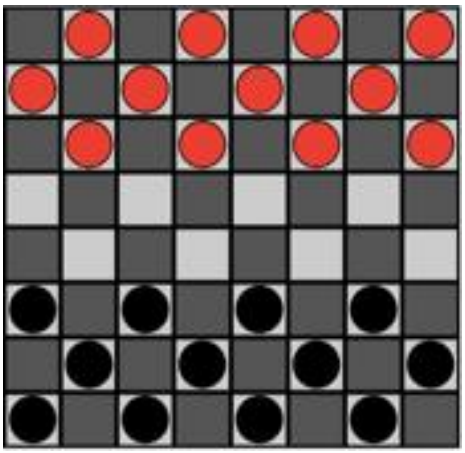

Figure 1. Initial configuration of a checkers board

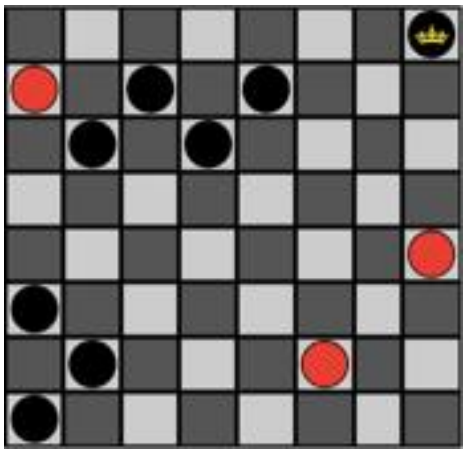

Figure 2. A checkers game in progress

Connect Four

In Connect Four, players take turns to drop a token into one of 8 columns, each 6 rows high. The grid suspended vertically so that tokens occupy the next empty row when dropped. The aim of the game is to construct a horizontal, vertical or diagonal row of 4 tokens. Figure 3 shows a completed game of Connect Four.

Pentago

Pentago is played on a board with 6 columns and 6 rows. These are split into 3x3 quadrants. Players take turns to place a marker in an unoccupied space and then rotate one of the quadrants clockwise or anti-clockwise by 90 degrees. Players win the game by making a row, column or diagonal of 5 markers of their own colour. Figure 4 shows a completed game won by the red player.

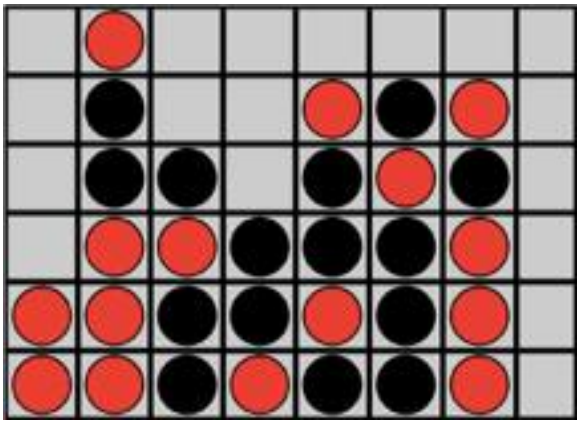

Figure 3. A completed game of Connect-Four with black as the winning player

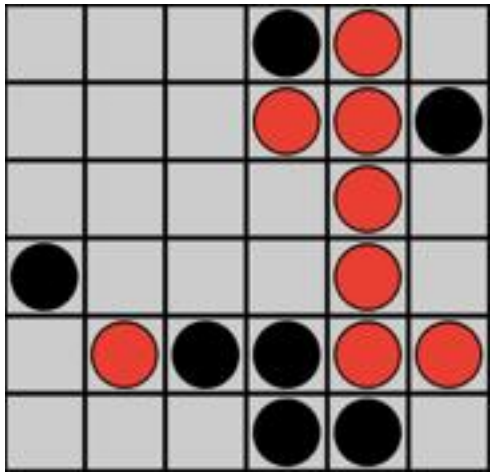

Figure 4. A completed game of Pentago with red as the winning player

Purpose of Project

The purpose of the project is to investigate the performance of different reinforcement learning approaches on a selection of different games under fixed conditions. The project will determine whether one algorithm performs better than the others in general or whether different approaches must be considered depending on the particular game being played.

Aims and Deliverables

Basic:

- Sufficient understanding of GDL to read and use the language
- Configure and utilise GGP-Base software
- Design and configure a basic general game player using Minimax with Monte-Carlo Search

Intermediate:

- Implement Monte-Carlo Tree Search

- Implement Upper Confidence Bound for Trees (UCT) enhancement
- Implement UCT with Decisive Moves enhancement
- Implement All-Moves-As-First (AMAF) enhancement
- Basic testing of general game playing agents

Advanced:

- More advanced testing of general game playing agents
- Use various measures of performance to provide analysis and evaluation of performance of algorithms in different games and against different opponents.
- Discussion of other enhancements to algorithms, game playing agent design
- Design and implement an additional algorithm

II. Design

Requirements

Table 1 shows the functional requirements identified for the program(s) that will be required for the project. FR-1 through FR-6 are handled by the server and player implementations in the GGP-Base package. FR-7 will require some modification to the server code in order to output the information that will be used in the analysis and discussion of testing results.

Table 2 shows the non-functional requirements that arise from the rules that will be applied for testing. Any failure of the agent to return a move through either error or timeout should be considered a failure as per the AAAI GGP competition rules. The moves returned should always be valid and legal for the current state of the game. Since agents are given a fixed time period to choose a move, with no penalty or bonus applied so long as the move is returned within the allotted time, it makes sense that agents should be designed in such a way as to make best use of the time they are given.

The GGP-Base package addresses a number of these requirements and a number of test-games will be played to ensure they are adequately satisfied and the package functions as expected.

Table 1: Functional Requirements

ID	Description	Priority
FR-1	Agents are able to identify legal moves in a given state	High
FR-2	Agents are able to use a reinforcement learning algorithm to select a valid move	High
FR-3	Agents are able to inform other players of move selections	High
FR-4	Agents are able to receive information about the selection of other players moves	High
FR-5	Agents are able to simulate an opponents move	High
FR-6	Agents are able to simulate a game	High
FR-7	Outcome of a game is recorded to a file	High

Table 2: Non-Functional Requirements

ID	Description	Priority
NFR-1	Agents only choose valid, legal moves	High
NFR-2	Agents always return a valid, legal move	High
NFR-3	Agents make maximum use of the turn time where appropriate	High

Choice of Programming Language

The project will be developed using Java. This is because the GGP-Base package is written in Java and it is also a language with which I am very familiar. Java is very portable with wide compatibility and includes a large number of libraries as standard that can be used in development of programs.

An object-oriented approach will be taken, with each agent implementing a common interface and using a different reinforcement learning algorithm. Data structures and other requirements will be implemented with their own classes and methods where applicable in order to promote modularity and extensibility.

Algorithm Design

Algorithm 1: Minimax with Monte Carlo Search (MM-MCS)

This algorithm operates in two phases: the *exploration* phase and the *probe* phase. In exploration, the algorithm expands the tree until some fixed depth is reached. During the probe phase, the algorithm repeatedly selects a leaf node with a uniform random probability and explores this by selecting random successor states until a terminal state is reached. The expected value of the fringe node is the total rewards gained from exploring it divided by the number of times it was visited. Figure 5 Illustrates the operation of the algorithm. In this implementation, recursive exploration and probing are performed as two separate stages to enable maximal use of the time allowed rather than using a fixed number of probes per fringe node. Once a time limit is reached, the algorithm performs a final minimax selection on the tree using values gained from simulation to determine the next move. Algorithm 1 describes the behaviour of MM-MCS.

Algorithm 2: UCT

Monte Carlo Tree Search functions similarly to MM-MCS in that both build a game tree and use random simulation of games to estimate action values. Monte Carlo Tree Search methods differ in that the entire tree is built by a combination of exploration and simulation, rather than having an initial exploration phase. MCTS-based methods have been used in every winning entry to the AAAI GGP competition since its introduction in 2007 (Finsson, 2012). UCT is a popular variant which, given enough time, is proven to converge to Minimax and is therefore optimal (Browne *et al*, 2012). MCTS algorithms operate in four phases: *selection, expansion, simulation, update.* Figure 6 illustrates the operation of the algorithm. Values in the nodes represent wins/visits to the nodes.

Selection: the algorithm selects successor nodes from the current tree according to some policy until it reaches a leaf node. The selection policy should balance exploration of the tree with exploitation of actions that are likely to lead to winning outcomes.

E*xpansion:* If the leaf node selected in the previous phase is not terminal, it is expanded by adding child nodes for the potential moves in the given state.

Simulation: One of the child nodes is selected and explored at random until a terminal state is reached.

Update: The reward at the terminal state is propagated to the visited nodes and the number of visits for each node is incremented by 1.

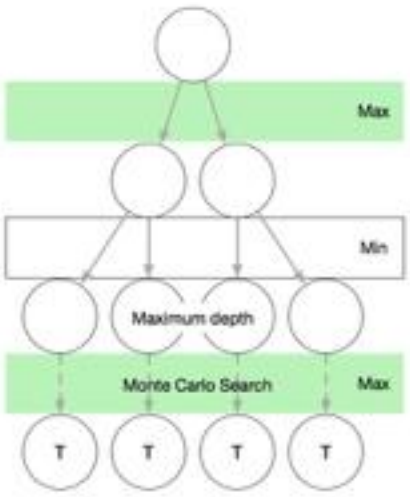

Figure 5. MM-MCS Search tree

Central to the move selection phase is the UCB1 formula derived by Kocsis & Szepesvári (2006). The formula is:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}} \quad (1)$$

Where w_i is the number of wins after the move i, n_i is the number of times the node has been visited and N is the number of times its parent has been visited. c is a variable parameter that controls the bias between exploration and exploitation the UCT bonus). Algorithm 2 is based on the descriptions found in the literature.

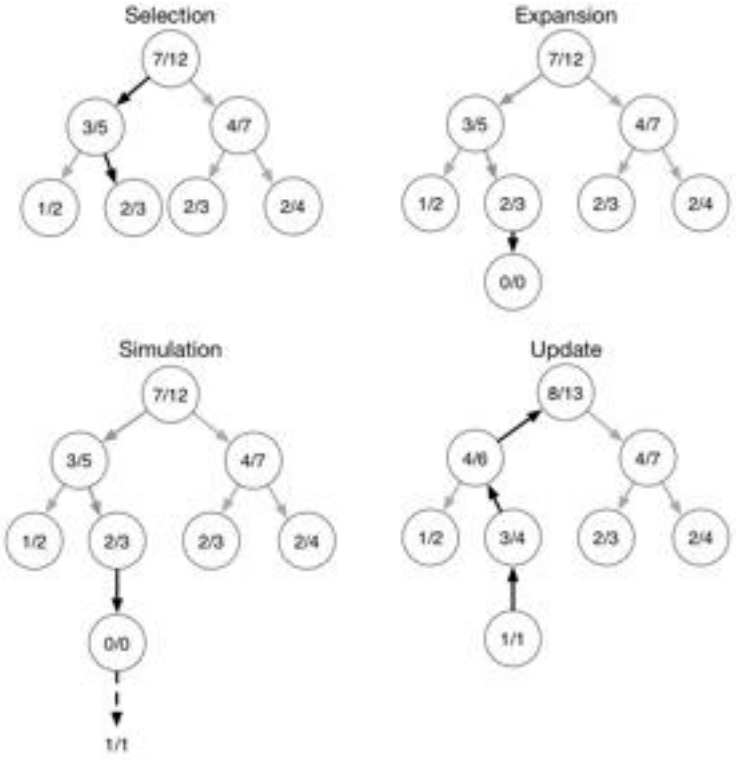

Figure 6. Steps of the MCTS algorithm

Algorithm 3: UCT with Decisive Moves

This modification to the standard UCT algorithm adds an additional check to see whether there is a decisive move (one that leads immediately to a win) or anti-decisive move (one that prevents the opponent from playing a decisive move) in the current state. If a move is found, this

is played otherwise the agent will fall back on the default policy of selection and simulation. Teytaud and Teytaud (2010) demonstrate that this should offer an improvement to performance (whilst taking into account the cost of performing the check).

Algorithm 4: All-Moves-As-First

The All-Moves-As-First approach described in (Helmbold & Parker-Wood, 2009) is an enhancement to the *Selection* phase of Monte Carlo Tree Search. When a simulation is played out, rather than only updating the node at the start of the simulation with the reward, for each move in the simulation the AMAF algorithm updates all nodes of the tree where that move can be played. The approach that will be used here is $\alpha - \text{AMAF}$, where the total score for an action is comprised of the UCT score and AMAF score. The formula is:

$$\alpha A + (1 - \alpha)U \tag{2}$$

Where A is the AMAF score, U is the UCT score and α is some fixed value.This approach extracts more information from each simulation but the downside is that this information may be less relevant in different states.

Testing & Evaluation

The algorithms described above will be evaluated under competition-style conditions. Each agent will play each game a number of times against every other agent to establish the winner. The starting player, number of moves and winning player. Each agent will be subject to the same time limit for choosing moves. Failure to return a move for any reason will result in an automatic loss. For each game, agents will play an equal number of rounds as first player. This will take into account any bias towards the player making the first move. For example, Checkers (Schaeffer *et al*, 2007) and Connect-Four (Allis, 1988) have both been solved, concluding that it is possible for the first player to force a win.

A second round of testing will involve having each player choose moves for a predefined set of scenarios to analyse their behaviour and whether they consistently choose good moves (i.e. block an opponent from winning) or miss obvious winning/losing moves.

There are two elements to the evaluation of the algorithms. First, I will examine the change in performance affected by changing the balance between the randomised monte-carol simulations and performing pre-computation to cut down the game tree. Second, I will attempt to create a relatively strong general game player using various enhancements that should be able to consistently beat the algorithms discussed above.

Algorithm 1 MM-MCS

```
1: procedure MAIN
2:     root ← new tree node
3:     root.state ← getCurrentState()
4:     root.score ← 0
5:     root.visits ← 0
6:     Expand(root, MAX_DEPTH)
7:     while time remaining do
8:         node ← randomly selected leaf node
9:         node.visits + +
10:        result = Simulate(node)
11:        node.score += result
12:    for each leaf node do
13:        node.value = node.score/node.visits
14:    selectedMove = Minimax(root)
15:    return selectedMove
16: function EXPAND(node, depth)
17:    if depth then == 0 return
18:    else
19:        possibleMoves ← getLegalMoves(node.state)
20:        for move in possibleMoves do
21:            child ← new tree node
22:            child.state ← getNextState(node.state, move)
23:            child.visits ← 0
24:            child.score ← 0
25:            node.children ← child
26:            Expand(child, depth − 1)
27: function SIMULATE(state)
28:    if state is terminal then return getGoal(state)
29:    else
30:        possibleMoves ← getLegalMoves(state)
31:        move ← randomly select a move
32:        newState ← getNextState(state, move)
33:        return Simulate(newState)
```

Algorithm 2 UCT

```
1: procedure MAIN
2:     visited ← new linked list
3:     current ← root node
4:     visited ← root node
5:     while current.isLeaf() do
6:         current ←current.select()
7:         visited ← current
8:     if current not terminal then
9:         Expand(current)
10:        newNode ← Select(current)
11:        reward ← Simulate(newNode.state)
12:    else
13:        reward ← get reward for current
14:        Update(node,reward)
15:    for node in visited do
16:        Update(reward)
17: function EXPAND(node)
18:    possibleMoves ← getLegalMoves(node.state)
19:    children ← new linked list
20:    for move in possibleMoves do
21:        child.state ← getNextState(current, move)
22:        child.wins ← 0
23:        child.visits ← 0
24:        children ← child
25: function SELECT(node)
26:    selected ← null
27:    best ← Double.minValue
28:    for child in node.children do
29:        uct ← child.wins/child.visits + c√(ln visits/child.visits)
30:        if uct > best then
31:            selected ← child
32:            best ← uct
33:    return selected
34: function SIMULATE(theState)
35:    if isTerminal(node) then
36:        return getGoal(theState)
37:    else
38:        possibleMoves ← getLegalMoves(theState)
39:        randomMove ← random move from possibleMoves
40:        newState ← getNextState(theState, randomMove)
41:        return Simulate(newState)
42: procedure UPDATE(node,value)
43:    node.visits ++
44:    if value == 100 then wins ++
```

References

Allis, V. (1988). A Knowledge-based Approach of Connect-Four, M.Sc. ThesisVrije Universiteit Report No. IR-163, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. IEEE Trans. Comput. Intell. AI Games, 4(1), pp.1-43.

Finsson, H. (2012). Generalized Monte-Carlo Tree Search Extensions. In: Proceedings of the Twenty Sixth AAAI Conference on Artificial Intelligence. AAAI, pp.1550-1556.

Helmbold, D. P., Parker-Wood, A. (2009). "All-Moves-As-First Heuristics in Monte-Carlo Go," in Proc. Int. Conf. Artif. Intell., Las Vegas, Nevada, pp. 605–610.

Kocsis, L., and Szepesva´ri, C. (2006). Bandit based Monte- Carlo planning. In European Conference on Machine Learning (ECML), 282–293.

Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M. (2008). General Game Playing: Game Description Language Specification. (online) Accessed 17.12.2014

Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P. and Sutphen, S. (2007). Checkers Is Solved. Science, 317(5844), pp.1518-1522.

Tesauro, G., Rajan V. T., and Segal. R. B. (2010). "Bayesian Inference in Monte-Carlo Tree Search," in Proc. Conf. Uncert. Artif. Intell., Catalina Island, California, pp. 580–588.

Teytaud, F., Teytaud, O. (2010). "On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms," in Proc. IEEE Symp. Comput. Intell. Games, no. 1, Dublin, Ireland, pp. 359–364.