Your Project Report Title Here
CS7IS2 Project (2019/2020)

Aishwarya Ravindran, Jagadish Ramamurthy, Manasi Mohan Nasrapur and Mukesh Arambakam

[1] ravindra@tcd.ie, ramamurj@tcd.ie, narsapum@tcd.ie
[2] arambakm@tcd.ie

Abstract. Connect-Four is a familiar and well-known board game whose objective is to get four slots in a row. This paper demonstrates the creation, testing and performance evaluation of the game with the help of three state-of-the art AI algorithms namely Minimax , Expectimax with alpha-beta pruning and Depth-Limited Search for improved performance and Monte Carlo Tree Search . Performance test was carried out over a series of matches from which win percentage of every agent was recorded. The evaluation of these outcomes concluded in Minimax at depth 6 with alpha-beta pruning and depth-limited search as the algorithm with highest win ratios among all these agents.

1 Introduction

Besides constructing intelligent systems, Artificial Intelligence (AI) is also the endeavour to understand them. It includes applications ranging from general purpose areas to particular tasks like proving mathematical theorems and diagnosing diseases. General Game Playing (GGP) is the structure of AI programs' ability to excel at game playing which is one of the oldest fields of ventures in AI.

Connect-Four, a two-player game where the objective of the game is to win by making four discs belonging to a player line horizontally, vertically or diagonally. The board is fashioned with six rows and seven columns where the discs are dropped vertically down occupying the last available slot within that respective column by taking turns. If the board is completely filled with discs but was unsuccessful in deciding a winner, then it ends with a tie. Although it is a solved game, it has 10^{13} possible positions on the board making it absurd and illogical to store the moves in memory. As ardent players of board games and an inclination towards working on Artificial Intelligence, in this paper, we aim to depict and illustrate the different ways AI can be utilized to create gameplay agents for the Connect-Four board game. Agents in AI are entities which act and direct movements towards accomplishing goals. However, goals alone are insufficient to develop high-quality behaviour. Utility based agents comprise of a utility function that maps a state onto a real number representing the associated degree of happiness[1]. For the agent to achieve the goal, it needs to be equipped with the capability of knowing the states, possible actions, transition model, goal test and path cost. The game's environment[3] comprises of *Accessible*, *Deterministic*, *Static* and *Discrete* environmental properties.

There are many approaches that can be employed to solve the Connect-Four game based on the difficulty level set for the AI. AI are of random, defensive and aggressive type[2]. Random AI as the name suggests randomly performs an action and is easiest to beat, Defensive AI prioritizes to oppose and block the opponents moves from winning and Aggressive AI gives precedence to only winning.

Connect-Four makes a good candidate for implementing and studying search algorithms to find the best solution from all feasible solutions. This paper demonstrates the creation, testing and performance evaluation of Connect-Four with the help of three state-of-the art AI algorithms namely Minimax algorithm, Expectimax Algorithm with alpha-beta pruning and Depth-Limited Search for improved performance and Monte Carlo Tree Search algorithm. In this paper, we aim to demonstrate a variation to the program with a choice to select the type of players where possible combinations of players bring Human versus Human, Human versus Agent and Agent versus Agent. Our experiment involves five types of agent-players namely: *Agent1* – Random, *Agent2* – Forward checking, *Agent3* – Minimax algorithm with alpha-beta pruning and depth-limited search, *Agent4* – Expectimax algorithm with alpha-beta pruning and depth-limited search and *Agent5* – Monte Carlo Tree Search.

Search strategies are evaluated by measuring their effectiveness of performance in solving the problem. Ideally, effectiveness of a search is calculated using the search cost associated with the time and memory consumed to obtain the solution. For the Connect-Four game, we aim to calculate win ratios of the players.

The rest of the paper is organized as follows: Section 2 mentions the Related Work on different techniques and approaches, Section 3 comprises of the Problem Definition and Algorithms implemented, Section 4 describes the Experimental Results obtained by the utilised methodology and lastly, Section 5 includes the Conclusions and final discussions of the main results along with an acknowledgement of future work.

2 Related Work

Game playing is an important area of artificial intelligence since they are an interesting means in comparing interaction between user and machine's behaviour directly. Once the rules, permissible moves and conditions to succeed or lose the game are acknowledged search procedures can be implemented to benefit exploring best moves. Game theory broadens decision theory to circumstances in which multiple agents interact. There has been extensive research performed in understanding different approaches to solving two-player games, strategical and turn-taking games and tactical adversarial games like Backgammon, Connect-Four and Chess. Typically, these games are modelled as decision trees using different AI algorithms. The general notion of characterizing and assessing the quality of a game is by playing the game with various controllers and algorithms and evaluating the performance through comparisons. By investigating the relative performance of different general game-playing algorithms, results suggest a significant positive relationship between intelligent agents, game design and success rate of the algorithm[6]. In fact, these parameters differentiate between good and bad game playing algorithms on a well-designed game with the restriction to human-designed games.

Comparative analysis conducted on different AI based machine learning techniques like Naïve Bayes classifier and support vector machines to evaluate the performance issues faced trying to achieve optimality[5] was helpful and was utilized in designing the characters of video games. Performance evaluation through a series of matches against a manually designed static AI, in a turn-based strategic game which uses dynamic scripting to provide actions was illustrated by a paper on turn based strategy games[9].

Effectiveness on games like Connect-Four with the objective to relate their performance to the strategies of the domains was explored by a paper on MCTS-minimax hybrids[8] that integrate shallow minimax searches into the MCTS framework. The paper investigated ways to combine the strategic toughness of MCTS with the tactical strength of minimax to exercise better results from universally useful hybrid search algorithms. Connect-Four game in a real-time environment with incorporation of time restraints[2] was studied by another paper which used an artificial intelligence based on influence mapping, minimax, minimax with alpha-beta pruning and A* to evaluate its performance with time constraints.

The researches discussed above served as an inspiration for our work. We approach the Connect 4 game through three different algorithms to analyse the performances and identify the best among them since there is a relationship between the results and agent incorporated with types of algorithms[6]. Performance was evaluated through series of matches[9] and MMTO with alpha-beta pruning was carried out similar to the technique carried out by a paper with the game of shogi[7].

3 Problem Definition and Algorithm

Connect-Four being a two-player mathematical 6x7 board game, can be solved strategically using a mathematical solution. Fig 1 depicts the factors that are to be defined to formulate the problem.

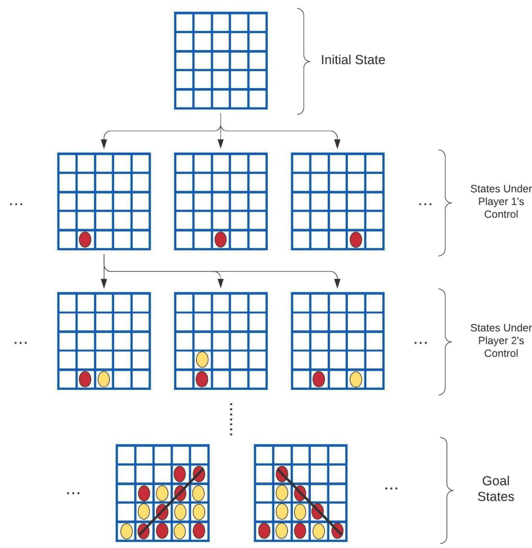

Fig. 1. Connect 4 - State Action

From a plethora of artificial intelligence algorithms available, a mixture of brute-force techniques and knowledge-based approaches can be utilised to solve the game. There are a few conclusions resolved from solving the game, for instance, the first player has a higher probability of winning if the game is started by placing the disc in the middle column. However, this highly depends on the agent's capacity to look ahead based on the algorithm imposed on it. The layout of the Connect-Four game and design flow of the game is shown in Fig 2 and Fig 3 respectively.

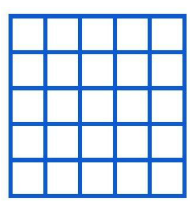

Fig. 2. Connect 4 - Empty Board

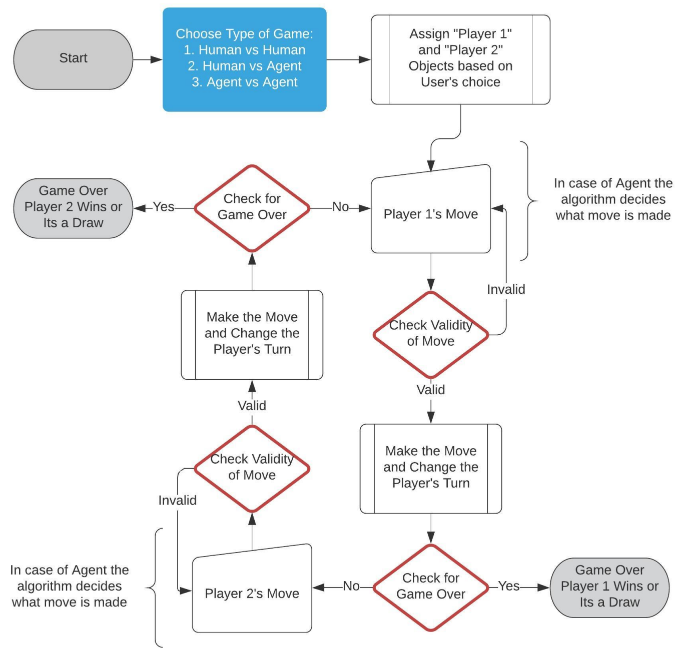

Fig. 3. Game Flow

3.1 Minimax Algorithm

Minimax, a decision-making algorithm is typically used in turn-based two-player games aiming to obtain the optimal next move. The general terms of the algorithm are maximiser, generally being the agent and minimizer, the agent's opponent. The algorithm is based on a zero-sum game theory concept where "the total utility score is divided among the players. An increase in one player's score results in decrease in another player's score"[10]. This recursive algorithm prepares an optimal move for the agent when always assuming that the opponent plays optimally. To achieve this, the agents looks ahead by using depth-first search for all feasible moves and calculates the resultant score. However, it is not feasible to explore the entire game tree for Connect-Four and hence we opt to backtrack the score using depth-limited search in place of depth-first search which imposes a cut off on the maximum depth of a path. In this paper, we look ahead by 5-6 moves to determine an intermediate score in the "evaluation" function for which the pseudocode is depicted below.

```
def evaluate_window(window):
    score = 0
    if window.count(agents_coin) == 4:
      score += 100
```

```
    elif window.count(agents_coin) == 3 and window.count(
                                EMPTY) == 1:
      score += 5
    elif window.count(agents_coin) == 2 and window.count(
                                EMPTY) == 2:
      score += 2

    if window.count(opponent_coin) == 3 and window.count(
                                EMPTY) == 1:
      score -= 4

    return score

def evaluate(state)
    score = 0
    ## Score center column
    score += state.center_column.count(agents_coin) * 3

    ## Score Horizontal
    for window in state.all_possible_horizontal_windows:
      score += evaluate_window(window)

    ## Score Vertical
    for window in state.all_possible_horizontal_windows:
      score += evaluate_window(window)

    ## Score positive sloped Diagonal
    for window in state.all_possible_pos_sloped_windows:
      score += evaluate_window(window)

    ## Score negative sloped Diagonal
    for window in state.all_possible_neg_solved_windows:
      score += evaluate_window(window)

    return score
```

Pruning is a technique implemented in search algorithms to reduce the length of the tree by eliminating the sections that are significantly negligent in terms of accuracy or efficiency. Therefore, we apply a pruning method to limit the number of nodes that the agent needs to examine when looking ahead. Alpha-beta pruning, a technique that can be applied to any depth of a tree is implemented on the standard minimax algorithm to obtain optimal moves. Below code depicts the pseudocode for the minimax algorithm.

```
  def minimax(state, depth, alpha, beta, maximizingPlayer):
    valid_moves = state.get_valid_moves()
    is_terminal = state.is_terminal_node()

    if depth == 0 or is_terminal:
```

```
    if is_terminal:
      if state.check_win(agent):
        return (None, 100000)
      elif state.check_win(opponent):
        return (None, -100000)
      else: # Game is over, no more valid moves
        return (None, 0)
    else: # Depth is zero
      return (None, evaluate(state))

  if maximizingPlayer:
    utility = -math.inf
    best_move = random.choice(valid_moves)
    for move in valid_moves:
      state.agent_makes_move(move)
      new_utility = minimax(state, depth-1, alpha, beta,
                            False)[1]

      if new_utility > utility:
        utility = new_utility
        best_move = move

      alpha = max(alpha, utility)
      if alpha >= beta:
        break
    return best_move, utility

  else: # Minimizing player
    utility = math.inf
    best_move = random.choice(valid_moves)
    for move in valid_moves:
      state.simulate_opponent_move(move)
      new_utility = minimax(state, depth-1, alpha, beta,
                            True)[1]

      if new_utility < utility:
        utility = new_utility
        best_move = move

      beta = min(beta, utility)
      if alpha >= beta:
        break
    return best_move, utility
```

3.2 Expectimax Algorithm

There are games that mirror the unpredictable external events by including a random element. Expectimax, a variant of minimax is an algorithm commonly used in zero-sum games where the outcome depends on the probability of the external element along with the player's skill. It generalises the games where transitions between states are made probabilistic. It uses a predefined opponent strategy to treat opponent decision nodes as chance nodes[1]. In other words, the opponent generally being the minimizer in minimax is replaced with the role of a player whose moves depend on the probability. These are no longer definite with a minimax value and calculated as an average or expected value. Below code depicts the pseudocode for the minimax algorithm.

```
utility = math.inf
      best_move = random.choice(valid_moves)
      for move in valid_moves:
        state.simulate_opponent_move(move)
        new_utility = expectimax(state, depth-1, alpha, beta,
                                  True)[1]

        if new_utility <= utility:
          utility = new_utility
          best_move = move

        beta = utility/no_of_valid_moves
        if alpha >= beta:
          break
      return best_move, utility
```

Apart from the probabilistic element, the algorithm works similar to minimax and hence we apply the same depth-limited search and alpha-beta pruning technique to improve the performance of the algorithm.

3.3 Monte Carlo Tree Search Algorithm

Monte Carlo Tree Search (MCTS) is a deeply utilized heuristic search algorithm known for incorporating an element of randomness into the search for variety of games. It was highly recommended as a search and planning structure for identifying optimal results in a given situation. This is demonstrated and confirmed because Monto Carlo rollouts of games gives it a strategic advantage over traditional depth-limited searches. Upper Confidence Bounds for Trees (UCT), a manifestation of MCTS was considered revolutionary for game-playing agents in artificial intelligence[4]. The pseudocode and expression for UCT is depicted below from which the highest value calculated is utilised in decision making models.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \tag{1}$$

Where,

– wi : number of wins for the node after the ith move
– ni : number of simulations for the node after the ith move
– Ni : total number of simulations after the ith move run by the parent node
– c : the exploration parameter, chosen as $\sqrt{2}$ for the game

```
# returns child with largest UCT value
def selection():
uct_value = lambda node: node.wins / node.visits + sqrt(2 *
                              log(parent_node.visits) / node
                              .visits)
# applies the above uct_value lamba function to all the child
                               nodes (children)
return sorted(children, key = uct_value)[-1]
```

The MCTS planning methodology incrementally constructs an asymmetric search tree guided in most assuring direction which is estimated iteratively and enhances its performance with an increase in the number of iterations ie., it samples moves rather than considering all legal moves from a given state[4]. An MCTS iteration usually consists of four consecutive phases: (1)*Selection* where the tree choses the node with largest UCT value, in other words the node which possess the best chance of winning out of all, (2)*Expansion* into new child nodes of the tree after randomly choosing the node, (3)*Rollout* where simulations are conducted on the expanded nodes to review the status of the game - win, loss or tie, (4)*Backpropagation* updates the results obtained in the rollout step along the chosen path like number of wins for the selected node, total number of simulations conducted on the selected node and parent node.

```
def montecarlo_tree_search(currentNode, state):
        rootnode = Node(state, PREV_PLAYER)

        if currentNode is not None:
            rootnode = currentNode

        for i in range(max_iterations):
            node = rootnode

            # selection
            while node.available_moves == [] and node.
                                        children != []:
                node = node.selection()
                player_makes_move(state, CURR_PLAYER, node.
                                            move)

            # expand
            if node.available_moves != []:
                move = random.choice(node.available_moves)
                player_makes_move(state, CURR_PLAYER, move)
                node = node.expand(move)
```

```
            # rollout
            while get_available_moves():
                move = random.choice(get_available_moves())
                player_makes_move(state, CURR_PLAYER, move)
                if check_player_win(PREV_PLAYER):
                    break

            # backpropagate
            while node is not None:
                node = node.parent

            if timeout:
                break

        win_ratio = lambda node: node.wins/node.visits
        sorted_children = sorted(rootnode.children, key =
                                  win_ratio)[::-1]

        return rootnode, sorted_children[0].move
```

4 Experimental Results

In this section, the experimental setup, evaluation criteria for the data and methodology used is described in detail followed by a discussion of findings and results.

4.1 Methodology

Apart from the three state-of-the-art artificial intelligence algorithms, our experimental setup consists of a random and forward checking agent for naïve playing. We aim to formulate different combinations of games to be played against each other[9] and evaluate their performance through those series of five matches. Based on the time and number of moves made by each of the players, win percentages are calculated over 5 matches inclusive of games that ended in a tie.

4.2 Results

To begin the experiment, a variation of minimax and expectimax with depth as 3 and 4 were set to play among all combinations over a series of 5 matches. A detailed report of total number of moves made and total time taken by both players and the winners of each game were recorded. From this data, the win percentages of each agent against their opponents were recorded as depicted by Fig 4.

% OF WINS BETWEEN PLAYER 1 (in red) AGAINST PLAYER 2 (in yellow)	Random	Forward Checking	Minimax (D=3)	Minimax (D=4)	Expectimax (D=3)	Expectimax (D=4)	MCTS (S = 5k, T = 2s)
Random	60 / 40	0 / 100	0 / 100	0 / 100	0 / 100	0 / 100	0 / 100
Forward Checking	80 / 20	20 / 80	0 / 100	0 / 100	20 / 80	20 / 40	0 / 100
Minimax (D=3)	100 / 0	100 / 0	100 / 0	100 / 0	100 / 0	100 / 0	60 / 40
Minimax (D=4)	100 / 0	100 / 0	0 / 100	100 / 0	100 / 0	100 / 0	40 / 60
Expectimax (D=3)	100 / 0	20 / 80	0 / 100	0 / 100	100 / 0	100 / 0	40 / 60
Expectimax (D=4)	100 / 0	60 / 40	0 / 100	0 / 100	100 / 0	100 / 0	60 / 40
MCTS (S = 5k, T = 2s)	100 / 0	100 / 0	0 / 100	20 / 80	40 / 60	20 / 80	100 / 0

Fig. 4. Win Ratio for Bots 2

Briefly, it is noticeable that the random agent has almost negligible wins against forward checking and the other agents. Based on manual analysis of these results thoroughly, it was apparent that the random and forward checking agent had insignificant and almost negligible win ratios against the agents equipped with Minimax, Expectimax and MCTS algorithms. This is a reflection of how good the agents equipped with algorithms have better win ratios i.e. they have a better chance of solving and winning the game against a random agent and naïve-play by the forward checking agent. Therefore, we resolved to exclude the random and forward checking agent from the next part of the experimental study.

The study was advanced by performing a series of another 5 matches with depth values as 5 and 6 for both minimax and expectimax and varying the number of iterations S as 5k and 10k with Time-out time T as 2s and 3s for the MCTS agent. A similar report of total number of moves made and total time taken by both players and the winners of each game were recorded from which the win percentages of each agents against their opponents was captured. Fig 5 depicts the recordings.

% OF WINS BETWEEN PLAYER 1 (in red) AGAINST PLAYER 2 (in yellow)	Minimax (D=5)	Minimax (D=6)	Expectimax (D=5)	Expectimax (D=6)	MCTS (S = 10k, T = 3s)	MCTS (S = 5k, T = 3s)	MCTS (S = 10k, T = 2s)
Minimax (D=5)	100 / 0	0 / 100	100 / 0	0 / 0	60 / 40	80 / 20	80 / 20
Minimax (D=6)	100 / 0	100 / 0	100 / 0	0 / 0	80 / 20	80 / 20	100 / 0
Expectimax (D=5)	0 / 100	0 / 100	0 / 100	0 / 100	60 / 40	40 / 60	80 / 20
Expectimax (D=6)	0 / 100	0 / 100	0 / 100	0 / 100	40 / 60	40 / 60	80 / 20
MCTS (S = 10k, T = 3s)	20 / 80	20 / 80	40 / 60	20 / 80	40 / 60	60 / 40	60 / 40
MCTS (S = 10k, T = 2s)	20 / 80	20 / 80	40 / 60	60 / 20	60 / 40	40 / 60	60 / 40
MCTS (S = 5k, T = 3s)	0 / 100	0 / 100	60 / 40	20 / 60	60 / 40	40 / 60	80 / 20

Fig. 5. Win Ratio for Bots 1

4.3 Discussion

In order to determine the performance of each of the agents, they were made to contest with each other and calculate certain metrics for evaluation. This is conducted based on the fact that win rates suggest a relationship between intelligent agents based on the different types of algorithms that they are equipped with. On analysis of the outcomes, it is observed that agents equipped with Expectimax and Monte Carlo Tree Search do substantially better than the Random and Forward-Checking agent. MCTS sampling is said to be robust and produce strong plays in contrast to minimax which is fragile towards noise in the evaluate function for the intermediate states. Even though MCTS has displayed reasonable success a primary weakness of MCTS, shared by most search heuristics, is that the dynamics of search are not yet fully understood, and the impact of decisions concerning parameter settings and enhancements to basic algorithms are hard to predict[11]. Therefore, it appears that the traditional approach of Minimax algorithm at depth 6 with alpha-beta pruning and depth-limited search has the highest number of wins among all the agents overall. This is because minimax always provides the optimal move to the agent under the impression that the opponent also makes optimal moves. The ability of our program to look ahead by 5-6 moves and resolve an intermediate score is beneficial and can be witnessed in our results.

5 Conclusions

In this paper, we illustrated the creation, testing and performance evaluation of Connect-Four by implementing three state-of-the art AI algorithms namely Minimax, Expectimax with alpha-beta pruning and Depth-Limited Search for improved performance and Monte Carlo Tree Search. According to our observations, Minimax at depth 6 with alpha-beta pruning and depth-limited search had the highest win ratios among all the agents. This was due to the program's ability to look ahead and Minimax's inherent performance of providing optimal moves to the agent. This makes minimax with pruning and search techniques probably most useful in similar board games. In the future, it could be interesting to consider using advanced algorithms like Reinforcement Learning and AlphaGo to solve the game and analyse its performance. To conclude, we would like to acknowledge **Keith Galli** ⟨ @ *KeithGalli* ⟩ for the git repository Connect4-Python from which the base code for the Connect-Four game was retrieved.

References

1. Russell, Stuart, and Peter Norvig. "Artificial intelligence: a modern approach." (2002)
2. Sarhan, Ahmad M., Adnan Shaout, and Michele Shock. "Real-Time Connect 4 Game Using Artificial Intelligence 1." (2009).
3. 0xADADA. 2003. "A Connect Four Playing AI Agent: Algorithm and Creation Process." December 15, 2003. https://0xadada.pub/2003/12/15/connect-four-playing-ai-agent/.
4. TVodopivec, Tom, Spyridon Samothrakis, and Branko Ster. "On Monte Carlo tree search and reinforcement learning." Journal of Artificial Intelligence Research 60 (2017): 881-936.
5. Nawalagatti, Amitvikram, And Prakash R. Kolhe. "A Comprehensive Review On Artificial Intelligence Based Machine Learning Techniques For Designing Interactive Characters.
6. Nielsen, Thorbjørn S., Gabriella AB Barros, Julian Togelius, and Mark J. Nelson. "General video game evaluation using relative algorithm performance profiles." In European Conference on the Applications of Evolutionary Computation, pp. 369-380. Springer, Cham, 2015.
7. Hoki, Kunihito, and Tomoyuki Kaneko. "Large-scale optimization for evaluation functions with minimax search." Journal of Artificial Intelligence Research 49 (2014): 527-568.
8. Baier, Hendrik, and Mark HM Winands. "MCTS-minimax hybrids." IEEE Transactions on Computational Intelligence and AI in Games 7, no. 2 (2014): 167-179.
9. Santoso, Sulaeman, and Iping Supriana. "Minimax guided reinforcement learning for turn-based strategy games." In 2014 2nd International Conference on Information and Communication Technology (ICoICT), pp. 217-220. IEEE, 2014.
10. Wikipedia Contributors. 2020. "Zero-Sum Game." Wikipedia. Wikimedia Foundation. March 19, 2020. https://en.wikipedia.org/wiki/Zero-sumgame.
11. Browne, Cameron B., Edward 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 Transactions on Computational Intelligence and AI in games 4, no. 1 (2012): 1-43.