CHALMERS TEKNISKA HÖGSKOLA

DESIGN OF AI SYSTEMS (DAT410)

Module 6: Game Playing Systems

Group 38

Himanshu Sahni
9812075498
sahni@chalmers.se
MPCAS

Nina Uljanić
9609134920
uljanic@chalmers.se
MPCAS

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part in other solutions.

Hours:
Himanshu Sahni: 15 h
Nina Uljanić: 25h

1. Reading and reflection

Himanshu Sahni

"Mastering the Game of Go with Deep Neural Networks and Tree Search" describes the development of a computer program, called AlphaGo, that was able to defeat a human world champion in the game of Go. The game of GO is classified as the most challenging for AI because of its large search space and the difficulty to evaluate board positions and moves. All games have an optimal function value that determines the outcome of that game, for GO the possible sequence of moves is b^d, where b is the number of legal moves and d is the game length. Usually b≈250 and d≈150. However, the search space can be reduced. Depth of search may be reduced by position evaluation, replacing subtree states s with an approximate value function that predicts the outcome from state s. The breadth could be reduced using Monte Carlo simulation from a policy p(a|s), which is a probability distribution over possible moves 'a' in position 's'. The strongest GO programs are based on Monte Carlo Tree Search and enhanced by policies that are trained to predict human expert moves. Neural Networks are used to reduce the effective breadth and depth of the search tree, evaluating positions by using a value network and sampling actions using a policy network. The pipeline of the neural network is: p_σ-> p_π -> p_p -> v_θ where,
p_σ are the lessons learned from human experts
p_π samples actions rapidly
p_p improves SL by optimizing the final outcome
v_θ predicts the winner of the game by RL policy network against itself.
Supervised Learning of Policy Networks: The SL policy Network alternates between the convolutional layers with weight σ and rectifier nonlinearities. Finally, a softmax layer outputs a probability distribution over all legal moves α.
Reinforcement Learning of Policy Networks: Initially we set $p_p = p_\sigma$, then play with random opponents to prevent overfitting. The reward functions are zero for all non-terminal state's time steps and terminal reward of +1 if win and -1 if lose.
Reinforcement Learning of Value Networks: It focuses on position evaluation using a value function that predicts the outcome from position 's' of games played by using policy 'p' for both players.
Alpha Go combines the policy and value networks in an MCTS algorithm that selects action by lookahead search. Using action value Q(s, a), visit count N(s, a), and prior probability p(s, a) and simulating the tree in complete games by descending without backup. At each time step t of each simulation, an action a_t is selected from state s_t

$$a_t = argmax(Q(s,a) + u(s_t, a)) \tag{1}$$

so as to maximize action value plus a bonus

$$u(s,a) \propto \frac{P(s,a)}{1 + N(s,a)} \tag{2}$$

When the traversal reaches a leaf node s_L at step L, the leaf node may be expanded. The leaf position s_L is processed just once by the SL policy network P_σ. The output probabilities are stored as prior probabilities P for each legal action a, P(s,a) = P_σ(a|s). The leaf node is evaluated in two very different ways: first, by the value network v_θ(sL); and second, by the outcome z_L of a random rollout played out until terminal step T using the fast rollout policy p_π; these evaluations are combined, using a mixing parameter λ, into a leaf evaluation V($_{sL}$)

$$V(_{sL}) = (1-\lambda)v_\theta(sL) + \lambda z_L \tag{3}$$

At the end of the simulation, the action values and visit counts of all traversed edges are updated. Each edge accumulates the visit count and means evaluation of all simulations passing through that edge

$$N(s,a) = \sum_{n=1}^{n} l(s,a,i) \tag{4}$$

.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{n=1}^{n} l(s,a,i) V(s^i L) \tag{5}$$

where s$^i{}_l$ is the leaf node from the i$^t h$ simulation, and 1(s, a, i) indicates whether an edge (s, a) was traversed during the i$^t h$ simulation. Once the search is complete, the algorithm chooses the most visited move from the root position.

Nina Uljanić

The paper describes the implementation of AlphaGo, a Go-playing AI agent. It introduces the concept of games of perfect information, of which Go is a part, and explains that for these games an optimal value function exists that can determine the game's outcome based on the current state of the game. These games can be solved by recursively computing the optimal value function in a search tree, where the nodes represent the possible moves. Given a game with b legal moves and game length d, its tree would contain approximately b^d possible (sequences of) moves. For the game Go, where $b = 250$ and $d = 150$, this means that the search tree would be gigantic. This is not feasible. There exists a solution to this problem: the search space can be reduced by reducing the depth and breadth of the search using Monte Carlo Tree Search. MCTS is an iterative algorithm that evaluates one node at a time using simulation, i.e. it attempts to predict the outcome of the given state. The policy used to select actions is improved over time, and, after many iterations, it converges to the optimal value function.

AlphaGo combines MCTS and deep neural networks. The training pipeline is made up of three stages: supervised learning of policy networks for predicting moves, reinforcement learning of policy networks for improving the policy network (maximize win, not prediction accuracy), and reinforcement learning of value networks for position evaluation (prediction). These are combined in an MCTS algorithm, which can be summarised as follows.

Each board state is set as the root of the tree. At each time step t, an action is selected from the state. This is equivalent to selecting a child node. The selection rule is such that the node with the most visits is selected. In the case of unexplored nodes, the leaf node is expanded, i.e. an unvisited child node is selected. The game is simulated for the selected node, and the action values and visit counts are recorded. All parent nodes are updated accordingly. After a set number of iterations, the algorithm chooses the most visited child node of the root. This move is played next by the AI.

I have found it particularly interesting to read about AlphaGo's usage of CPU multithreading, parallel computations on GPUs, and distribution over multiple machines.

2. Implementation

For this task, we chose to implement a Monte Carlo Tree Search tic-tac-toe. In our implementation, the human player starts first. After the human has selected a move, it is the computer's turn. For each computer step, we run the Monte Carlo Tree Search algorithm. This is done by creating a new instance of the MCTS object (tree) for each step, initializing the tree with the current state as the root node. It is from this state that we branch out and try to find the next best move using the four steps of the MCTS algorithm. They are implemented as follows.

In selection, we check whether the current node has any unexplored children nodes (available actions). If yes, we move to step 2 in order to expand. Else, if all children nodes have been visited at least once, we select the child node with the highest UCT value, that is, we assign it as "current", and move to step 3.

In expansion, we create a child node from the state selected in step 1, and move to step 3.

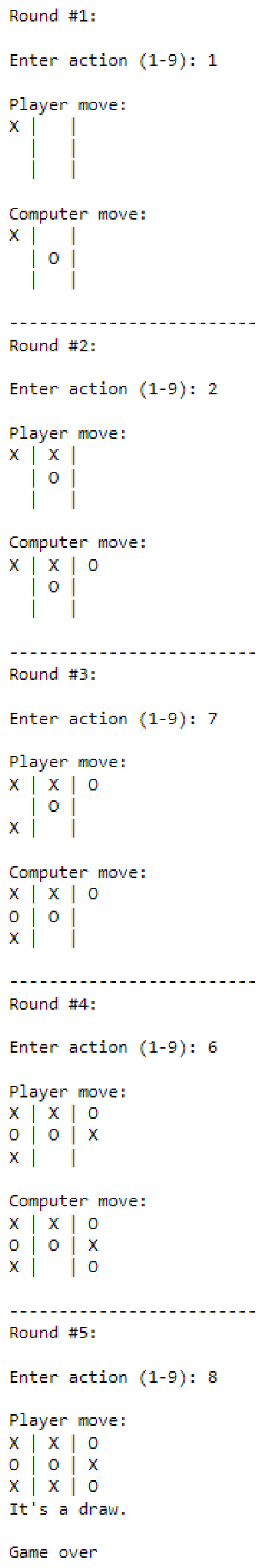
In simulation, we generate an end state for the selected state. That is, we randomly assign 'X's and 'O's to the available slots, making sure to alternate players. This is done by fetching indices of available slots in a list, shuffling this list to simulate random plays, and filling the slots accordingly. The simulated state is then evaluated, and we move to step 4.

In backpropagation, the selected node is updated. If the node has a parent, its UCT value is updated - only nodes with parents have a UCT value because the UCT function ($UCT(v_i) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$) relies on the number of visits of the parent. The update function is then recursively called on the parent node so that all preceding nodes are updated as well, all the way to the root of the tree.

After a set number of iterations, the result of the search can be extracted. The computer plays the action represented by the root's child which has the highest number of visits. See Fig. 1 below for an example game output. The program can be summarised as follows:

1. Take user input.
2. Generate the tree with the current state as root.
3. Run the MCTS algorithm for a set number of iterations.
4. Make a move according to the result of the MCTS algorithm.
5. Repeat until the game has finished, i.e. until either player has won, or it's drawn.

The idea behind the algorithm is impressive, however we have not found it to be particularly efficient in our implementation. We originally tried using tree search for the simulation step in order to get the best evaluation possible, but we soon found out that we do not posses the computational resources needed for such an extensive search. The games do not always end in a draw; the computer sometimes plays moves that are obviously "bad." This is most likely due to the randomness of the search, and the number of iterations for which the algorithm runs. Besides that, the algorithm runs fairly quickly, depending on the number of iterations. If applied to larger grids, the search would have to be more extensive, consequently requiring more computational power.



```
Round #1:

Enter action (1-9): 1

Player move:
X |   |
  |   |
  |   |

Computer move:
X |   |
  | O |
  |   |

-------------------------
Round #2:

Enter action (1-9): 2

Player move:
X | X |
  | O |
  |   |

Computer move:
X | X | O
  | O |
  |   |

-------------------------
Round #3:

Enter action (1-9): 7

Player move:
X | X | O
  | O |
X |   |

Computer move:
X | X | O
O | O |
X |   |

-------------------------
Round #4:

Enter action (1-9): 6

Player move:
X | X | O
O | O | X
X |   |

Computer move:
X | X | O
O | O | X
X |   | O

-------------------------
Round #5:

Enter action (1-9): 8

Player move:
X | X | O
O | O | X
X | X | O
It's a draw.

Game over
```

Figure 1: Output from the MCTS tic-tac-toe game

Individual Summary and Reflection

Lecture 6: Game playing systems (2023-02-21)

Lecture summary by Himanshu Sahni

Games are about taking action and these actions determine our future. The aim is to take actions that can improve our chances of winning and we take into account both good and bad futures. The branching paths say that the best path must be picked. Search trees formalize the structure to be followed to check the possible futures and by identifying good futures (wins), we can note the actions that we have to take to win. This can be used to set the path to victory. We should evaluate a state to validate victory or defeat. However, the action of the opponent can also change the outcome. For every move and position, we must compute the probabilities and then take an action with the highest possible success rate. Minimax optimization is used to minimize the maximum success of your opponent. For example,

In two-player games, there are two actors with (potentially) different policies, π (player), μ (opponent)

Let R (π, μ) denote the success rate of π versus μ

Minimax optimization w.r.t. π optimizes **min max R (π, μ)** Basically, an agent takes an action based on a policy for a specific task after exploration which depends on the actual state given the previous state and the action taken.
There are two different types of games: Fully observed and Partially observed There are two different types of action spaces: Discrete and Continuous In the case of large state spaces, we can't explore every state and we cannot store the value of each state due to time and storage limitations. Monte Carlo tree search generates experiments with randomly selected actions by repeating the process: 1. Selection, 2. Expansion, 3. Simulation, 4. Backpropagation.

If we keep selecting/simulating randomly, we will not improve but the statistics we collect can be used to improve the selection policy and the simulation/roll-out policy. A greedy policy selects the best action according to some metric. An ϵ-greedy policy chooses the greedy action with probability $1-\epsilon$, and a random action with probability ϵ and trades off exploration and exploitation a little.

For storage, we use neural networks, and during backpropagation, we update the counters and the parameters of neural nets as well.

Module 5 reflection by Himanshu Sahni

In the previous module, we looked into diagnostic systems and their use in the medical domain. The most important learning is that the models should be interpretable. Interpretability may be deemed as accountability when an inquiry into a specific situation needs to be carried out to identify possible areas for improvement or errors. The implementation part of the assignment helped us to understand the meaning of interpretability in an interactive way as we formulated a relatively simple yet effective classifier utilizing rules.

Lecture summary by Nina Uljanić

Games are inherently about actions. In zero-sum games, one player's win is equal to the other player's loss. The goal in these games is to choose actions that improve our chances of winning. As we take action, we change the possible future actions, and to win, we must consider good and bad futures when selecting our actions. We can formalize the best paths using search trees in order to be able to select the best action.

Search trees can be used to enumerate possible futures. To guarantee success against the opponent, we must find a policy that minimizes their maximum success. This is done with the minimax algorithm: $\min_{\pi}\max_{\mu}(\bar{R}(\mu, \pi))$ where π and μ are the policies of the two players, and $\bar{R}$ is the result of the game. Finding the solution can be tricky, as an exhaustive search is not always feasible due to a large number of possible states. Not all states can be evaluated. Moreover, there are execution time and data storage constraints.

A solution to this problem is the Monte Carlo tree search. It deals with random sampling and is based on experiments with randomly selected actions. It repeats the four-step process of selection, expansion, simulation, and backpropagation. Selection refers to finding unexplored nodes, i.e. actions that haven't been tried, expansion refers to the selection of an unvisited child node, simulation refers to the process of continuing playing from the expansion node until a terminal node is reached, and backpropagation refers to updating statistics of nodes based on the result of the simulation, starting from the expansion node and ending with the root node. The statistics collected from the search can be used to improve the selection and simulation policies.

An important note to consider is that there exists a tradeoff between exploration and exploitation when choosing the best-so-far action or finding a better one. Upper confidence trees (UCT) are a solution to this problem. It involves the application of the UCB algorithm to search trees: $UCT(v_i) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$, where v_i is the current node, v is the root node, Q is the accumulated result, and N is the number of visits. The UCT formula balances exploitation and exploration by giving a higher score to nodes that have not been visited as much, encouraging the search algorithm to explore less-visited areas of the search tree, while also taking into account the rewards obtained from previous visits. The first part of the formula represents exploitation, while the second represents exploration.

In conclusion, selecting actions in a zero-sum game requires a strategic approach that involves accounting for possible future actions. Search trees and algorithms such as Monte Carlo tree search and the minimax algorithm are used to achieve this goal. The exploration/exploitation tradeoff can be addressed by using upper confidence trees.

Module 5 reflection by Nina Uljanić

In module 5, we have learned about the use of AI in medicine for diagnostic purposes. It served as an overview of the development of early expert systems, as well as the more recent advances in machine learning and statistics that have led to the development of better algorithms for image analysis and learning with high dimension data. We touched on the topic of potential benefits of using AI in medicine, including improved consistency and speed of diagnosis and the ability to explore data beyond known explanations. Concerns about the reliability and reproducibility of AI systems were mentioned, as well as the need for robustness, fairness, accountability, and interpretability in these systems. Finally, it emphasized the importance of ethical considerations in the use of AI in medicine, particularly in regards to privacy and fairness given the sensitive nature of medical data. Overall, it was a nice introduction to the use of AI in medicine, as well as the potential benefits and concerns associated with this field.

Module 6 - Game playing systems

March 3, 2023

1 Monte Carlo Tree Search Tic Tac Toe

[ ]:
```
import numpy as np
import copy
import random
import math
```

[ ]:
```
def get_player_mark(player):
    if player == 0: # player
        return 'X'
    else:
        return 'O' # computer

def generate_state_from_input(player, action, state): # position 1-9 expressed␣
 ↪in terms of matrix rows and columns
    y_pos, x_pos = map_input_to_action(action) # map to matrix coordinates
    state[y_pos][x_pos] = get_player_mark(player)
    return state

def map_input_to_action(input):
    # Map the action number to the corresponding matrix index
    if action == 1:
        row, col = 0, 0
    elif action == 2:
        row, col = 0, 1
    elif action == 3:
        row, col = 0, 2
    elif action == 4:
        row, col = 1, 0
    elif action == 5:
        row, col = 1, 1
    elif action == 6:
        row, col = 1, 2
    elif action == 7:
        row, col = 2, 0
    elif action == 8:
        row, col = 2, 1
    elif action == 9:
        row, col = 2, 2
    else:
        print("Invalid action number")

    return row, col


def print_grid(grid):
    for row in grid:
        print(" | ".join([" " if col is None else col for col in row]))

def display_move(state, player):
    text = "Player move:" if player == 0 else "Computer move:"
    print("\n{}".format(text))
    print_grid(state)

def is_game_finished(state):
    val = evaluate_state(state)
    if val is None: # if no winner
        for row in state:
            for col in row:
                if col is None: # the game is not over
                    #print("The game is still in progress.\n")
                    return (False, val)
        return (True, val)
    return (True, val)

def evaluate_state(state):
    # check rows
    for row in state:
        if all(x == row[0] for x in row):
            return decode_winner(row[0])

        # check columns
        for col in range(3):
            if all(state[row][col] == state[0][col] for row in range(3)):
                return decode_winner(state[0][col])

    # check diagonals
    if state[0][0] == state[1][1] == state[2][2]:
        return decode_winner(state[0][0])

    if state[0][2] == state[1][1] == state[2][0]:
        return decode_winner(state[0][2])

    # No winner found; (1) it's a draw, or 2) game still in progress
```

```
        return None

def decode_winner(element):
    if element == 'X': # player
        return 0 # player won
    elif element == 'O':
        return 1 # computer won
    return None
    # checking if a game is still in progress is done elsewhere, by counting␣
 ↪Nones

def evaluate_end_state(end, result):
    if end:
        if result == 0:
            print("The player has won!")
        elif result == 1:
            print("The computer has won!")
    else:
        print("It's a draw.")
    print("\nGame over")
```

[ ]:
```
class Node:
    def __init__(self, parent, children, state, player):
        self.parent = parent
        self.children = children
        self.state = state
        self.value = 0 # score (wins and loses) (q-value)
        self.visits = 0 # nr of visits (n)
        self.uct = 0
        self.terminal = False
        self.player = player

    def get_n_actions(self):
        n_actions = sum(elem is None for row in self.state for elem in row)
        return n_actions

    def has_unvisited_children(self):
        if any(None in row for row in self.state):
            return True
        else:
            return False

    def spawn_child(self, state):
        child = Node(self, [], state, (self.player + 1 ) % 2)
        self.children.append(child)
        return child
    def select_unvisited_child(self):
        new_state = self.select_state()
        child = self.spawn_child(new_state)
        self.children.append(child)
        return child

    def select_visited_child(self):
        selected = max(self.children, key=lambda node: node.uct)
        if not isinstance(selected, Node): # if multiple with same value,␣
↪choose random one
            selected = random.choice(selected)
        return selected

    def generate_states(self):
        states = []
        available_indices = [(i, j) for i, row in enumerate(self.state) for j,␣
↪elem in enumerate(row) if elem is None]
        n_states_to_generate = len(available_indices) # generate a state for␣
↪each of these
        for i in range(n_states_to_generate):
            new_state = copy.deepcopy(self.state)
            row, col = available_indices[i] # the slot to be filled in
            new_state[row][col] = get_player_mark(self.player) # fill with␣
↪corresponding mark
            states.append(new_state)
        return states

    def select_state(self):
        states = self.generate_states()
        selected_state = random.choice(states)
        return selected_state

    def update_stats(self, result): # and backpropagate
        self.visits = self.visits + 1
        self.value = self.value + result
        if self.parent is not None:
            self.parent.update_stats(result)
            self.update_uct()

    def update_uct(self):
        exploitation = self.value / self.visits
        exploration = math.sqrt(2*math.log(self.parent.visits)/self.visits) #␣
↪2=c
        self.uct = exploitation + exploration
```

[ ]:
```python
class MCTS:
    def __init__(self, player, state):
        self.root = Node(None, [], state, player)
        self.node = self.root # starting node is root in first iteration, i.e.␣
 ↪at initialization
        self.player = player

    def algorithm(self, iterations):
        for i in range(iterations):
            self.select()
            if self.node.terminal or not self.node.has_unvisited_children():
                print("Node {} is terminal".format(self.node))
                print(self.node.state)
                # no expansion -> simulate and backpropagate
                sim_result = self.simulate();
                print(sim_result)
                self.backpropagate(sim_result)
                self.reset_start_node()
                continue
            self.expand()
            sim_result = self.simulate()
            self.backpropagate(sim_result)
            self.reset_start_node()
            self.player = (self.player + 1) % 2 # alternate players
        return self.select_move()

    def reset_start_node(self): # set start node to root after each iteration
        self.node = self.root

    def select_move(self):
        selected = max(self.root.children, key=lambda node: node.visits)
        if not isinstance(selected, Node):
            selected = random.choice(selected)
        return selected.state

    ### Algorithm steps

    def select(self):
        if len(self.node.children) == 0 and (self.node.get_n_actions() == 0):
            return
        n_actions = self.node.get_n_actions()
        if self.node.terminal or self.node.has_unvisited_children: # at least␣
 ↪one unexplored child node exists
            return # skip selection, move on to next step
        self.node = self.node.select_visited_child() # visited child selected
        self.select() # continue searching
    def expand(self):
        self.node = self.select_unvisited_child() # selects and creates␣
↪the node, storing it in parent.children list

    def simulate(self): # step 3
        sim_state = copy.deepcopy(self.node.state) # using current state as␣
↪start point for simulation
        result = 0 # score, or added to node.value at the end

        available_indices = [(i, j) for i, row in enumerate(sim_state) for j,␣
↪elem in enumerate(row) if elem is None] # find indices of available slots,␣
↪shuffle
        random.shuffle(available_indices) # randomize available slots

        player = self.player
        for n in range(len(available_indices)):
            player = (player + 1) % 2 # alternating player 0 (player) and 1␣
↪(computer); player starts (computer has played in the action selection step␣
↪above
            row, col = available_indices[n] # the slot to be filled in
            sim_state[row][col] = get_player_mark(player) # fill with␣
↪corresponding mark
            outcome = evaluate_state(sim_state) # check if win/loss/draw
            if outcome == 0:
                result = -1
            elif outcome == 1:
                result = 1
            #if outcome is None: # implied
            #    result = 0
        return result

    def backpropagate(self, result):
        self.node.update_stats(result) # recursively update self and parent, if␣
↪it exists
```

[ ]:
```
player = 0 # 'X'
computer = 1 # 'O'
iterations = 1000
grid = [[None, None, None], [None, None, None], [None, None, None]] # empty grid

r = 5 # if player starts
for i in range(r):
    print("\n-------------------------")
    print("Round #{}:\n".format(i+1))
    action = int(input("Enter action (1-9): "))
    grid = generate_state_from_input(player, action, grid)
    display_move(grid, player)
    end, result = is_game_finished(grid)
    if end:
        evaluate_end_state(end, result)
        break

    if i < r:
        mcts = MCTS(computer, grid)
        new_grid = mcts.algorithm(iterations) # run the algorithm - computer␣
↪plays
        display_move(new_grid, computer)
        end, _ = is_game_finished(new_grid)
        if end:
            evaluate_end_state(end, result)
            break
    grid = new_grid
```