Game AI through Markov decision processes

By William Dahl

Introduction:

A Markov Model is used extensively in probability theory. It is a type of stochastic model and is used to model changes in a system (Gagniuc). In a Markov model, future states are assumed to be only dependent on the current state. These assumptions make solving a Markov model a tractable problem and thus, makes Markov models the most desired type of stochastic model in the fields of predictive modeling and probabilistic forecasting. There are four different types of Markov models, they are the: Markov Chain, Hidden Markov model, Markov decision process, and Partially observable Markov decision process. A Markov chain simply models a random variable within a system that changes its state thorough out time (Gagniuc). A Hidden Markov model is a Markov chain where some states of the model that cannot be observed, or measured to be modeled. A Markov decision process is a Markov chain with a respective action vector for transitioning states. A Partially observable Markov decision process is a Markov decision process with states that cannot be modeled, or observed. These models are NP complete but through using approximations they have been applied to controlling simple robots (Kaelbling). The main defining differences between these models are with er the system is autonomous or if it is controlled by an action vector, as well as weather the state of the system are completely observable or if some cannot be measured.

This Project will explore the application Markov decision processes to game A.I. Markov decision processes are typically modeled as a 4-tuple (S, A, P_a, R_a). S is the set of states, A is the set of actions that can be taken from each state, $P_a(s,s')$ is the probability that given the action a the system will transition from state s to state s'. $R_a(s,s')$ is the expected reward from transition to that new state s'

given the action a (Bellman). A Markov decision Process is essentially a Deterministic finite state automaton. We have a series of set of states. Machine takes in a series of inputs that are actions as apposed to the characters in a string. Based on the action given a the next state to transition to is determined. After each transition a reward is outputted. The goal of solving a Markov decision process is to obtain a optimal policy for the system to follow to result in a desired outcome. Solving the Markov decision process is done by iterating over the policy starting from a zero value function until the optimal policy is reached. This is done by looping over all of the states in the system and choosing the action that given the maximum reward from the current state. This is a two step algorithm involving a value update and then a policy update (*Bellman*).

$$V(s) := \sum_{s'} P_{\pi(s)}(s, s') \left(R_{\pi(s)}(s, s') + \gamma V(s') \right)$$

$$\pi(s) := \operatorname{argmax}_a \left\{ \sum_{s'} P(s' \mid s, a) \left(R(s' \mid s, a) + \gamma V(s') \right) \right\}$$

This method of solving the Markov decision process is done through dynamic programming. Dynamic programming is done by breaking a problem down into smaller problems. There are many variants to the two step method shown above. The Method that I chose to use is known as value iteration. In value iteration is more optimize method than the traditional two step method and only requires one step (Bellman). It preforms the value update and calculates the value of the policy at the state when it is needed to update the value.

$$V_{i+1}(s) := \max_a \left\{ \sum_{s'} P_a(s, s') \left(R_a(s, s') + \gamma V_i(s') \right) \right\},$$

The goal of this project is to solve the Markov decision problem using the value iteration method and use it to train a bot to find its way to the goal state on a given game board. The goal is for the algorithm to work no matter the board size, layout, goal state, and start state.

Methodology:

The language I chose was python3. I chose python because it is higher level language and would be easy to work with in structuring my data and performing on it. I Modeled my Markov decision process as a Game board. It consisted of a set of states and a set of actions. The board is represented as a 2D list with each index representing a state. In the board one, state is marked with a ‘G’, this is the Goal State. I dynamically created the set of states so that the state count would change depending on how large or the shape of the game board. I chose to represent the states as a tuple consisting of the indices of the state on the game board.

E.X: The state (0,0) would represent board[0][0].

The actions that can be taken from each state are ‘up’, ‘down’, ‘right’, and ‘left’. These actions are represented as a list of strings containing actions respectively.

E.X: [‘up’, ‘down’, ‘right’, ‘left’]

The Transition function for the Markov model Takes in the current state and an action. Then, based on the action given, the transition is carried out. If the transition would result in the next state being outside of the state space, then the state remains the same and no transition is made. If a transition can be made, the next state is checked to see if it is the goal state. If it is then the next state is returned along with a reward of 0, other wise the new state is returned with a reward of -1. The reward for every state other than the goal state is -1, and the reward for the goal state is 0. I chose to uses these numbers so that when a transition is made the score goes down. This is to encourage the bot to make the least amount of moves as possible to get to the goal state. The goal state gives a reward of 0 so that the score will actually go up when it is calculated cementing in the policy that the goal state is where to go to get the maximum score.

E.X.: board.Trans((0,0), ‘right’) will yield (0,1) as the next state and -1 as the reward (assuming (0,1) is not the goal state)

The Prob_of_Action() function takes in the current state and calculates the probability distribution of the actions from the state.

E.X.: The probability distribution for the state (0,0) in a 4x4 grid would be 1/3 as 3 moves can be done from that state.

The Train() function takes in the game environment and the acceptable error between the iterative policies.

1. I create and initialize the value and police hash tables to 0 value functions.
    1. Values is a hash map with the state acting as the key and the score for that state as the value.
    2. Policy is a hash map with the key as the action to take at that state.
2. Next, I iterate over the states map creating a new hash map for every state that holds the score for the actions from that state.
3. I get the scores of the of the actions by using the look_ahead() functions
    1. This function gets the next state and the reward of that state when the action is executed from the current state.
    2. It then computes the new score for that action for the current state.
    3. This is where the Bellman value equation is used, with the probability being that of the 1/moves from the state.
4. Next, I get the max score from hash map with all the action scores and subtract that from the score of the current state to get the ratio of change between the scores.

5. I set the score of the new state to the max score and get the action corresponding to the max score.
6. Next I set the policy for that state to the action with the optimal score.
7. We continue to do this until the change in the states scores is sufficiently small and we have our optimal policy.

E.X: The optimal policy will look something like this:

{(0, 0): 'down', (0, 1): 'down', (0, 2): 'down', (0, 3): 'down', (1, 0): 'down', (1, 1): 'down', (1, 2): 'down', (1, 3): 'down', (2, 0): 'down', (2, 1): 'down', (2, 2): 'down', (2, 3): 'down', (3, 0): 'right', (3, 1): 'right', (3, 2): 'right', (3, 3): 'down'}

Now that we have our optimal policy its time to play the game. We give the agent a board to use, a start state and a policy to follow. The agent will then follow the policy for each state it enters until the goal state is reached. The agents position on the board is marked as an ‘X’. The board is printed out with the agents position after every transition the agent makes.

Testing:

To test the generated polices I created 2 game boards:

```
# Game boards
# 'G' is the Goal state
board = [[' ', ' ', ' ', ' '],
         [' ', ' ', ' ', ' '],
         [' ', ' ', ' ', ' '],
         [' ', ' ', ' ', 'G']]

board2 = [[' ', ' ', ' ', 'G'],
          [' ', ' ', ' '],
          [' ', ' ', ' ', ' ', ' '],
          [' ', ' '],
          [' ']]
```

Game board 1 is a simple 4x4 board with the goal state, ‘G’, in the bottom right hand corner. Game board 2, on the other hand, is an irregularly shaped game board with the goal state being in the top right corner. The generated policies must be sufficient in getting the agent from the start state to the goal state in the minimum amount of step required. The start state given for game board 1 is (0,0) and the start state for game board 2 is (4,0). The minimum amount of steps required for game board 1 is 6 and the minimum for game board 2 is 7 from their respective start states.

Results:

The resulting generated polices are:

Policy for Game Board 1:

{(0, 0): 'down', (0, 1): 'down', (0, 2): 'down', (0, 3): 'left', (1, 0): 'right', (1, 1): 'down', (1, 2): 'down', (1, 3): 'left', (2, 0): 'right', (2, 1): 'right', (2, 2): 'down', (2, 3): 'left', (3, 0): 'right', (3, 1): 'right', (3, 2): 'right', (3, 3): 'up'}

Policy for Game Board 2:

{(0, 0): 'right', (0, 1): 'right', (0, 2): 'right', (0, 3): 'up', (1, 0): 'right', (1, 1): 'up', (1, 2): 'up', (2, 0): 'up', (2, 1): 'up', (2, 2): 'left', (2, 3): 'left', (2, 4): 'left', (3, 0): 'up', (3, 1): 'up', (4, 0): 'up'}

When the game board are run through the play function with the generated policies, The agent in game board 1 reaches the goal state 6 steps, this is the minimum number of steps needed to get from its start state to the goal state. The agent Game Board 2 reached the Goal state in 7 steps which was the minimum number of steps required to get to the goal state from the start state. Thus, both agents got to the Goal state in the minimum amount of steps using the generated policies for each game board showing that generated polices are the optimal polices and the value iteration to solve the Markov decision process is a success.

User Guide:

To run the program simply type “python3 MDP.py” into the terminal. A board must be created in the form of a 2D list, the board does not need to be a symmetric matrix. The Goal state must be labeled in the board by placing a ‘G’ in the index of the goal state. See the screen shot of the board used for testing as an example. This board must then be put into a GameBoard() object to create a Markov Decision Process based on the board as the environment. Next, run the newly created GameBoard object through the Train() function getting the resulting policy. Then, run the agent through the game board using the policy by using the play() function. Pass the GameBoard, the start state (as a tuple: e.x. (0,0)) and the the generated policy into the function. When you run the program the steps that the agent takes through out the game board will be printed to standard out.

Example:

```
# Game boards
# 'G' is the Goal state
board = [[' ', ' ', ' ', ' '],
         [' ', ' ', ' ', ' '],
         [' ', ' ', ' ', ' '],
         [' ', ' ', ' ', 'G']]

board2 = [[' ', ' ', ' ', 'G'],
          [' ', ' ', ' '],
          [' ', ' ', ' ', ' ', ' '],
          [' ', ' '],
          [' ']]

# Makes Game board from board
game_board = GameBoard(board)
#trains the polcy given the game board
policy = Train(game_board)
#plays the game using the trained policy
play(game_board, (0,0), policy)

game_board2 = GameBoard(board2)
policy = Train(game_board2)
play(game_board2, (4,0), policy)
```

Conclusion:

A Markov Model can be used to Model probabilistic changes in a random variable through time. The assumption in Markov models that the future only depends on the present makes the Markov model a very versatile stochastic process (Gagniuc). This assumption makes the ability to predict future event over time a possible problem to solve and to be computed. The Markov decision process is a type of Markov model that is controlled through an action vector that controls the action that can be taken from each state (*Ashraf)*. In this way a Markov decision process is very similar to that of a non-deterministic truing machine. The states in the system are like that of the cells on the tape and the action vector is like that of instruction set. It is non-deterministic as the transition from each state is not deterministic as different state can have multiple actions taken from it with a certain probability. As this is that of a non-deterministic nature, solving a Markov decision problem is a NP class problem (Kaelbling*)*. A Markov decision process can be solved through he use of dynamic programming and iterative steps (Howard). The use of the bellman optimal equation, also know as value iteration (the method used in this project) Allows for the solution to the model being approximated to a sufficient amount (*Ashraf)*. Through the use of the value iteration , the values in the states can be proximate and a action policy can be generated to help solve the process. As an NP class problem the solution (the policy) to the Markov decision problem can be checked in polynomial time, thus policy evaluation is a P class problem and can be computed (Xu*)*. It is through this policy evaluation that we come up with the optimal solution to the Markov decision process. Using these policy generated from solving these Markov decisions problems we can apply these polices to A.I. The policy can be used to direct an agent through a map or a maze, And can be used in adversarial A.I in games to pose as a threat to a player. An example of this would be the ghosts in pac-man. Here the location of pac-man in the map in the goal state and the ghosts are the agents trying to get to the goal state and kill pac-man. Another possible application would be to GPS where the goal state is the end destination, the start state is the starting destination,

actions include Turing left and right along roads and the states are the address between the start state and the goal state.

References:

Gagniuc, Paul A. (2017). Markov Chains: From Theory to Implementation and Experimentation. USA, NJ: John Wiley & Sons.

Kaelbling, L. P.; Littman, M. L.; Cassandra, A. R. (1998). "Planning and acting in partially observable stochastic domains"

“Markov Model.” Wikipedia, Wikimedia Foundation, 9 May 2019, en.wikipedia.org/wiki/Markov_model.

Bellman, R. (1957). "A Markovian Decision Process". Journal of Mathematics and Mechanics.

Howard, Ronald A. (1960). Dynamic Programming and Markov Processes. The M.I.T. Press.

“Markov Decision Process.” Wikipedia, Wikimedia Foundation, 24 Apr. 2019, en.wikipedia.org/wiki/Markov_decision_process.

“Dynamic Programming.” Wikipedia, Wikimedia Foundation, 27 Apr. 2019, en.wikipedia.org/wiki/Dynamic_programming.

Ashraf, Mohammad. “Reinforcement Learning Demystified: Solving MDPs with Dynamic Programming.” Towards Data Science, Towards Data Science, 18 May 2018, towardsdatascience.com/reinforcement-learning-demystified-solving-mdps-with-dynamic-programming-b52c8093c919.

Xu, Huan, and Shie Manor. Probabilistic Goal Markov Decision Processes. Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, pp. 6–6, Probabilistic Goal Markov Decision Processes.

Ashraf, Mohammad. “Reinforcement Learning Demystified: Markov Decision Processes (Part 1).” Towards Data Science, Towards Data Science, 11 Apr. 2018, towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690.

Appendix:

Code:

```
# William Dahl
# ICSI 409
# Final Project
# May 10th, 2019

# Implmenetation of a Markov Model
# Applies a Markov Decision Process to Reinforcment Learning
# Trains a bot to find the most optimal route to The goal in a given Game Board

# Makes states off of given game board
# returns a list of states
def makeStates(board):
    # holds created states
    states = []

    # loops through the game board
    for i in range(len(board)):
        for j in range(len(board[i])):
            # states are represented as their coordinates on the board
            states.append((i,j))
```

```
    # retuns a list of the sates for the board
    return states

# Game board enviroment
class GameBoard():
    # intializes the game board with a given board
    def __init__(self, board):
        self.board = board # 2D list for board
        self.states = makeStates(board) # list of states
        self.actions = ['up', 'down', 'right', 'left'] # possable actions to take in each state

    # Returns the next state and the reward from that state after
    # the given action is taken
    def Trans(self, state, action):
        row = state[0]
        col = state[1]

        # Checks given action
        if action == 'up':
            # Checks if action will move position off of the game board
            if row > 0 and len(self.board[row-1])-1 > col: # if move will stay on board
                #checks if new state is the Goal state
                if self.board[row-1][col] == 'G':
                    # returns the new state and the reward of 0 for getting to the Goal state
                    return (row-1,col), 0
                else:
                    # returns the new state and -1 for the reward from any other state
                    return (row-1, col), -1
            else: # if move will move off of board
                # checks if current state is goal state
```

```
        if self.board[row][col] == 'G':
            # returns current state and 0 for reward
            return state, 0
        else:
            # returns curent state and -1 for reward
            return state, -1

if action == 'down':
    if row < len(self.board)-1 and len(self.board[row+1])-1 > col:
        if self.board[row+1][col] == 'G':
            return (row+1,col), 0
        else:
            return (row+1, col), -1
    else:
        if self.board[row][col] == 'G':
            return state, 0
        else:
            return state, -1

if action == 'right':
    if col < len(self.board[row])-1:
        if self.board[row][col+1] == 'G':
            return (row,col+1), 0
        else:
            return (row, col+1), -1
    else:
        if self.board[row][col] == 'G':
            return state, 0
        else:
            return state, -1
```

```
    if action == 'left':
        if col > 0:
            if self.board[row][col-1] == 'G':
                return (row,col-1), 0
            else:
                return (row, col-1), -1
        else:
            if self.board[row][col] == 'G':
                return state, 0
            else:
                return state, -1

#calculates the probabilty of the action occuring
def Prob_of_Action(self, state):
    pos_actions = 0 #count of possable actions from the state
    #test if action can happen from the state using the transition fucntion
    for action in self.actions:
        #if the state returned is not the same state then the action can happen
        if self.Trans(state, action)[0] != state:
            pos_actions += 1

    # returns the probabily of the action being able to happen.
    return 1/pos_actions

# Displays the game board to std out
def print(self):
    for row in self.board:
        print(row)
```

```
# Preforms a look ahead to the next state if a action is taken
# returns the reulting score for taking the given action from the given state
def look_ahead(env, values, action, state):
    # gets the next state after the given action is taken
    next_state, reward = env.Trans(state, action)
    prob = env.Prob_of_Action(state)
    # prob of .25 that the action is take as there are 4 actions
    # gets the new overall score for the action from the state
    new_value = prob * (reward + values[next_state])
    return new_value #returns the new score

# Trains the model given for a given game enviroment
# trains until the error is within the given amount
def Train(env):
    values = dict()# Holds the scores for each state
    policy = dict()# Holds the policy from each state

    # itialzes the value and policys to 0
    for state in env.states:
        values[state] = 0
        policy[state] = 'up'

    # Trains the policy until it is within the given accuracy
    while True:
        # change in policy
        change = 0
        # loops over the states
        for state in env.states:
            action_scores = dict()#holds the score for each action for the current state
            # loops over the actions
```

```
            for action in env.actions:
                # sets the score for the action
                action_scores[action] = look_ahead(env, values, action, state)

            # Gets the best possiable action to exectute based on the score
            # for the action in action_scores
            max_action_score = max(action_scores.values())
            # updates the current change in policy for this iteration
            change = max(change, abs(max_action_score - values[state]))
            #sets the new score for the current state to the max score in action_scores
            values[state] = max_action_score
            # the best possaible action from the current state
            best_action = max(action_scores, key=action_scores.get)

            # sets the value for the new best action for the current state to 1
            # and all other to 0
            # the action with the value of 1 will be the one taken when the state is entered
            policy[state] = best_action
        #determines if the change in the policy interation is susfecintly small
        if change <= 0:
            break

    # returns the newily trained policy
    return policy

# Plays the Game given the game board, start state, and the policy to use
def play(board, state, policy):
    row = state[0]
    col = state[1]
```

```
    # Exectues actions from the policy until the goal state is reached
    while board.board[row][col] != 'G':
        # lables the bots position on the board as an 'X'
        board.board[row][col] = 'X'
        #prints the current state of the game board
        board.print()
        print()
        # gets the new state after the action is executed
        new_state = board.Trans(state, policy[state])[0]

        #clears the old state
        board.board[row][col] = ' '
        #sets new state corrdinates
        row = new_state[0]
        col = new_state[1]
        #iterates through
        state = new_state

    print("WIN!")

# Game boards
# 'G' is the Goal state
board = [[' ', ' ', ' ', ' '],
         [' ', ' ', ' ', ' '],
         [' ', ' ', ' ', ' '],
         [' ', ' ', ' ', 'G']]

board2 = [[' ', ' ', ' ', 'G'],
          [' ', ' ', ' '],
          [' ', ' ', ' ', ' ', ' '],
```

```
        [' ', ' '],
        [' ']]

# Makes Game board from board
game_board = GameBoard(board)
#trains the polcy given the game board
policy = Train(game_board)
#plays the game using the trained policy
play(game_board, (0,0), policy)

game_board2 = GameBoard(board2)
policy = Train(game_board2)
play(game_board2, (4,0), policy)
```

Link to video:

https://www.screencast.com/t/5QaQxjx9