High Level Description 02. May 2022

Kathrin Rupp

[kathrin.rupp@code.berlin](mailto:kathrin.rupp@code.berlin)

4th Semester

Professor: Frank Trollmann

SE\_ SE\_14 Artificial Intelligence Basics

Standart Assessment

CS50’s Introduction to Artificial Intelligence with Python

**Planning**

Discription

Tic Tac Toe is a game in which two players (AI will take either MAX or MIN) seek in alternate turns to complete a row, a column, or a diagonal with either three O's or three X's drawn in the spaces of a grid of nine squares; noughts and crosses. MAX aims to maximize its value and MIN aims to minimize its value. The game starts with an empty board as the initialize state and the players take turn in filling the squares on the board. The game ends when it reaches the terminal state that is defined by either a “draw” when no one wins and the board is fully filled or “MAX or MIN wins” if one of them manage to put X for MAX or O for MIN in 3 consequtive times in a diagonal, vertical, or horizontal line.

 The player function should take a board state as input, and return which player’s turn it is (either X or O).

* In the initial game state, X gets the first move. Subsequently, the player alternates with each additional move.
* Any return value is acceptable if a terminal board is provided as input (i.e., the game is already over).

 The actions function should return a set of all of the possible actions that can be taken on a given board.

* Each action should be represented as a tuple (i, j) where i corresponds to the row of the move (0, 1, or 2) and j corresponds to which cell in the row corresponds to the move (also 0, 1, or 2).
* Possible moves are any cells on the board that do not already have an X or an O in them.
* Any return value is acceptable if a terminal board is provided as input.

 The result function takes a board and an action as input, and should return a new board state, without modifying the original board.

* If action is not a valid action for the board, your program should [raise an exception](https://docs.python.org/3/tutorial/errors.html#raising-exceptions).
* The returned board state should be the board that would result from taking the original input board, and letting the player whose turn it is make their move at the cell indicated by the input action.
* Importantly, the original board should be left unmodified: since Minimax will ultimately require considering many different board states during its computation. This means that simply updating a cell in board itself is not a correct implementation of the result function. You’ll likely want to make a [deep copy](https://docs.python.org/3/library/copy.html#copy.deepcopy) of the board first before making any changes.

 The winner function should accept a board as input, and return the winner of the board if there is one.

* If the X player has won the game, your function should return X. If the O player has won the game, your function should return O.
* One can win the game with three of their moves in a row horizontally, vertically, or diagonally.
* You may assume that there will be at most one winner (that is, no board will ever have both players with three-in-a-row, since that would be an invalid board state).
* If there is no winner of the game (either because the game is in progress, or because it ended in a tie), the function should return None.

 The terminal function should accept a board as input, and return a boolean value indicating whether the game is over.

* If the game is over, either because someone has won the game or because all cells have been filled without anyone winning, the function should return True.
* Otherwise, the function should return False if the game is still in progress.

 The utility function should accept a terminal board as input and output the utility of the board.

* If X has won the game, the utility is 1. If O has won the game, the utility is -1. If the game has ended in a tie, the utility is 0.
* You may assume utility will only be called on a board if terminal(board) is True.

 The minimax function should take a board as input, and return the optimal move for the player to move on that board.

* The move returned should be the optimal action (i, j) that is one of the allowable actions on the board. If multiple moves are equally optimal, any of those moves is acceptable.
* If the board is a terminal board, the minimax function should return None

Algorithms

Minimax

Knowing based on the state whose turn it is, the algorithm can know whether the current player, when playing optimally, will pick the action that leads to a state with a lower or a higher value. This way, alternating between minimizing and maximizing, the algorithm creates values for the state that would result from each possible action. To give a more concrete example, we can imagine that the maximizing player asks at every turn: “if I take this action, a new state will result. If the minimizing player plays optimally, what action can that player take to bring to the lowest value?” However, to answer this question, the maximizing player has to ask: “To know what the minimizing player will do, I need to simulate the same process in the minimizer’s mind: the minimizing player will try to ask: ‘if I take this action, what action can the maximizing player take to bring to the highest value?’” This is a recursive process, and it could be hard to wrap your head around it; looking at the pseudo code below can help. Eventually, through this recursive reasoning process, the maximizing player generates values for each state that could result from all the possible actions at the current state. After having these values, the maximizing player chooses the highest one.

The Maximizer Considers the Possible Values of Future States.

To put it in pseudocode, the Minimax algorithm works the following way:

* Given a state *s*
  + The maximizing player picks action *a* in *Actions(s)* that produces the highest value of *Min-Value(Result(s, a))*.
  + The minimizing player picks action *a* in *Actions(s)* that produces the lowest value of *Max-Value(Result(s, a))*.
* Function *Max-Value(state)*
  + *v = -∞*
  + if *Terminal(state)*:

​ return *Utility(state)*

* + for *action* in *Actions(state)*:

​ *v = Max(v, Min-Value(Result(state, action)))*

return *v*

* Function *Min-Value(state)*:
  + *v = ∞*
  + if *Terminal(state)*:

​ return *Utility(state)*

* + for *action* in *Actions(state)*:

​ *v = Min(v, Max-Value(Result(state, action)))*

return *v*

Alpha-Beta Pruning

A way to optimize Minimax, Alpha-Beta Pruning skips some of the recursive computations that are decidedly unfavorable. After establishing the value of one action, if there is initial evidence that the following action can bring the opponent to get to a better score than the already established action, there is no need to further investigate this action because it will decidedly be less favorable than the previously established one.

This is most easily shown with an example: a maximizing player knows that, at the next step, the minimizing player will try to achieve the lowest score. Suppose the maximizing player has three possible actions, and the first one is valued at 4. Then the player starts generating the value for the next action. To do this, the player generates the values of the minimizer’s actions if the current player makes this action, knowing that the minimizer will choose the lowest one. However, before finishing the computation for all the possible actions of the minimizer, the player sees that one of the options has a value of three. This means that there is no reason to keep on exploring the other possible actions for the minimizing player. The value of the not-yet-valued action doesn’t matter, be it 10 or (-10). If the value is 10, the minimizer will choose the lowest option, 3, which is already worse than the preestablished 4. If the not-yet-valued action would turn out to be (-10), the minimizer will this option, (-10), which is even more unfavorable to the maximizer. Therefore, computing additional possible actions for the minimizer at this point is irrelevant to the maximizer, because the maximizing player already has an unequivocally better choice whose value is 4.

Result

You can not win against AI. The best you can reach is a draw.

**Reasoning and Planning**

Description

In a Knights and Knaves puzzle, the following information is given: Each character is either a knight or a knave. A knight will always tell the truth: if knight states a sentence, then that sentence is true. Conversely, a knave will always lie: if a knave states a sentence, then that sentence is false.

The objective of the puzzle is, given a set of sentences spoken by each of the characters, determine, for each character, whether that character is a knight or a knave.

For example, consider a simple puzzle with just a single character named A. A says “I am both a knight and a knave.”

Logically, we might reason that if A were a knight, then that sentence would have to be true. But we know that the sentence cannot possibly be true, because A cannot be both a knight and a knave – we know that each character is either a knight or a knave, but not both. So, we could conclude, A must be a knave.

That puzzle was on the simpler side. With more characters and more sentences, the puzzles can get trickier! Your task in this problem is to determine how to represent these puzzles using propositional logic, such that an AI running a model-checking algorithm could solve these puzzles for us.

* Puzzle 0 is the puzzle from the Background. It contains a single character, A.
  + A says “I am both a knight and a knave.”
* Puzzle 1 has two characters: A and B.
  + A says “We are both knaves.”
  + B says nothing.
* Puzzle 2 has two characters: A and B.
  + A says “We are the same kind.”
  + B says “We are of different kinds.”
* Puzzle 3 has three characters: A, B, and C.
  + A says either “I am a knight.” or “I am a knave.”, but you don’t know which.
  + B says “A said ‘I am a knave.’”
  + B then says “C is a knave.”
  + C says “A is a knight.”

In each of the above puzzles, each character is either a knight or a knave. Every sentence spoken by a knight is true, and every sentence spoken by a knave is false.

Algorithms

Interference Algorithm with model checking approach works be enumerating all possible models to check that a is true in all models in which Knowledge Base is true. Models here means to assign “True” and/or “False” to every propositional character.

Model

The model is an assignment of a truth value to every proposition. To reiterate, propositions are statements about the world that can be either true or false. However, knowledge about the world is represented in the truth values of these propositions. The model is the truth-value assignment that provides information about the world.

For example, if P: “It is raining.” and Q: “It is Tuesday.”, a model could be the following truth-value assignment: {P = True, Q = False}. This model means that it is raining, but it is not Tuesday. However, there are more possible models in this situation (for example, {P = True, Q = True}, where it is both raining an a Tuesday). In fact, the number of possible models is 2 to the power of the number of propositions. In this case, we had 2 propositions, so 2²=4 possible models.

Knowledge Base (KB)

The knowledge base is a set of sentences known by a knowledge-based agent. This is knowledge that the AI is provided about the world in the form of propositional logic sentences that can be used to make additional inferences about the world.

Entailment (⊨)

If α ⊨ β (α entails β), then in any world where α is true, β is true, too.

For example, if α: “It is a Tuesday in January” and β: “It is a Tuesday,” then we know that α ⊨ β. If it is true that it it a Tuesday in January, we also know that it is a Tuesday. Entailment is different from implication. Implication is a logical connective between two propositions. Entailment, on the other hand, is a relation that means that if all the information in α is true, then all the information in β is true.

Inference

Inference is the process of deriving new sentences from old ones.

For instance, in the Harry Potter example earlier, sentences 4 and 5 were inferred from sentences 1, 2, and 3.

There are multiple ways to infer new knowledge based on existing knowledge. First, we will consider the Model Checking algorithm.

To determine if KB ⊨ α (in other words, answering the question: “can we conclude that α is true based on our knowledge base”)

Enumerate all possible models.

If in every model where KB is true, α is true as well, then KB entails α (KB ⊨ α).

Consider the following example:

P: It is a Tuesday. Q: It is raining. R: Harry will go for a run. KB: (P ∧ ¬Q) → R (in words, P and not Q imply R) P (P is true) ¬Q (Q is false) Query: R (We want to know whether R is true or false; Does KB ⊨ R?)

To answer the query using the Model Checking algorithm, we enumerate all possible models.

Result

AI can say who is knight and knave