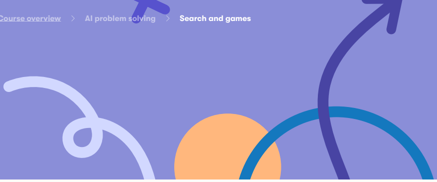
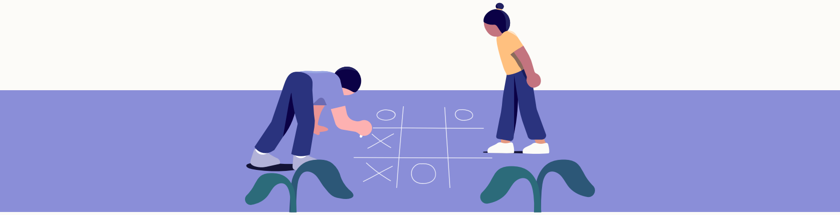


III. Search and games

In this section, we will study a classic AI problem: games. The simplest scenario, which we will focus on for the sake of clarity, are two-player, perfect-information games such as tic-tac-toe and chess.

Example: playing tic tac toe

Maxine and Minnie are true game enthusiasts. They just love games. Especially two-person, perfect information games such as tic-tac-toe or chess. One day they were playing tic-tac-toe. Maxine, or Max as her friends call her, was playing with X. Minnie, or Min as her friends call her, had the Os. Min had just played her turn and the board looked as follows:



Max was looking at the board and contemplating her next move, as it was her turn, when she suddenly buried her face in her hands in despair, looking quite like Garry Kasparov playing Deep Blue in 1997.

Yes, Min was close to getting three Os on the top row, but Max could easily put a stop to that plan. So why was Max so pessimistic?

Game trees

To solve games using AI, we will introduce the concept of a game tree. The different states of the game are represented by nodes in the game tree, very similar to the above planning problems. The idea is just slightly different. In the game tree, the nodes are arranged in levels that correspond to each player's turns in the game so that the “root” node of the tree (usually depicted at the top of the diagram) is the beginning position in the game. In tic-tac-toe, this would be the empty grid with no Xs or Os played yet. Under root, on the second level, there are the possible states that can result from the first player’s moves, be it X or O. We call these nodes the “children” of the root node.

Each node on the second level, would further have as its children nodes the states that can be reached from it by the opposing player's moves. This is continued, level by level, until reaching states where the game is over. In tic-tac-toe, this means that either one of the players gets a line of three and wins, or the board is full and the game ends in a tie.

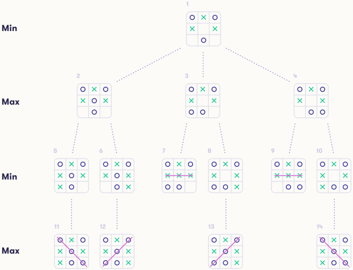
Minimizing and maximizing value

In order to be able to create game AI that attempts to win the game, we attach a numerical value to each possible end result. To the board positions where X has a line of three so that Max wins, we attach the value +1, and likewise, to the positions where Min wins with three Os in a row we attach the value -1. For the positions where the board is full and neither player wins, we use the neutral value 0 (it doesn't really matter what the values are as long as they are in this order so that Max tries to maximize the value, and Min tries to minimize it).

A sample game tree

Consider, for example, the following game tree which begins not at the root but in the middle of the game (because otherwise, the tree would be way too big to display). Note that this is different from the game shown in the illustration in the beginning of this section. We have numbered the nodes with numbers 1, 2, ..., 14.

The tree is composed of alternating layers where it is either Min's turn to place an O or Max's turn to place an X at any of the vacant slots on the board. The player whose turn it is to play next is shown at the left.



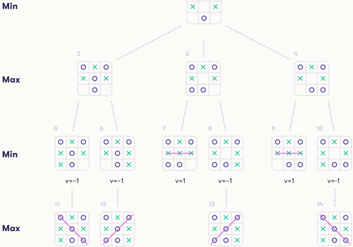
The game continues at the board position shown in the root node, numbered as (1) at the top, with Min's turn to place O at any of the three vacant cells. Nodes (2)–(4) show the board positions resulting from each of the three choices respectively. In the next step, each node has two possible choices for Max to play X each, and so the tree branches again.

When starting from the above starting position, the game always ends in a row of three: in nodes (7) and (9), the winner is Max who plays with X, and in nodes (11)–(14) the winner is Min who plays with O.

Note that since the players' turns alternate, the levels can be labeled as Min levels and Max levels, which indicates whose turn it is.

Being strategic

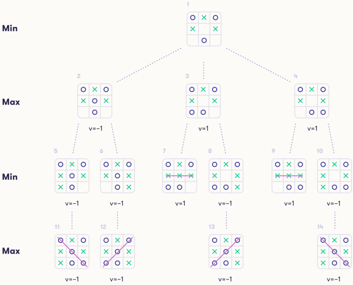
Consider nodes (5)–(10) on the second level from the bottom. In nodes (7) and (9), the game is over, and Max wins with three X's in a row. The value of these positions is +1. In the remaining nodes, (5), (6), (8), and (10), the game is also practically over, since Min only needs to place her O in the only remaining cell to win. In other words, we know how the game will end at each node on the second level from the bottom. We can therefore decide that the value of nodes (5), (6), (8), and (10) is also –1.



Here comes the interesting part. Let’s consider the values of the nodes one level higher towards the root: nodes (2)–(4). Since we observed that both of the children of (2), i.e., nodes (5) and (6), lead to Min’s victory, we can without hesitation attach the value -1 to node (2) as well. However, for node (3), the left child (7) leads to Max’s victory, +1, but the right child (8) leads to Min winning, -1. What is the value of node (3)? Think about this for a while, keeping in mind who makes the choice at node (3).

Since it is Max’s turn to play, she will of course choose the left child, node (7). Thus, every time we reach the board position in node (3), Max can ensure victory, and we can attach the value +1 to node (3).

The same holds for node (4): again, since Max can choose where to put her X, she can always ensure victory, and we attach the value +1 to node (4).



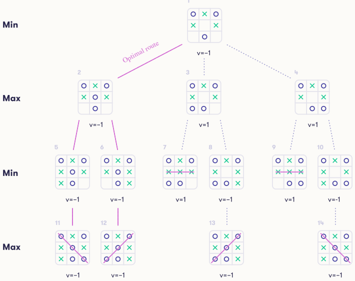
Determining who wins

The most important lesson in this section is to apply the above kind of reasoning repeatedly to determine the result of the game in advance from any board position.

So far, we have decided that the value of node (2) is –1, which means that if we end up in such a board position, Min can ensure winning, and that the reverse holds for nodes (3) and (4): their value is +1, which means that Max can be sure to win if she only plays her own turn wisely.

Finally, we can deduce that since Min is an experienced player, she can reach the same conclusion, and thus she only has one real option: play the O in the middle of the board.

In the diagram below, we have included the value of each node as well as the optimal game play starting at Min's turn in the root node.



The value of the root node = who wins

The value of the root node, which is said to be the value of the game, tells us who wins (and how much, if the outcome is not just plain win or lose): Max wins if the value of the game is +1, Min if the value is –1, and if the value is 0, then the game will end in a draw. In other games, the value may also take other values (such as the monetary value of the chips in front of you in poker for example).

This all is based on the assumption that both players choose what is best for them and that what is best for one is the worst for the other (so called "zero-sum game").

Note

Finding the optimal moves

Having determined the values of all the nodes in the game tree, the optimal moves can be deduced: at any Min node (where it is Min's turn), the optimal choice is given by the child node whose value is minimal, and conversely, at any Max node (where it is Max's turn), the optimal choice is given by the child node whose value is maximal. Sometimes there are many equally good choices that are, well, equally good, and the outcome will be the same no matter which one of them is picked.

The Minimax algorithm

We can exploit the above concept of the value of the game to obtain an algorithm called the Minimax algorithm. It guarantees optimal game play in, theoretically speaking, any deterministic, two-person, perfect-information zero-sum game. Given a state of the game, the algorithm simply computes the values of the children of the given state and chooses the one that has the maximum value if it is Max's turn, and the one that has the minimum value if it is Min's turn.

The algorithm can be implemented using a few lines of code. However, we will be satisfied with having grasped the main idea. If you are interested in taking a look at the actual algorithm (alert: programming required) feel free to check out, for example, Wikipedia: Minimax.

Sounds good, can I go home now?

As stated above, the Minimax algorithm can be used to implement optimal game play in any deterministic, two-player, perfect-information zero-sum game. Such games include tic-tac-toe, connect four, chess, Go, etc. Rock-paper-scissors is not in this class of games since it involves information hidden from the other player; nor are Monopoly or backgammon which are not deterministic. So as far as this topic is concerned, is that all folks, can we go home now? The answer is that in theory, yes, but in practice, no.

Note

The problem of massive game trees

In many games, the game tree is simply way too big to traverse in full. For example, in chess the average branching factor, i.e., the average number of children (available moves) per node is about 35. That means that to explore all the possible scenarios up to only two moves ahead, we need to visit approximately 35 x 35 = 1225 nodes – probably not your favorite pencil-and-paper homework exercise. A look-ahead of three moves requires visiting 42875 nodes; four moves 1500625; and ten moves 2758547353515625 (that’s about 2.7 quadrillion) nodes. In Go, the average branching factor is estimated to be about 250. Go means no-go for Minimax.

More tricks: Managing massive game trees

A few more tricks are needed to manage massive game trees. Many of them were crucial elements in IBM’s Deep Blue computer defeating the chess world champion, Garry Kasparov, in 1997.

If we can afford to explore only a small part of the game tree, we need a way to stop the Minimax algorithm before reaching an end-node, i.e., a node where the game is over and the winner is known. This is achieved by using a so called **heuristic evaluation function** that takes as input a board position, including the information about which player’s turn is next, and returns a score that should be an estimate of the likely outcome of the game continuing from the given board position.

Note

Good heuristics

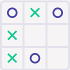
Good heuristics for chess, for example, typically count the amount of material (pieces) weighted by their type: the queen is usually considered worth about two times as much as a rook, three times a knight or a bishop, and nine times as much as a pawn. The king is of course worth more than all other things combined since losing it amounts to losing the game. Further, occupying the strategically important positions near the middle of the board is considered an advantage and the heuristics assign higher value to such positions.

The minimax algorithm presented above requires minimal changes to obtain a **depth-limited** version where the heuristic is returned at all nodes at a given depth limit: the depth simply refers to the number of steps that the game tree is expanded before applying a heuristic evaluation function.

Unanswered

Exercise 7: Why so pessimistic, Max?

Let's return to the tic-tac-toe game described in the beginning of this section. To narrow down the space of possible end-games to consider, we can observe that Max must clearly place an X on the top row to avoid imminent defeat:



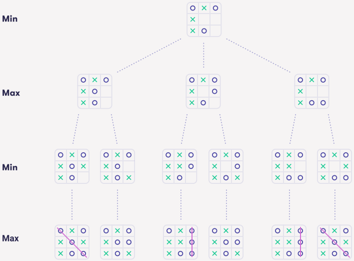
Now it's Min's turn to play an O. Evaluate the value of this state of the game as well as the other states in the game tree where the above position is the root, using the Minimax algorithm.

Your task:
Look at the game tree starting from the below board position. Using a pencil and paper, fill in the values of the bottom-level nodes where the game is over. Note that this time some of the games end in a draw, which means that the values of the node is 0 (instead of −1 or 1).

Next continue filling the values of the nodes in the next level up. Since there is no branching at that level, the values on the second-lowest level are the same as at the bottom level.

On the second-highest level, fill in the values by choosing for each node the maximum of the values of the child nodes – as you notice, this is a MAX level. Finally, fill in the root node's value by choosing the minimum of the root node's child nodes' values. This is the value of the game.

Enter the value of the game as your answer.



Select the correct answer

1

-1

0

Submit

Note

The limitations of plain search

It may look like we have a method to solve any problem by specifying the states and transitions between them, and finding a path from the current state to our goal. Alas, things get more complicated when we want to apply AI in real world problems. Basically, the number of states in even a moderately complex real-world scenario grows out of hand, and we can't find a solution by exhaustive search ("brute force") or even by using clever heuristics.

Moreover, the transitions which take us from one state to the next when we choose an action are not deterministic. This means that whatever we choose to do will not always completely determine the outcome because there are factors that are beyond our control, and that are often unknown to us.

The algorithms we have discussed above can be adapted to handle some randomness, for example randomness in choosing cards from a shuffled deck or throwing dice. This means that we will need to introduce the concept of uncertainty and probability. Only thus we can begin to approach real-world AI instead of simple puzzles and games. This is the topic of Chapter 3.

After completing Chapter 2 you should be able to:

- Formulate a real-world problem as a search problem
- Formulate a simple game (such as tic-tac-toe) as a game tree
- Use the minimax principle to find optimal moves in a limited-size game tree

Please join the Elements of AI community at Spectrum to discuss and ask questions about this chapter.

You reached the end of Chapter 2!

Correct answers

○ 0 %

Exercises completed

0 /25

Next Chapter

Real world AI

Start →