Tic Tac Toe Non-AI

* Create a list for the game board, initially containing dashes.
* Define functions for displaying the board, checking for wins, and checking for ties.
* In an infinite loop, alternate between the turns of the two players.
* For each turn, prompt the player to choose a position, mark it if available, and display the updated board.
* Check for a win or tie, announce the result, and end the game if necessary.

Magic Square Non-AI

* The algorithm starts by creating an empty n x n matrix.
* It then sets the position of 1 at the middle of the top row.
* The algorithm fills in the matrix by iterating over the numbers from 1 to n^2.
* If a filled cell is encountered, the algorithm places the next number below the previously filled number.
* The algorithm handles edge cases where the next position is out of range or already filled.
* The resulting matrix is returned as the magic square.

Non Queens Non-AI using Backtracking

* Define a function "is\_valid" to check if a queen can be placed on a particular row and column of the chessboard without attacking any other queens.
* Define a recursive function "solve\_n\_queens" to place n queens on an n x n chessboard using backtracking.
* Create an n x n chessboard and call "solve\_n\_queens" with the current column as 0 to place the queens on the board.
* If a valid solution is found, print the board. Otherwise, print a message indicating that no solution exists for the given value of n.
* Define a function "n\_queens" that creates the chessboard and calls "solve\_n\_queens" to solve the n-queens problem.
* Define a main function that takes the value of n as input from the user and calls "n\_queens" to solve the problem.

Water Jug Non-AI BFS @DFS

* Define a starting node with the initial amount in the jugs and a goal amount to be reached.
* Create a queue and append the starting node to it.
* While the queue is not empty, remove the first element from the queue and check if it is the goal node.
* If it is, print the path and exit.
* If not, generate all possible next moves from the current node and append them to the queue.
* Continue until the queue is empty or the goal is found.

8 Puzzle Using Hill Climbing

Define a class EightPuzzle with methods to represent and manipulate a puzzle state.

Define a hill\_climbing algorithm that takes an EightPuzzle object as input and returns the solution state using a local search heuristic.

The hill\_climbing algorithm works by repeatedly choosing the best available state and moving to it until a solution is found or a local maxima or plateau is reached.

The EightPuzzle class has a heuristic method that calculates the Manhattan distance between the current state and the goal state.

The get\_successors method returns all possible successor states that can be reached from the current state by moving the empty tile in one of four directions.

The hill\_climbing algorithm terminates when it reaches the goal state or a local maxima or plateau.

8 Puzzle A\*

Define a class Puzzle

Initialize the class variables board, goal, startX, startY, queue and generatedBoards

Define a method calcHeuristic to calculate the heuristic value of the current board state

Define a method getValidMoves to get all the valid moves that can be played in the current board state

Define a method playMove to play a move on the current board state and return the new board state

Define a method astar to implement the A\* algorithm for solving the 8-puzzle problem

Append the initial board state to the queue and generatedBoards list

Loop until the maximum number of steps is reached

Pop the board with the lowest heuristic value from the queue

Get all the valid moves for the current board state

Play each move and add the resulting board state to the queue if it has not already been generated

Sort the queue in decreasing order of heuristic value

If the goal state is reached, print the number of steps taken and exit the program

If the maximum number of steps is reached, return None

City Distance A\*

Define a Graph class that stores a graph as a dictionary and can add edges between nodes

Define a Node class that stores a node's name, parent, g, h, and f values

Implement the A\* algorithm that takes a graph, heuristics, start, and end nodes

Create an open list and a closed list and add the start node to the open list

While the open list is not empty, sort it and get the current node from the top

Add the current node to the closed list

If the current node is the goal node, return the path from start to end

Get the neighbors of the current node and add them to the open list with their respective g, h, and f values

If a neighbor is already in the open list and its f value is less than the current neighbor's f value, skip it

Return None if there is no path from start to end

A\* Robot Navigation

Initialize the class with the required values and create an empty queue and visited list

Define a method to calculate manhattan distance and print the distance between start and goal position

Define a method to get the neighbours of the current position

Define a method for A\* search algorithm with the following steps:

While the queue is not empty, do the following:

Get the element with the highest priority from the queue

If the element's position is the goal position, print the steps taken and exit

If not, mark the element as visited and get its neighbors

Add the neighbors to the queue in decreasing order of priority

Define a method to print the table

Create an instance of the Robot class and call the necessary methods

Print the current state of the table

Call the A\* search method

Map Coloring

Define a class named map\_coloring.

Initialize the colors and the map.

Define a function to print the map.

Define a function to check if a color can be used for a state without conflicts.

Define a function to assign a color to a state.

Start the main function to solve the map coloring problem.

Print the map.

Assign a color to each state based on the neighbors and available colors.

Print the final solution of the map coloring problem.

CryptArithmetic

Initialize is\_solved and solution\_count to False and 0, respectively.

Take inputs for word1, word2, and result.

Create a list of unique letters from word1, word2, and result.

If the number of unique letters is greater than 10, print "No Solution" and exit.

Call the solve() method with the list of unique letters, empty values, visited array, and the equation array.

In the solve() method, check if all the letters have been assigned a value.

If all the letters have been assigned a value, create a letter\_value\_map to map each letter to its assigned value.

Check if the first letter of word1, word2, and result have been assigned the value 0. If so, return.

Compute the value of word1 and word2 using the letter\_value\_map, and check if their sum is equal to the value of result.

If the sum is equal, increment solution\_count, print the solution, and set is\_solved to True

CrossWord Puzzle

Define function printMatrix to print the matrix and its dimensions.

Define function checkHorizontal to check if a word can be placed horizontally in the matrix and return the modified matrix.

Define function checkVertical to check if a word can be placed vertically in the matrix and return the modified matrix.

Define function solvePuzzle to recursively solve the crossword puzzle by iterating through all possible positions for each word.

If the index is less than the length of the list of words, get the current word and calculate the maximum length possible for the word in the matrix.

For each row and column, try placing the word vertically and recursively call solvePuzzle if the placement is valid.

For each row and column, try placing the word horizontally and recursively call solvePuzzle if the placement is valid.

If the index is equal to the length of the list of words, print the final matrix.

Define the main function to initialize the matrix, words and call the solvePuzzle function.

Call the main function if the script is being run directly.

Tic-Tac-Toe MiniMax Algorithm

The code defines a Tic Tac Toe game that can be played between a user and a computer.

The board is represented as a list of integers.

The ConstBoard function is used to display the current state of the board.

The User1Turn function allows the user to make a move on the board.

The minimax function is used to implement the computer's move using the minmax algorithm.

The CompTurn function makes the computer's move.

The analyzeboard function is used to analyze the game state and check for a winner.

The main function is used to initialize the game and manage the turns of the players.

The game ends when a player wins or there is a draw.

The outcome of the game is displayed at the end.