|  |
| --- |
| 1. Genetic Algorithms for Solving the Traveling Salesman Problem   import random  def distance(c1, c2):      return ((c1[0] - c2[0])\*\*2 + (c1[1] - c2[1])\*\*2)\*\*0.5  def route\_distance(route, cities):      return sum(distance(cities[route[i]], cities[route[i + 1]]) for i in range(len(route) - 1))  def generate\_population(pop\_size, cities):      return [random.sample(list(cities.keys()), len(cities)) for \_ in range(pop\_size)]  def crossover\_and\_mutate(p1, p2, mutation\_rate=0.1):      start, end = sorted(random.sample(range(len(p1)), 2))      child = p1[start:end] + [c for c in p2 if c not in p1[start:end]]        if random.random() < mutation\_rate:          idx1, idx2 = random.sample(range(len(child)), 2)          child[idx1], child[idx2] = child[idx2], child[idx1]          print(f"Mutation occurred: Swapped cities at indices {idx1} and {idx2}")      return child  def evolve\_population(population, cities):        p1, p2 = random.sample(population, 2)        child = crossover\_and\_mutate(p1, p2)        worst\_route = max(population, key=lambda route: route\_distance(route, cities))      population.remove(worst\_route)      population.append(child)  def run\_genetic\_algorithm(cities, pop\_size, num\_iterations):        population = generate\_population(pop\_size, cities)      for \_ in range(num\_iterations):          evolve\_population(population, cities)        best\_route = min(population, key=lambda route: route\_distance(route, cities))      best\_distance = route\_distance(best\_route, cities)      return best\_route, best\_distance  if \_\_name\_\_ == "\_\_main\_\_":      cities = {'A': (0, 0), 'B': (5, 2), 'C': (6, 3), 'D': (3, 4), 'E': (2, 5)}      pop\_size = 50      num\_iterations = 100      best\_route, best\_distance = run\_genetic\_algorithm(cities, pop\_size, num\_iterations)      print("Best route:", best\_route)      print("Total distance:", best\_distance) |
| 1. Use Local Search Algorithms for Solving the N-Queens Problem   import random  def conflicts(board, row, col):        count = 0      for i in range(len(board)):          if i != row:              if board[i] == col or abs(i - row) == abs(board[i] - col):                  count += 1      return count  def evaluate(board):        total\_conflicts = 0      for row in range(len(board)):          total\_conflicts += conflicts(board, row, board[row])      return total\_conflicts  def random\_board(size):        return [random.randint(0, size-1) for \_ in range(size)]  def local\_search(size, max\_iter=1000):        board = random\_board(size)      for \_ in range(max\_iter):          current\_conflicts = evaluate(board)          if current\_conflicts == 0:              return board          row = random.randint(0, size-1)          col = board[row]          current\_conflicts -= conflicts(board, row, col)          min\_conflicts = float('inf')          best\_move = col          for new\_col in range(size):              if new\_col != col:                  new\_conflicts = conflicts(board, row, new\_col)                  if new\_conflicts < min\_conflicts:                      min\_conflicts = new\_conflicts                      best\_move = new\_col          if min\_conflicts < current\_conflicts:              board[row] = best\_move      return None  # No solution found within max\_iter  size = 8  solution = local\_search(size)  if solution:      print("Solution found:")      for row in solution:          print(" . " \* row + " Q " + " . " \* (size - row - 1))  else:      print("No solution found within maximum iterations.") |
| 1. Depth-First Search for Solving the Tower of Hanoi Problem   def tower\_of\_hanoi(n,source,target,auxiliary):      if n == 1 :          print(f"move disk 1 from {source} to {target}")          return      tower\_of\_hanoi(n-1,source,auxiliary,target)      print(f"move  disk {n} from {source} to {target} ")      tower\_of\_hanoi(n-1,auxiliary,target,source)  while True:      try:          num\_disks=int(input("enter the number of disks\n"))          if num\_disks > 0:              print("valid input")              break          else :              print("invalid input")      except ValueError:          print("enter numerical value dumbass")  tower\_of\_hanoi(num\_disks,'A','B','C') |
| 1. Breadth-First Search for Solving the Tower of Hanoi Problem-   from collections import deque  def print\_move(source, target, disk):      print(f"Move disk {disk} from {source} to {target}")  def hanoi(n, source, target, auxiliary):        queue = deque([(source, target, auxiliary)])      while queue:          source, target, auxiliary = queue.popleft()            if len(target) == n:              return source, target, auxiliary            if source and (not auxiliary or source[-1] < auxiliary[-1]):              print\_move('source', 'auxiliary', source[-1])              new\_source, new\_auxiliary = source[:-1], auxiliary + [source[-1]]              queue.append((new\_source, target, new\_auxiliary))            if source and (not target or source[-1] < target[-1]):              print\_move('source', 'target', source[-1])              new\_source, new\_target = source[:-1], target + [source[-1]]              queue.append((new\_source, new\_target, auxiliary))            if auxiliary and (not target or auxiliary[-1] < target[-1]):              print\_move('auxiliary', 'target', auxiliary[-1])              new\_auxiliary, new\_target = auxiliary[:-1], target + [auxiliary[-1]]              queue.append((source, new\_target, new\_auxiliary))  source = [3, 2, 1]  target = []  auxiliary = []  hanoi(3, source, target, auxiliary) |
| 1. A\* Search for Solving the Eight Puzzle Problem   import heapq  def manhattan\_distance(puzzle, goal):      distance = 0      for i in range(9):          if puzzle[i] == 0 or goal[i] == 0:              continue          x1, y1 = divmod(i, 3)          x2, y2 = divmod(puzzle.index(goal[i]), 3)          distance += abs(x1 - x2) + abs(y1 - y2)      return distance  def is\_solvable(puzzle, goal):      puzzle = [i for i in puzzle if i != 0]      inversions = 0      for i in range(len(puzzle) - 1):          for j in range(i + 1, len(puzzle)):              if puzzle[i] > puzzle[j]:                  inversions += 1      print("Number of inversions: ",inversions)      return inversions % 2 == 0  def solve\_puzzle(puzzle, goal):      if not is\_solvable(puzzle, goal):          return [], []      heap = []      heapq.heappush(heap, (0, puzzle, []))      visited = set()      while heap:          cost, current, path = heapq.heappop(heap)          if current == goal:              return cost, path          index = current.index(0)          x, y = divmod(index, 3)          for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):              x2, y2 = x + dx, y + dy              if 0 <= x2 < 3 and 0 <= y2 < 3:                  next\_puzzle = list(current)                  next\_index = x2 \* 3 + y2                  next\_puzzle[index], next\_puzzle[next\_index] = next\_puzzle[next\_index], next\_puzzle[index]                  next\_path = path + [(x, y, x2, y2)]                  next\_cost = cost + 1 + manhattan\_distance(next\_puzzle, goal)                  if tuple(next\_puzzle) not in visited:                      visited.add(tuple(next\_puzzle))                      heapq.heappush(heap, (next\_cost, next\_puzzle, next\_path))      return [], []  if \_\_name\_\_ == '\_\_main\_\_':      puzzle = [1, 2, 3, 7, 4, 6, 0, 5, 8]      goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]      cost, path = solve\_puzzle(puzzle, goal)      if cost:          print('\nFound solution with cost', cost)          print('\nStart State:')          for i in range(3):              print(puzzle[i\*3:i\*3+3])          for step, move in enumerate(path, start=1):              x1, y1, x2, y2 = move              puzzle[x1\*3+y1], puzzle[x2\*3+y2] = puzzle[x2\*3+y2], puzzle[x1\*3+y1]              print(f'\nStep {step}:')              for i in range(3):                  print(puzzle[i\*3:i\*3+3])      else:          print('No solution found') |
| 1. Iterative Deepening Depth-First Search for Solving the Eight Puzzle Problem   from collections import deque  # Function to check if the puzzle is solved  def is\_goal(state):      goal\_state = [[1, 2, 3],                    [4, 5, 6],                    [7, 8, 0]]      return state == goal\_state  # Function to find the possible moves  def possible\_moves(state):      moves = []      for i in range(3):          for j in range(3):              if state[i][j] == 0:                  if i > 0:                      moves.append((i, j, i - 1, j))                  if i < 2:                      moves.append((i, j, i + 1, j))                  if j > 0:                      moves.append((i, j, i, j - 1))                  if j < 2:                      moves.append((i, j, i, j + 1))      return moves  # Function to apply a move to the state  def apply\_move(state, move):      new\_state = [row[:] for row in state]      i, j, new\_i, new\_j = move      new\_state[i][j], new\_state[new\_i][new\_j] = new\_state[new\_i][new\_j], new\_state[i][j]      return new\_state  # Function for iterative deepening depth-first search  def iddfs(start\_state):      depth = 0      while True:          result = depth\_limited\_dfs(start\_state, depth)          if result is not None:              return result, depth          depth += 1  # Function for depth-limited DFS  def depth\_limited\_dfs(state, depth\_limit):      stack = deque([(state, [])])      while stack:          current\_state, path = stack.pop()          if is\_goal(current\_state):              return path          if len(path) < depth\_limit:              for move in possible\_moves(current\_state):                  new\_state = apply\_move(current\_state, move)                  stack.append((new\_state, path + [move]))      return None  # Function to print the puzzle state  def print\_puzzle(state):      for row in state:          print(row)  # Example usage  if \_\_name\_\_ == "\_\_main\_\_":      initial\_state = [[1, 2, 3],                       [0, 4, 6],                       [7, 5, 8]]      print("Start State:")      print\_puzzle(initial\_state)      print("\nSolving...")      solution, cost = iddfs(initial\_state)      if solution:          print("\nFound solution with cost", cost)          step = 1          current\_state = initial\_state          for move in solution:              print("\nStep {}:".format(step))              step += 1              current\_state = apply\_move(current\_state, move)              print\_puzzle(current\_state)      else:          print("No solution found.") |
| 1. Uniform Cost Search for Solving the Eight Puzzle Problem   import heapq  # Define the goal state  goal\_state = tuple(range(1, 9)) + (0,)  # Define a function to find the possible moves from a given state  def find\_moves(state):      moves = []      empty\_idx = state.index(0)      if empty\_idx not in [0, 1, 2]:  # Up          new\_state = list(state)          new\_state[empty\_idx], new\_state[empty\_idx - 3] = new\_state[empty\_idx - 3], new\_state[empty\_idx]          moves.append((tuple(new\_state), "Up"))      if empty\_idx not in [0, 3, 6]:  # Left          new\_state = list(state)          new\_state[empty\_idx], new\_state[empty\_idx - 1] = new\_state[empty\_idx - 1], new\_state[empty\_idx]          moves.append((tuple(new\_state), "Left"))      if empty\_idx not in [2, 5, 8]:  # Right          new\_state = list(state)          new\_state[empty\_idx], new\_state[empty\_idx + 1] = new\_state[empty\_idx + 1], new\_state[empty\_idx]          moves.append((tuple(new\_state), "Right"))      if empty\_idx not in [6, 7, 8]:  # Down          new\_state = list(state)          new\_state[empty\_idx], new\_state[empty\_idx + 3] = new\_state[empty\_idx + 3], new\_state[empty\_idx]          moves.append((tuple(new\_state), "Down"))      return moves  # Define a function to calculate the cost of a move  def move\_cost(state1, state2):      return 1  # Uniform cost for each move  # Uniform Cost Search algorithm  def uniform\_cost\_search(initial\_state):      frontier = [(0, initial\_state, [])]  # Priority queue sorted by path cost      explored = set()  # Set to keep track of explored states      while frontier:          path\_cost, current\_state, path = heapq.heappop(frontier)          if current\_state == goal\_state:              return path          explored.add(current\_state)          for move, move\_dir in find\_moves(current\_state):              if move not in explored:                  new\_cost = path\_cost + move\_cost(current\_state, move)                  new\_path = path + [(move\_dir, move)]                  heapq.heappush(frontier, (new\_cost, move, new\_path))      return None  # No solution found  # Function to print state  def print\_state(state):      for i in range(3):          print(state[i \* 3:i \* 3 + 3])  # Example usage  if \_\_name\_\_ == "\_\_main\_\_":      initial\_state = (1, 2, 3, 0, 4, 6, 7, 5, 8)  # Example initial state      solution = uniform\_cost\_search(initial\_state)      if solution:          print("Start State:")          print\_state(initial\_state)          step\_count = 0          for step, (direction, state) in enumerate(solution, 1):              step\_count += 1              print("\nStep", step\_count, ":", direction)              print\_state(state)          print("\nCost to solve the Eight Puzzle Problem:", len(solution))      else:          print("No solution found.") |
| 1. Heuristic Search Algorithms for Solving the Missionaries and Cannibals Problem   from queue import PriorityQueue  def heuristic(state):      m\_left, c\_left, b\_pos, m\_right, c\_right = state      return (m\_left + c\_left - 2) // 2 + (m\_right + c\_right - 2) // 2  def is\_valid(state):      m\_left, c\_left, b\_pos, m\_right, c\_right = state      if m\_left < 0 or c\_left < 0 or m\_right < 0 or c\_right < 0:          return False      if m\_left > 3 or c\_left > 3 or m\_right > 3 or c\_right > 3:          return False      if (c\_left > m\_left > 0) or (c\_right > m\_right > 0):          return False      return True  def next\_states(state):      m\_left, c\_left, b\_pos, m\_right, c\_right = state      if b\_pos == 'left':          moves = [(2, 0), (0, 2), (1, 1), (1, 0), (0, 1)]          next\_states = [(m\_left-m, c\_left-c, 'right', m\_right+m, c\_right+c) for m, c in moves]      else:          moves = [(-2, 0), (0, -2), (-1, -1), (-1, 0), (0, -1)]          next\_states = [(m\_left+m, c\_left+c, 'left', m\_right-m, c\_right-c) for m, c in moves]      return [state for state in next\_states if is\_valid(state)]  def a\_star(start\_state):      frontier = PriorityQueue()      frontier.put((heuristic(start\_state), [start\_state]))      explored = set()        while not frontier.empty():          path = frontier.get()[1]          current\_state = path[-1]            if current\_state == (0, 0, 'right', 3, 3):              return path            for next\_state in next\_states(current\_state):              if next\_state not in explored:                  new\_path = path + [next\_state]                  frontier.put((len(new\_path) + heuristic(next\_state), new\_path))                  explored.add(next\_state)        return None  path = a\_star((3, 3, 'left', 0, 0))  for state in path:      print(state) |
| 1. Use Breadth-First Search (BFS) for solving the Missionaries and Cannibals problem:   from collections import deque  def get\_possible\_moves(state):      left\_missionaries, left\_cannibals, right\_missionaries, right\_cannibals, boat\_position = state      possible\_moves = []        transitions = [          (1, 0),          (2, 0),          (0, 1),          (0, 2),          (1, 1),      ]        for transition in transitions:          m, c = transition          if boat\_position == 'left':              new\_left\_m = left\_missionaries - m              new\_left\_c = left\_cannibals - c              new\_right\_m = right\_missionaries + m              new\_right\_c = right\_cannibals + c              new\_boat\_position = 'right'          else:              new\_left\_m = left\_missionaries + m              new\_left\_c = left\_cannibals + c              new\_right\_m = right\_missionaries - m              new\_right\_c = right\_cannibals - c              new\_boat\_position = 'left'            if (              new\_left\_m >= 0              and new\_left\_c >= 0              and new\_right\_m >= 0              and new\_right\_c >= 0              and (new\_left\_m == 0 or new\_left\_m >= new\_left\_c)              and (new\_right\_m == 0 or new\_right\_m >= new\_right\_c)          ):              possible\_moves.append(                  (                      new\_left\_m,                      new\_left\_c,                      new\_right\_m,                      new\_right\_c,                      new\_boat\_position,                  )              )      return possible\_moves  def bfs(start, goal):        queue = deque([(start, [])])      visited = set()      while queue:          current\_state, path = queue.popleft()          if current\_state == goal:              return path          if current\_state not in visited:              visited.add(current\_state)                for new\_state in get\_possible\_moves(current\_state):                  if new\_state not in visited:                      new\_path = path + [new\_state]                      queue.append((new\_state, new\_path))      return None  if \_\_name\_\_ == "\_\_main\_\_":        start\_state = (3, 3, 0, 0, 'left')        goal\_state = (0, 0, 3, 3, 'right')        solution = bfs(start\_state, goal\_state)      if solution:          print("Breadth-First Search Solution:")          for step, state in enumerate(solution):              print(f"Step {step}: {state}")      else:          print("No solution found.") |
| 1. Use Depth-First Search (DFS) for solving the Missionaries and Cannibals problem:   def get\_possible\_moves(state):      left\_missionaries, left\_cannibals, right\_missionaries, right\_cannibals, boat\_position = state      possible\_moves = []      transitions = [          (1, 0),          (2, 0),          (0, 1),          (0, 2),          (1, 1),      ]        for transition in transitions:          m, c = transition          if boat\_position == 'left':              new\_left\_m = left\_missionaries - m              new\_left\_c = left\_cannibals - c              new\_right\_m = right\_missionaries + m              new\_right\_c = right\_cannibals + c              new\_boat\_position = 'right'          else:              new\_left\_m = left\_missionaries + m              new\_left\_c = left\_cannibals + c              new\_right\_m = right\_missionaries - m              new\_right\_c = right\_cannibals - c              new\_boat\_position = 'left'            if (              new\_left\_m >= 0              and new\_left\_c >= 0              and new\_right\_m >= 0              and new\_right\_c >= 0              and (new\_left\_m == 0 or new\_left\_m >= new\_left\_c)              and (new\_right\_m == 0 or new\_right\_m >= new\_right\_c)          ):              possible\_moves.append(                  (                      new\_left\_m,                      new\_left\_c,                      new\_right\_m,                      new\_right\_c,                      new\_boat\_position,                  )              )      return possible\_moves  def dfs(state, goal, path, visited):      if state == goal:          return path      visited.add(state)        for new\_state in get\_possible\_moves(state):          if new\_state not in visited:              new\_path = path + [new\_state]              result = dfs(new\_state, goal, new\_path, visited)              if result:                  return result      return None  if \_\_name\_\_ == "\_\_main\_\_":      start\_state = (3, 3, 0, 0, 'left')      goal\_state = (0, 0, 3, 3, 'right')      visited = set()      path = [start\_state]        solution = dfs(start\_state, goal\_state, path, visited)      if solution:          print("Depth-First Search Solution:")          for step, state in enumerate(solution):              print(f"Step {step}: {state}")      else:          print("No solution found.") |
| 1. Use Iterative Deepening Depth-First Search (IDDFS)for solving the Missionaries and Cannibals problem:   import copy  def get\_possible\_moves(state):      left\_missionaries, left\_cannibals, right\_missionaries, right\_cannibals, boat\_position = state      possible\_moves = []      transitions = [          (1, 0),          (2, 0),          (0, 1),          (0, 2),          (1, 1),      ]        for transition in transitions:          m, c = transition          if boat\_position == 'left':              new\_left\_m = left\_missionaries - m              new\_left\_c = left\_cannibals - c              new\_right\_m = right\_missionaries + m              new\_right\_c = right\_cannibals + c              new\_boat\_position = 'right'          else:              new\_left\_m = left\_missionaries + m              new\_left\_c = left\_cannibals + c              new\_right\_m = right\_missionaries - m              new\_right\_c = right\_cannibals - c              new\_boat\_position = 'left'            if (              new\_left\_m >= 0              and new\_left\_c >= 0              and new\_right\_m >= 0              and new\_right\_c >= 0              and (new\_left\_m == 0 or new\_left\_m >= new\_left\_c)              and (new\_right\_m == 0 or new\_right\_m >= new\_right\_c)          ):              possible\_moves.append(                  (                      new\_left\_m,                      new\_left\_c,                      new\_right\_m,                      new\_right\_c,                      new\_boat\_position,                  )              )      return possible\_moves  def depth\_limited\_search(state, goal, limit, path):      if state == goal:          return path      if len(path) >= limit:          return None        for new\_state in get\_possible\_moves(state):          if new\_state not in path:              new\_path = path + [new\_state]              result = depth\_limited\_search(new\_state, goal, limit, new\_path)              if result:                  return result      return None  def iddfs(start, goal, max\_depth):        for depth in range(1, max\_depth + 1):          result = depth\_limited\_search(start, goal, depth, [start])          if result:              return result      return None  if \_\_name\_\_ == "\_\_main\_\_":        start\_state = (3, 3, 0, 0, 'left')        goal\_state = (0, 0, 3, 3, 'right')        solution = iddfs(start\_state, goal\_state, 20)      if solution:          print("Solution found:")          for step, state in enumerate(solution):              print(f"Step {step}: {state}")      else:          print("No solution found within the depth limit.") |
| 1. Use Uniform Cost Search (UCS)for solving the Missionaries and Cannibals problem: Brignt   import heapq  def get\_possible\_moves(state):      left\_missionaries, left\_cannibals, right\_missionaries, right\_cannibals, boat\_position = state      possible\_moves = []        transitions = [          (1, 0),          (2, 0),          (0, 1),          (0, 2),          (1, 1),      ]        for transition in transitions:          m, c = transition          if boat\_position == 'left':              new\_left\_m = left\_missionaries - m              new\_left\_c = left\_cannibals - c              new\_right\_m = right\_missionaries + m              new\_right\_c = right\_cannibals + c              new\_boat\_position = 'right'          else:              new\_left\_m = left\_missionaries + m              new\_left\_c = left\_cannibals + c              new\_right\_m = right\_missionaries - m              new\_right\_c = right\_cannibals - c              new\_boat\_position = 'left'              if (              new\_left\_m >= 0              and new\_left\_c >= 0              and new\_right\_m >= 0              and new\_right\_c >= 0              and (new\_left\_m == 0 or new\_left\_m >= new\_left\_c)              and (new\_right\_m == 0 or new\_right\_m >= new\_right\_c)          ):              possible\_moves.append(                  (                      new\_left\_m,                      new\_left\_c,                      new\_right\_m,                      new\_right\_c,                      new\_boat\_position,                  )              )      return possible\_moves  def ucs(start, goal):        queue = []      heapq.heappush(queue, (0, start, []))      visited = set()      while queue:          cost, current\_state, path = heapq.heappop(queue)          if current\_state == goal:              return path          if current\_state not in visited:              visited.add(current\_state)                for new\_state in get\_possible\_moves(current\_state):                  if new\_state not in visited:                      new\_path = path + [new\_state]                      heapq.heappush(queue, (cost + 1, new\_state, new\_path))      return None  if \_\_name\_\_ == "\_\_main\_\_":        start\_state = (3, 3, 0, 0, 'left')        goal\_state = (0, 0, 3, 3, 'right')        solution = ucs(start\_state, goal\_state)      if solution:          print("Uniform Cost Search Solution:")          for step, state in enumerate(solution):              print(f"Step {step}: {state}")      else:          print("No solution found.") |
| 1. Use Greedy Best-First Search for solving the Missionaries and Cannibals problem   import heapq  import random  def heuristic(state):      left\_missionaries, left\_cannibals, right\_missionaries, right\_cannibals, boat\_position = state      return left\_missionaries + left\_cannibals  def get\_possible\_moves(state):      left\_missionaries, left\_cannibals, right\_missionaries, right\_cannibals, boat\_position = state      possible\_moves = []        transitions = [          (1, 0),          (2, 0),          (0, 1),          (0, 2),          (1, 1),      ]        for transition in transitions:          m, c = transition          if boat\_position == 'left':              new\_left\_m = left\_missionaries - m              new\_left\_c = left\_cannibals - c              new\_right\_m = right\_missionaries + m              new\_right\_c = right\_cannibals + c              new\_boat\_position = 'right'          else:              new\_left\_m = left\_missionaries + m              new\_left\_c = left\_cannibals + c              new\_right\_m = right\_missionaries - m              new\_right\_c = right\_cannibals - c              new\_boat\_position = 'left'            if (              new\_left\_m >= 0              and new\_left\_c >= 0              and new\_right\_m >= 0              and new\_right\_c >= 0              and (new\_left\_m == 0 or new\_left\_m >= new\_left\_c)              and (new\_right\_m == 0 or new\_right\_m >= new\_right\_c)          ):              possible\_moves.append(                  (                      new\_left\_m,                      new\_left\_c,                      new\_right\_m,                      new\_right\_c,                      new\_boat\_position,                  )              )      return possible\_moves  def gbfs(start, goal):        queue = []      heapq.heappush(queue, (heuristic(start), start, []))      visited = set()      while queue:          \_, current\_state, path = heapq.heappop(queue)          if current\_state == goal:              return path          if current\_state not in visited:              visited.add(current\_state)                for new\_state in get\_possible\_moves(current\_state):                  if new\_state not in visited:                      new\_path = path + [new\_state]                      heapq.heappush(queue, (heuristic(new\_state), new\_state, new\_path))      return None  if \_\_name\_\_ == "\_\_main\_\_":        start\_state = (3, 3, 0, 0, 'left')        goal\_state = (0, 0, 3, 3, 'right')        solution = gbfs(start\_state, goal\_state)      if solution:          print("Greedy Best-First Search Solution:")          for step, state in enumerate(solution):              print(f"Step {step}: {state}")      else:          print("No solution found.") |
| 1. Use A\* Search for solving the Missionaries and Cannibals problem   import heapq  def heuristic(state):      left\_missionaries, left\_cannibals, \_, \_, \_ = state      return left\_missionaries + left\_cannibals  # Total people left to cross  def get\_possible\_moves(state):      left\_missionaries, left\_cannibals, right\_missionaries, right\_cannibals, boat\_position = state      possible\_moves = []        transitions = [          (1, 0),          (2, 0),          (0, 1),          (0, 2),          (1, 1),      ]        for transition in transitions:          m, c = transition          if boat\_position == 'left':              new\_left\_m = left\_missionaries - m              new\_left\_c = left\_cannibals - c              new\_right\_m = right\_missionaries + m              new\_right\_c = right\_cannibals + c              new\_boat\_position = 'right'          else:              new\_left\_m = left\_missionaries + m              new\_left\_c = left\_cannibals + c              new\_right\_m = right\_missionaries - m              new\_right\_c = right\_cannibals - c              new\_boat\_position = 'left'            if (              new\_left\_m >= 0              and new\_left\_c >= 0              and new\_right\_m >= 0              and new\_right\_c >= 0              and (new\_left\_m == 0 or new\_left\_m >= new\_left\_c)              and (new\_right\_m == 0 or new\_right\_m >= new\_right\_c)          ):              possible\_moves.append(                  (                      new\_left\_m,                      new\_left\_c,                      new\_right\_m,                      new\_right\_c,                      new\_boat\_position,                  )              )      return possible\_moves  def astar(start, goal):        queue = []      heapq.heappush(queue, (0, 0, start, [start]))      visited = set()      while queue:          total\_cost, path\_cost, current\_state, path = heapq.heappop(queue)          if current\_state == goal:              return path          if current\_state not in visited:              visited.add(current\_state)                for new\_state in get\_possible\_moves(current\_state):                  if new\_state not in visited:                      new\_path\_cost = path\_cost + 1                      total\_cost = new\_path\_cost + heuristic(new\_state)                      new\_path = path + [new\_state]                      heapq.heappush(queue, (total\_cost, new\_path\_cost, new\_state, new\_path))      return None  if \_\_name\_\_ == "\_\_main\_\_":        start\_state = (3, 3, 0, 0, 'left')        goal\_state = (0, 0, 3, 3, 'right')        solution = astar(start\_state, goal\_state)      if solution:          print("A\* Search Solution:")          for step, state in enumerate(solution):              print(f"Step {step}: {state}")      else:          print("No solution found.") |
| 1. Water Jug Problem solving by using production system approach   from collections import deque  def fill\_first\_jug(x, y, a):      return (a, y)  def fill\_second\_jug(x, y, b):      return (x, b)  def empty\_first\_jug(x, y):      return (0, y)  def empty\_second\_jug(x, y):      return (x, 0)  def pour\_from\_second\_to\_first(x, y, a, b):      return (min(x + y, a), max(0, x + y - a))  def pour\_from\_first\_to\_second(x, y, a, b):      return (max(0, x + y - b), min(x + y, b))  # BFS function  def bfs(initial\_state, goal\_state, a, b):      queue = deque([(initial\_state, [initial\_state])])      visited = set()      while queue:          state, path = queue.popleft()          if state == goal\_state:              return path          if state in visited:              continue          visited.add(state)          x, y = state          if x < a:              queue.append((fill\_first\_jug(x, y, a), path + [fill\_first\_jug(x, y, a)]))          if y < b:              queue.append((fill\_second\_jug(x, y, b), path + [fill\_second\_jug(x, y, b)]))          if x > 0:              queue.append((empty\_first\_jug(x, y), path + [empty\_first\_jug(x, y)]))          if y > 0:              queue.append((empty\_second\_jug(x, y), path + [empty\_second\_jug(x, y)]))          if y > 0:              queue.append((pour\_from\_second\_to\_first(x, y, a, b), path + [pour\_from\_second\_to\_first(x, y, a, b)]))          if x > 0:              queue.append((pour\_from\_first\_to\_second(x, y, a, b), path + [pour\_from\_first\_to\_second(x, y, a, b)]))      return False  def main():      initial\_state = (0, 0)      goal\_state = (4, 0)      a = 5      b = 3      result = bfs(initial\_state, goal\_state, a, b)      if result:          print("Goal state is reachable")          print("Steps:")          for step in result:              print(step)      else:          print("Goal state is not reachable")  if \_\_name\_\_ == '\_\_main\_\_':      main() |
| 1. Tic Tac Toe game implementation by Magic Square Method   import random  def print\_board(board):      for row in board:          print(" | ".join(row))          print("-" \* 13)  def is\_winner(board, player):      for row in board:          if all(cell == player for cell in row):              return True      for col in range(3):          if all(board[row][col] == player for row in range(3)):              return True      if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):          return True      return False  def is\_board\_full(board):      return all(cell != ' ' for row in board for cell in row)  def get\_user\_move():      while True:          try:              move = int(input("Enter your move (1-9): "))              if 1 <= move <= 9:                  return move              else:                  print("Invalid move. Please enter a number between 1 and 9.")          except ValueError:              print("Invalid input. Please enter a number.")  def calculate\_computer\_move(board, player\_symbol, computer\_symbol):      magic\_square = [          [8, 3, 4],          [1, 5, 9],          [6, 7, 2]      ]      empty\_cells = [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']      for i, j in empty\_cells:          temp\_board = [row[:] for row in board]          temp\_board[i][j] = computer\_symbol          if is\_winner(temp\_board, computer\_symbol):              return i \* 3 + j + 1      for i, j in empty\_cells:          temp\_board = [row[:] for row in board]          temp\_board[i][j] = player\_symbol          if is\_winner(temp\_board, player\_symbol):              return i \* 3 + j + 1      return random.choice(empty\_cells)[0] \* 3 + random.choice(empty\_cells)[1] + 1  def play\_tic\_tac\_toe():      board = [[' ' for \_ in range(3)] for \_ in range(3)]      user\_symbol, computer\_symbol = 'X', 'O'      print("Welcome to Tic-Tac-Toe using Magic Square technique!")      print\_board(board)      for move\_num in range(1, 10):          current\_player = user\_symbol if move\_num % 2 == 1 else computer\_symbol          if current\_player == user\_symbol:              user\_move = get\_user\_move()              row, col = divmod(user\_move - 1, 3)          else:              computer\_move = calculate\_computer\_move(board, user\_symbol, computer\_symbol)              row, col = divmod(computer\_move - 1, 3)              print(f"Computer chooses position {computer\_move}")          while board[row][col] != ' ':              print("ERROR! That position is already taken. Choose a different one.")              if current\_player == user\_symbol:                  user\_move = get\_user\_move()                  row, col = divmod(user\_move - 1, 3)              else:                  computer\_move = calculate\_computer\_move(board, user\_symbol, computer\_symbol)                  row, col = divmod(computer\_move - 1, 3)          board[row][col] = user\_symbol if current\_player == user\_symbol else computer\_symbol          print\_board(board)          if is\_winner(board, current\_player):              print(f"{current\_player} wins!")              break          if is\_board\_full(board):              print("It's a tie!")              break  play\_tic\_tac\_toe() |
| 1. Tic Tac Toe Problem solving by using Adversarial Search approach.   from random import choice  from math import inf  board = [[0, 0, 0],           [0, 0, 0],           [0, 0, 0]]  def Gameboard(board):      chars = {1: 'X', -1: 'O', 0: ' '}      for x in board:          for y in x:              ch = chars[y]              print(f'| {ch} |', end='')          print('\n' + '---------------')      print('===============')  def Clearboard(board):      for x, row in enumerate(board):          for y, col in enumerate(row):              board[x][y] = 0  def winningPlayer(board, player):      conditions = [[board[0][0], board[0][1], board[0][2]],                       [board[1][0], board[1][1], board[1][2]],                       [board[2][0], board[2][1], board[2][2]],                       [board[0][0], board[1][0], board[2][0]],                       [board[0][1], board[1][1], board[2][1]],                       [board[0][2], board[1][2], board[2][2]],                       [board[0][0], board[1][1], board[2][2]],                       [board[0][2], board[1][1], board[2][0]]]      if [player, player, player] in conditions:          return True      return False  def gameWon(board):      return winningPlayer(board, 1) or winningPlayer(board, -1)  def printResult(board):      if winningPlayer(board, 1):          print('X has won! ' + '\n')      elif winningPlayer(board, -1):          print('O\'s have won! ' + '\n')      else:          print('Draw' + '\n')  def blanks(board):      blank = []      for x, row in enumerate(board):          for y, col in enumerate(row):              if board[x][y] == 0:                  blank.append([x, y])      return blank  def boardFull(board):      if len(blanks(board)) == 0:          return True      return False  def setMove(board, x, y, player):      board[x][y] = player  def playerMove(board):      e = True      moves = {1: [0, 0], 2: [0, 1], 3: [0, 2],               4: [1, 0], 5: [1, 1], 6: [1, 2],               7: [2, 0], 8: [2, 1], 9: [2, 2]}      while e:          try:              move = int(input('Enter a number between 1-9: '))              if move < 1 or move > 9:                  print('Invalid Move! Try again!')              elif not (moves[move] in blanks(board)):                  print('Invalid Move! Try again!')              else:                  setMove(board, moves[move][0], moves[move][1], 1)                  Gameboard(board)                  e = False          except(KeyError, ValueError):              print('Enter a number!')  def getScore(board):      if winningPlayer(board, 1):          return 10      elif winningPlayer(board, -1):          return -10      else:          return 0  def abminimax(board, depth, alpha, beta, player):      row = -1      col = -1      if depth == 0 or gameWon(board):          return [row, col, getScore(board)]      else:          for cell in blanks(board):              setMove(board, cell[0], cell[1], player)              score = abminimax(board, depth - 1, alpha, beta, -player)              if player == 1:                  # X is always the max player                  if score[2] > alpha:                      alpha = score[2]                      row = cell[0]                      col = cell[1]              else:                  if score[2] < beta:                      beta = score[2]                      row = cell[0]                      col = cell[1]              setMove(board, cell[0], cell[1], 0)              if alpha >= beta:                  break          if player == 1:              return [row, col, alpha]          else:              return [row, col, beta]  def o\_comp(board):      if len(blanks(board)) == 9:          x = choice([0, 1, 2])          y = choice([0, 1, 2])          setMove(board, x, y, -1)          Gameboard(board)      else:          result = abminimax(board, len(blanks(board)), -inf, inf, -1)          setMove(board, result[0], result[1], -1)          Gameboard(board)  def x\_comp(board):      if len(blanks(board)) == 9:          x = choice([0, 1, 2])          y = choice([0, 1, 2])          setMove(board, x, y, 1)          Gameboard(board)      else:          result = abminimax(board, len(blanks(board)), -inf, inf, 1)          setMove(board, result[0], result[1], 1)          Gameboard(board)  def makeMove(board, player, mode):      if mode == 1:          if player == 1:              playerMove(board)          else:              o\_comp(board)      else:          if player == 1:              o\_comp(board)          else:              x\_comp(board)  def pvc():      while True:          try:              order = int(input('Enter to play 1st or 2nd: '))              if not (order == 1 or order == 2):                  print('Please pick 1 or 2')              else:                  break          except(KeyError, ValueError):              print('Enter a number')      Clearboard(board)      if order == 2:          currentPlayer = -1      else:          currentPlayer = 1      while not (boardFull(board) or gameWon(board)):          makeMove(board, currentPlayer, 1)          currentPlayer \*= -1      printResult(board)  print("TIC-TAC-TOE using MINIMAX with ALPHA-BETA Pruning")  pvc() |