8 PUZZLE

```
from heapq import heappush, heappop
goal = (1, 2, 3, 4, 5, 6, 7, 8, 0)        # target board
# Manhattan-distance heuristic
def h(s):
    return sum(abs((v-1)//3 - i//3) + abs((v-1)%3 - i%3)
               for i, v in enumerate(s) if v)
# neighbouring boards after one slide
def nbrs(s):
    i = s.index(0); r, c = divmod(i, 3)
    for dr, dc in ((1,0), (-1,0), (0,1), (0,-1)):
        nr, nc = r+dr, c+dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            j = nr*3 + nc
            t = list(s); t[i], t[j] = t[j], t[i]
            yield tuple(t)
def astar(start):
    pq, seen = [(h(start), 0, start, [])], {start}
    while pq:
        f, g, s, path = heappop(pq)
        if s == goal:
            return path                    # solved
        for n in nbrs(s):
            if n not in seen:
                seen.add(n)
                heappush(pq, (g+1+h(n), g+1, n, path+[n]))
start = tuple(map(int, input("Enter 9 numbers (0 = blank): ").split()))
solution = astar(start)
print(f"\nSolved in {len(solution)} moves:")
for state in solution:
    print(state[:3], "\n", state[3:6], "\n", state[6:], "\n")
```

Python – Map Coloring Problem (Backtracking)

```
colors = ['Red', 'Green', 'Blue']
neighbors = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'E'],
    'D': ['B', 'E'],
    'E': ['C', 'D']
}
def is_valid(state, node, color):
    for neighbor in neighbors[node]:
        if state.get(neighbor) == color:
            return False
    return True
def solve(state, nodes):
    if not nodes:
        return state
    node = nodes[0]
    for color in colors:
        if is_valid(state, node, color):
            state[node] = color
            result = solve(state, nodes[1:])
            if result:
                return result
            state.pop(node)
    return None
solution = solve({}, list(neighbors.keys()))
print("Coloring:", solution)
```

Python – Alpha Beta Pruning

```
def alphabeta(node, depth, alpha, beta, maximizingPlayer, values):
    if depth == 0:
```

```
        return values[node]
    if maximizingPlayer:
        maxEval = float('-inf')
        for child in node * 2 + 1, node * 2 + 2:
            if child < len(values):
                eval = alphabeta(child, depth - 1, alpha, beta, False, values)
                maxEval = max(maxEval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
        return maxEval
    else:
        minEval = float('inf')
        for child in node * 2 + 1, node * 2 + 2:
            if child < len(values):
                eval = alphabeta(child, depth - 1, alpha, beta, True, values)
                minEval = min(minEval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
        return minEval
tree_values = [3, 5, 6, 9, 1, 2, 0, -1]
result = alphabeta(0, 3, float('-inf'), float('inf'), True, tree_values)
print("Alpha Beta Result:", result)
```

Python – TSP using Brute Force

```
from itertools import permutations
cities = ['A', 'B', 'C', 'D']
distances = {
    ('A', 'B'): 10, ('A', 'C'): 15, ('A', 'D'): 20,
    ('B', 'C'): 35, ('B', 'D'): 25,
    ('C', 'D'): 30
```

```
}
def distance(path):
   dist = 0
   for i in range(len(path) - 1):
      dist += distances.get((path[i], path[i+1]), distances.get((path[i+1], path[i]), 0))
   dist += distances.get((path[-1], path[0]), distances.get((path[0], path[-1]), 0))
   return dist
min_path, min_cost = None, float('inf')
for perm in permutations(cities):
   d = distance(perm)
   if d < min_cost:
      min_cost = d
      min_path = perm
print("Shortest path:", min_path)
print("Distance:", min_cost)
```

Python – Depth First Search

```
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': [],
   'D': [],
   'E': ['F'],
   'F': []}
visited = set()
def dfs(node):
   if node not in visited:
      print(node)
      visited.add(node)
      for neighbor in graph[node]:
         dfs(neighbor)
dfs('A')
```

Python – Breadth First Search

```
from collections import deque
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
def bfs(start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print(vertex, end=' ')
            visited.add(vertex)
            queue.extend(graph[vertex])
bfs('A')
```

HTML – Web Page with Meta, Title, Anchor Tags

```
<!DOCTYPE html>
<html>
<head>
    <title>Online Shopping</title>
    <meta name="description" content="Best online shopping site">
</head>
<body>
    <h1>Welcome to My Shop</h1>
    <a href="products.html">View Products</a>
</body></html>
```

Python – A Search*

```
from queue import PriorityQueue
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('D', 2), ('E', 5)],
    'C': [('E', 1), ('F', 3)],
    'D': [], 'E': [], 'F': []
}
heuristics = {'A': 7, 'B': 6, 'C': 4, 'D': 1, 'E': 2, 'F': 0}
def a_star(start, goal):
    pq = PriorityQueue()
    pq.put((0 + heuristics[start], 0, start, [start]))
    visited = set()
    while not pq.empty():
        est_total, cost, node, path = pq.get()
        if node == goal:
            print("Path:", path)
            print("Cost:", cost)
            return
        if node not in visited:
            visited.add(node)
            for neighbor, weight in graph.get(node, []):
                pq.put((cost + weight + heuristics[neighbor], cost + weight, neighbor, path + [neighbor]))
a_star('A', 'F')
```

Python – Sudoku Solver using Backtracking

```
def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(start_row, start_row + 3):
```

```
        for j in range(start_col, start_col + 3):
            if board[i][j] == num:
                return False
    return True
def solve(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                for num in range(1, 10):
                    if is_valid(board, i, j, num):
                        board[i][j] = num
                        if solve(board):
                            return True
                        board[i][j] = 0
                return False
    return True
def print_board(board):
    for row in board:
        print(row)
board = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]
if solve(board):
```

```
    print_board(board)
else:
    print("No solution exists")
```

Python – Bidirectional Search

```
from collections import deque
graph = {
    1: [2, 3],
    2: [4],
    3: [5],
    4: [],
    5: [6],
    6: []
}
def bfs(start, goal):
    front = {start}
    back = {goal}
    visited = set()
    while front and back:
        if front & back:
            return True
        visited.update(front)
        next_front = set()
        for node in front:
            for neighbor in graph.get(node, []):
                if neighbor not in visited:
                    next_front.add(neighbor)
        front = next_front
        front, back = back, front  # swap
    return False
print("Found?" if bfs(1, 6) else "Not Found")
```

Python – 8 Queens Problem

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or \
           board[i] - i == col - row or \
           board[i] + i == col + row:
            return False
    return True
def solve_queens(n, row=0, board=[]):
    if row == n:
        print("Solution:", board)
        return
    for col in range(n):
        if is_safe(board, row, col):
            solve_queens(n, row + 1, board + [col])
solve_queens(8)
```

Python – Water Jug Problem (4L and 3L jugs, get 2L in 4L)

```
from collections import deque
def get_successors(state):
    a, b = state
    return set([
        (4, b), (a, 3), (0, b), (a, 0),
        (a - min(a, 3 - b), b + min(a, 3 - b)),  # A->B
        (a + min(b, 4 - a), b - min(b, 4 - a))   # B->A
    ])
def bfs():
    visited = set()
    queue = deque([((0, 0), [])])
    while queue:
        (a, b), path = queue.popleft()
        if (a, b) in visited:
```

```
continue
        visited.add((a, b))
        path = path + [(a, b)]
        if a == 2:
            for state in path:
                print(state)
            return
        for next_state in get_successors((a, b)):
            queue.append((next_state, path))
bfs()
```

Python – Cryptarithm: SEND + MORE = MONEY

```
from constraint import *
problem = Problem()
letters = "SENDMORY"
problem.addVariables(letters, range(10))
problem.addConstraint(AllDifferentConstraint())
def valid(s, e, n, d, m, o, r, y):
    send = 1000*s + 100*e + 10*n + d
    more = 1000*m + 100*o + 10*r + e
    money = 10000*m + 1000*o + 100*n + 10*e + y
    return send + more == money and s != 0 and m != 0
problem.addConstraint(valid, letters)
solutions = problem.getSolutions()
for s in solutions:
    print(s)
```

Python – Missionaries and Cannibals

```
def is_valid(state):
    m1, c1, b, m2, c2 = state
    return (0 <= m1 <= 3 and 0 <= c1 <= 3 and
            0 <= m2 <= 3 and 0 <= c2 <= 3 and
            (m1 == 0 or m1 >= c1) and
```

```
        (m2 == 0 or m2 >= c2))
def get_moves(state):
    m1, c1, b, m2, c2 = state
    moves = [(1,0),(2,0),(0,1),(0,2),(1,1)]
    results = []
    for m, c in moves:
        if b == 1:
            new_state = (m1 - m, c1 - c, 0, m2 + m, c2 + c)
        else:
            new_state = (m1 + m, c1 + c, 1, m2 - m, c2 - c)
        if is_valid(new_state):
            results.append(new_state)
    return results
def solve():
    from collections import deque
    start = (3, 3, 1, 0, 0)
    goal = (0, 0, 0, 3, 3)
    queue = deque([(start, [])])
    visited = set()
    while queue:
        state, path = queue.popleft()
        if state == goal:
            print("Solution:", path + [state])
            return
        for move in get_moves(state):
            if move not in visited:
                visited.add(move)
                queue.append((move, path + [state]))
solve()
```

Python – Tic Tac Toe Game

```
board = [' ' for _ in range(9)]
```

```
def print_board():
    for i in range(3):
        print(board[i*3], '|', board[i*3+1], '|', board[i*3+2])
def is_winner(player):
    wins = [(0,1,2),(3,4,5),(6,7,8),
            (0,3,6),(1,4,7),(2,5,8),
            (0,4,8),(2,4,6)]
    return any(board[a]==board[b]==board[c]==player for a,b,c in wins)
def play():
    turn = 'X'
    for _ in range(9):
        print_board()
        move = int(input(f"Enter position for {turn} (0-8): "))
        if board[move] == ' ':
            board[move] = turn
            if is_winner(turn):
                print_board()
                print(f"{turn} wins!")
                return
            turn = 'O' if turn == 'X' else 'X'
        else:
            print("Invalid move.")
    print("It's a draw!")
# Uncomment to play:
# play()
```