Waterjug

```
from collections import deque
def water_jug_bfs(cap_a, cap_b, target):
    visited = set()
    queue = deque([((0, 0), [])])
    while queue:
        (a, b), path = queue.popleft()
        if a == target or b == target:
            path.append((a, b))
            return path
        if (a, b) in visited:
            continue
        visited.add((a, b))
        queue.append(((cap_a, b), path + [(cap_a, b)]))

        queue.append(((a, cap_b), path + [(a, cap_b)]))

        queue.append(((0, b), path + [(0, b)]))

        queue.append(((a, 0), path + [(a, 0)]))

        new_a = a - min(a, cap_b - b)
        new_b = b + min(a, cap_b - b)
        queue.append(((new_a, new_b), path + [(new_a, new_b)]))
        new_a = a + min(b, cap_a - a)
        new_b = b - min(b, cap_a - a)
        queue.append(((new_a, new_b), path + [(new_a, new_b)]))

    return "No solution found"
```

```
cap_a = 4
cap_b = 3
target = 2

solution_path = water_jug_bfs(cap_a, cap_b, target)
print("Steps to reach the target amount of water:")
for step in solution_path:
    print(step)
```

minmax

```
def minimax(depth, index, is_max, values, alpha, beta):
    if depth == 3:
        return values[index]
    if is_max:
        max_eval = float('-inf')
        for i in range(2):
            eval = minimax(depth + 1, index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):
            eval = minimax(depth + 1, index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
```

```
                break
        return min_eval
values = [3, 5, 6, 9, 1, 2, 0, -1]
print("Optimal value:", minimax(0, 0, True, values, float('-inf'), float('inf')))
```

maze

```
import random

WALL, PATH = 1, 0


class MazeGenerator:
    def __init__(self, width, height):
        self.width, self.height = width, height
        self.maze = [[WALL] * width for _ in range(height)]


    def generate_maze(self, r, c):
        self.maze[r][c] = PATH
        for dy, dx in random.sample([(0, 2), (0, -2), (2, 0), (-2, 0)], 4):
            nr, nc = r + dy, c + dx
            if 0 < nr < self.height and 0 < nc < self.width and self.maze[nr][nc] == WALL:
                self.maze[nr][nc] = self.maze[r + dy // 2][c + dx // 2] = PATH
                self.generate_maze(nr, nc)
    def print_maze(self):
        for row in self.maze:
            print(''.join('#' if cell == WALL else ' ' for cell in row))
maze = MazeGenerator(7, 7)
maze.generate_maze(1, 1)
maze.print_maze()
```

dfs

```
def dfs_maze(maze, start, end):
    rows, cols = len(maze), len(maze[0])
    path = []
    visited = set()
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    def dfs(x, y):
        if x < 0 or y < 0 or x >= rows or y >= cols or maze[x][y] == 1 or (x, y) in visited:
            return False
        visited.add((x, y))
        path.append((x, y))
        if (x, y) == end:
            return True
        for dx, dy in directions:
            if dfs(x + dx, y + dy):
                return True
        path.pop()
        return False
    if dfs(*start):
        maze_with_path = [row[:] for row in maze]
        for x, y in path:
            maze_with_path[x][y] = "*"

        print("Maze with path from start to end:")
        for row in maze_with_path:
            print(" ".join(str(cell) if cell != "*" else "*" for cell in row))
    else:
        print("No path found")
maze = [
    [0, 1, 0, 0, 0],
```

```
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]
start = (0, 0)
end = (4, 4)
dfs_maze(maze, start, end)
```

8 queens

```
import random
def calculate_heuristic(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def get_best_move(board, col):
    n = len(board)
    min_conflicts = calculate_heuristic(board)
    best_row = board[col]
    for row in range(n):
        if row != board[col]:
            new_board = list(board)
            new_board[col] = row
            conflicts = calculate_heuristic(new_board)
```

```
            if conflicts < min_conflicts:
                min_conflicts = conflicts
                best_row = row
    return best_row
def solve_n_queens(n=8):
    board = [random.randint(0, n - 1) for _ in range(n)]
    while True:
        heuristic = calculate_heuristic(board)
        if heuristic == 0:
            return board
        for col in range(n):
            best_row = get_best_move(board, col)
            board[col] = best_row
        if calculate_heuristic(board) >= heuristic:
            board = [random.randint(0, n - 1) for _ in range(n)]
def print_board(board):
    n = len(board)
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()
solution = solve_n_queens()
print("One possible solution for the 8-Queens problem:")
print_board(solution)
```

8 puzzle

```
import heapq

goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = divmod(state[i][j] - 1, 3)
                distance += abs(x - i) + abs(y - j)
    return distance

def get_neighbors(state):
    neighbors = []
    x, y = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors
```

```
def a_star(start):
    queue = [(manhattan_distance(start), 0, start, [])]
    visited = set()

    while queue:
        cost, steps, state, path = heapq.heappop(queue)

        state_tuple = tuple(tuple(row) for row in state)

        if state == goal_state:
            return path + [state]

        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        for neighbor in get_neighbors(state):
            new_path = path + [state]
            heapq.heappush(queue, (steps + 1 + manhattan_distance(neighbor), steps + 1, neighbor,
new_path))

    return None
def print_state(state):
    for row in state:
        print(row)
    print()
start_state = [
    [1, 2, 3],
    [5, 0, 6],
```

```
    [4, 7, 8]
]

solution = a_star(start_state)
print("Steps to solve the puzzle:")
for step in solution:
    print_state(step)
```