Homework4

孙皓宇 202418019427074

问题描述

八数码问题（8-Puzzle Problem）

八数码问题是一个经典的智力游戏，也是一个NP完全问题。游戏的目标是通过滑动数字块，将一个3x3的板上的数字从初始状态移动到目标状态。板上有8个数字块和一个空格，数字块可以滑动到空格的位置，但不能跳过其他数字块。每个数字块只能移动到相邻的空格位置。

特点：

- **初始状态**：一个3x3的板上的数字被打乱，空格（通常用0表示）位于某个位置。
- **目标状态**：板上的数字按照1到8的顺序排列，空格位于最后的位置。
- **操作**：只能将空格旁边的数字块（上、下、左、右）移动到空格的位置。

示例：
初始状态：

```
1 2 3
4 0 5
7 8 6
```

目标状态：

```
1 2 3
4 5 6
7 8 0
```

八皇后问题（8-Queens Problem）

八皇后问题是一个古老的国际象棋问题，也是一个NP完全问题。问题的目标是在8x8的棋盘上放置8个皇后，使得它们互不攻击。在国际象棋中，皇后可以攻击同一行、同一列或对角线上的任何棋子。

特点：

- **棋盘**：一个8x8的棋盘。
- **目标**：在棋盘上放置8个皇后，使得任意两个皇后都不在同一行、同一列或同一对角线上。
- **操作**：在棋盘的空格上放置皇后，确保不违反互不攻击的规则。

示例：
一个可能的解决方案：

```
Q . . . . . . .
. . . . . Q . .
. . . . . . Q .
. Q . . . . . .
. . . . Q . . .
. . . . . . . Q
. . Q . . . . .
. . . . Q . . .
```

在这里，Q代表皇后，.代表空格。

这两个问题都是组合优化问题，它们涉及到在有限的搜索空间中寻找最优解。八数码问题和八皇后问题都可以通过启发式搜索算法来求解，尽管可能无法保证总是找到最优解，但这些算法能够在合理的时间内找到一个可行的解或者接近最优的解。

算法介绍

1. 爬山法（Hill Climbing）

爬山法是一种局部搜索算法，用于在给定的搜索空间中寻找最优解。它从初始解开始，逐步移动到相邻的解，如果新解比当前解更优（代价函数值更低），则接受新解，否则保持当前解不变。

最陡上升爬山法：

- 在每一步，算法会评估当前解的所有邻居解，并选择代价函数值最低的邻居解作为下一步的当前解。
- 如果所有邻居解都不如当前解，算法可能会陷入局部最优解。

首选爬山法：

- 类似于最陡上升爬山法，但在选择邻居解时，如果存在多个代价函数值相同的邻居解，算法会随机选择其中一个。
- 这种方法可以增加搜索的多样性，减少陷入局部最优解的风险。

工作原理：

1. 从初始解开始。
2. 评估所有邻居解的代价函数值。
3. 选择代价函数值最低的邻居解作为新的当前解。
4. 重复步骤2和3，直到达到终止条件（如达到最大迭代次数或找到最优解）。

2. 随机重启爬山法（Random Restart Hill Climbing）

随机重启爬山法是对爬山法的改进，它通过随机重启来避免陷入局部最优解。

工作原理：

1. 运行爬山法直到找到局部最优解。
2. 随机选择一个新的初始解，重新开始爬山法。
3. 重复步骤1和2，直到达到终止条件（如达到最大迭代次数或找到最优解）。

这种方法通过随机重启增加了搜索的多样性，有助于跳出局部最优解，提高找到全局最优解的概率。

3. 模拟退火算法（Simulated Annealing）

模拟退火算法是一种概率性算法，它受到物理学中退火过程的启发。算法允许以一定的概率接受更差的解，以此来跳出局部最优解。

工作原理：

1. 从初始解开始，设置一个初始温度T。
2. 在当前解的邻居解中随机选择一个新解。
3. 计算新解的代价函数值ΔE（新解的代价函数值减去当前解的代价函数值）。
4. 如果ΔE < 0（新解更优），接受新解。
5. 如果ΔE ≥ 0（新解更差），以概率e^(-ΔE/T)接受新解。
6. 降低温度T（例如，T = αT，其中0 < α < 1）。
7. 重复步骤2-6，直到达到终止条件（如温度降到足够低或达到最大迭代次数）。

模拟退火算法的关键参数是温度T和降温速率α。初始温度较高时，算法更可能接受较差的解，随着温度的降低，接受较差解的概率逐渐降低，搜索过程逐渐趋于局部搜索。

这三种算法各有特点，爬山法简单直观，但容易陷入局部最优解；随机重启爬山法通过重启增加了搜索的多样性；模拟退火算法通过概率性地接受较差解，有助于跳出局部最优解，但计算成本较高。在实际应用中，可以根据问题的特点和计算资源选择合适的算法。

实验设计

1. 生成问题实例

八数码问题实例生成：

- 初始状态可以通过随机打乱1到8的数字，并保留一个空格（0）来生成。例如，从一个有序状态开始，随机选择一个非空格的数字块，将其与空格交换位置，重复这个过程多次，直到达到一个随机状态。

八皇后问题实例生成：

- 由于八皇后问题的目标是找到一个解决方案，而不是从一个初始状态移动到另一个状态，因此不需要生成初始状态。我们可以直接在空棋盘上开始搜索。

2. 算法实现细节

爬山法实现细节：

- **初始化**：选择一个初始解作为起点。
- **邻居生成**：对于八数码问题，可以通过将空格（0）向上、向下、向左或向右移动来生成邻居解。对于八皇后问题，可以通过在棋盘上随机放置一个皇后并检查是否满足条件来生成邻居解。
- **代价函数评估**：对于八数码问题，代价函数可以是与目标状态的差异度量，如曼哈顿距离之和。对于八皇后问题，代价函数可以是受到攻击的皇后对数。

- **选择和更新**：选择代价函数值最低的邻居解作为新的当前解，并重复这个过程，直到无法找到更好的解。

随机重启爬山法实现细节：

- **重启机制**：在每次达到局部最优解后，随机生成一个新的初始解，并重新开始爬山法。
- **参数设置**：设置重启次数或最大迭代次数作为终止条件。

模拟退火算法实现细节：

- **温度调度**：设置初始温度和降温速率。
- **邻居选择**：随机选择一个邻居解，并计算其代价函数值。
- **接受准则**：如果新解更好，则接受；如果新解更差，则以一定的概率接受，这个概率随着温度的降低而降低。
- **温度降低**：在每次迭代后降低温度，直到达到终止条件。

通用实现细节：

- **数据结构**：使用数组或矩阵来表示八数码问题的状态，使用二维数组来表示八皇后问题的状态。
- **效率优化**：在生成邻居解时，避免重复计算，利用缓存等技术提高效率。
- **终止条件**：设置最大迭代次数、达到一定质量的解或运行时间限制作为算法的终止条件。

3. 实验参数和设置

- **实例数量**：生成足够多的实例以确保结果的统计意义。
- **算法参数**：为每种算法设置合适的参数，如初始温度、降温速率、重启次数等。
- **性能度量**：记录每次运行的搜索耗散（如迭代次数、CPU时间）和是否找到最优解。
- **统计分析**：对多次运行的结果进行统计分析，计算平均搜索耗散和问题的解决率。

实现代码

八数码问题

```
import random
import math

acc1 = 0
acc2 = 0
acc3 = 0
acc4 = 0

# 生成一个初始的8数码状态，0表示空格
def generate_8puzzle():
    puzzle = list(range(1, 9)) + [0]
    for _ in range(100):  # 打乱100次
        idx1, idx2 = random.sample(range(9), 2)
        puzzle[idx1], puzzle[idx2] = puzzle[idx2], puzzle[idx1]
    return puzzle

# 打印八数码问题的状态
```

```python
def print_puzzle(puzzle):
    for i in range(0, len(puzzle), 3):
        print(" ".join(str(x) if x != 0 else ' ' for x in puzzle[i:i+3]))

# 计算曼哈顿距离作为代价函数
def manhattan_distance(puzzle, goal):
    distance = 0
    for i in range(9):
        if puzzle[i] != 0:
            correct_row, correct_col = divmod(goal.index(puzzle[i]), 3)
            current_row, current_col = divmod(i, 3)
            distance += abs(correct_row - current_row) + abs(correct_col -
current_col)
    return distance

# 生成一个新的邻居状态
def get_neighbors(puzzle):
    zero_index = puzzle.index(0)
    row, col = divmod(zero_index, 3)
    neighbors = []
    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  # 上下左右移动
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_puzzle = list(puzzle)
            new_puzzle[zero_index], new_puzzle[new_row * 3 + new_col] =
new_puzzle[new_row * 3 + new_col], new_puzzle[zero_index]
            neighbors.append(new_puzzle)
    return neighbors

# 最陡上升爬山法
def steepest_ascent_hill_climbing(puzzle, goal, max_iterations=100):
    current_puzzle = list(puzzle)
    best_puzzle = list(puzzle)
    best_distance = manhattan_distance(puzzle, goal)
    iterations = 0
    while iterations < max_iterations:
        neighbors = get_neighbors(current_puzzle)
        best_neighbor = None
        min_distance = float('inf')
        for neighbor in neighbors:
            distance = manhattan_distance(neighbor, goal)
            if distance < min_distance:
                best_neighbor = neighbor
                min_distance = distance
        if best_neighbor is not None and min_distance < best_distance:
            best_distance = min_distance
            best_puzzle = list(best_neighbor)
            current_puzzle = best_neighbor
        else:
            break
        iterations += 1
    return best_puzzle, iterations
```

```python
# 首选爬山法
def first_choice_hill_climbing(puzzle, goal, max_iterations=100):
    current_puzzle = list(puzzle)
    best_puzzle = list(puzzle)
    best_distance = manhattan_distance(puzzle, goal)
    iterations = 0
    while iterations < max_iterations:
        neighbors = get_neighbors(current_puzzle)
        if not neighbors:
            break
        for neighbor in neighbors:
            distance = manhattan_distance(neighbor, goal)
            if distance < best_distance:
                best_distance = distance
                best_puzzle = list(neighbor)
                current_puzzle = list(neighbor)
                break  # 一旦找到更好的邻居，立即接受它
        iterations += 1
    return best_puzzle, iterations

# 随机重启爬山法
def random_restart_hill_climbing(goal, max_restarts=100, max_iterations=100):
    restarts = 0
    while restarts < max_restarts:
        # 从一个随机初始状态开始
        initial_puzzle = generate_8puzzle()
        solution, iterations = steepest_ascent_hill_climbing(initial_puzzle,
goal, max_iterations)

        # 如果找到了目标状态，返回解
        if solution == goal:
            # print(f"Solution found after {restarts + 1} restarts and
{iterations} iterations.")
            return 1

        restarts += 1

    # print("Solution not found within the given restarts.")
    return 0

# 模拟退火算法
def simulated_annealing(goal, initial_temp=10000, min_temp=1, alpha=0.99,
max_iterations=1000):
    current_puzzle = generate_8puzzle()
    current_distance = manhattan_distance(current_puzzle, goal)
    temperature = initial_temp
    iteration = 0

    while temperature > min_temp and iteration < max_iterations:
        neighbors = get_neighbors(current_puzzle)
        new_puzzle = random.choice(neighbors)
        new_distance = manhattan_distance(new_puzzle, goal)
```

```
        # 计算接受新状态的概率
        if new_distance < current_distance:
            current_puzzle = new_puzzle
            current_distance = new_distance
        else:
            delta = current_distance - new_distance
            acceptance_prob = math.exp(delta / temperature)
            if random.random() < acceptance_prob:
                current_puzzle = new_puzzle
                current_distance = new_distance

        # 降温
        temperature *= alpha
        iteration += 1

    return current_puzzle, current_distance, iteration


# 模拟退火算法
def simulated_annealing2(goal, initial_temp=1000000, cooling_rate=0.99999,
min_temp=1):
    current_puzzle = generate_8puzzle()
    current_distance = manhattan_distance(current_puzzle, goal)
    best_puzzle = list()
    best_distance = current_distance
    temperature = initial_temp

    while temperature > min_temp:
        neighbor = random.choice(get_neighbors(current_puzzle))
        neighbor_distance = manhattan_distance(neighbor, goal)

        if neighbor_distance < current_distance:  # 如果找到了更好的解
            current_puzzle = neighbor
            current_distance = neighbor_distance
            if neighbor_distance < best_distance:  # 更新最佳解
                best_puzzle = neighbor
                best_distance = neighbor_distance
        elif random.random() < math.exp((current_distance - neighbor_distance)
/ temperature):  # 接受较差的解
            current_puzzle = neighbor
            current_distance = neighbor_distance

        temperature *= cooling_rate  # 降温

    return best_distance == 0

# 八数码问题的目标状态
goal_puzzle = [1, 2, 3, 4, 5, 6, 7, 8, 0]

# # 生成一个初始状态
# initial_puzzle = generate_8puzzle()
# print("Initial Puzzle:")
# print_puzzle(initial_puzzle)
```

```
# # 使用最陡上升爬山法求解
# solution, iterations = steepest_ascent_hill_climbing(initial_puzzle,
goal_puzzle)
# print("\nSolution by Steepest Ascent Hill Climbing:")
# print_puzzle(solution)
# print(f"Iterations: {iterations}")

# # 使用首选爬山法求解
# solution, iterations = first_choice_hill_climbing(initial_puzzle,
goal_puzzle)
# print("\nSolution by First Choice Hill Climbing:")
# print_puzzle(solution)
# print(f"Iterations: {iterations}")

for i in range(10):
    # initial_puzzle = generate_8puzzle()
    # if goal_puzzle == steepest_ascent_hill_climbing(initial_puzzle,
goal_puzzle):
    #     acc1 += 1
    # if goal_puzzle == first_choice_hill_climbing(initial_puzzle,
goal_puzzle):
    #     acc2 += 1
    # acc3 += random_restart_hill_climbing(goal_puzzle)
    # 使用模拟退火算法求解
    # solution, distance, iterations = simulated_annealing(goal_puzzle)

    # print(f"Solution found with final distance {distance} after {iterations}
iterations.")
    # print_puzzle(solution)
    acc4 += simulated_annealing2(goal_puzzle)


# print(acc1/100)
# print(acc2/100)
# print(acc3/100)
print(acc4)
```

八皇后问题

```
import random
import math
import time

# 生成一个随机解
def generate_initial_state(n=8):
    # 随机放置 n 个皇后，每个皇后在不同的列
    return [random.randint(0, n-1) for _ in range(n)]

# 计算当前状态的代价：皇后之间的冲突数
```

```python
def calculate_conflicts(state):
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                conflicts += 1
    return conflicts

# 获取邻居状态：通过在某一列中移动皇后来生成邻居
def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        # 尝试将皇后从当前行移动到其他行
        for row in range(n):
            if row != state[col]:  # 不和当前行重合
                neighbor = state[:]
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

# 最陡上升爬山法（Steepest-Ascent Hill Climbing）
def steepest_ascent_hill_climbing(n=8):
    current_state = generate_initial_state(n)
    current_conflicts = calculate_conflicts(current_state)

    while True:
        neighbors = get_neighbors(current_state)
        next_state = None
        next_conflicts = current_conflicts

        # 选择代价最小的邻居
        for neighbor in neighbors:
            conflict_count = calculate_conflicts(neighbor)
            if conflict_count < next_conflicts:
                next_state = neighbor
                next_conflicts = conflict_count

        # 如果没有找到更优解，退出
        if next_conflicts >= current_conflicts:
            break
        current_state = next_state
        current_conflicts = next_conflicts

    return current_state, current_conflicts

# 首选爬山法（First-Choice Hill Climbing）
def first_choice_hill_climbing(n=8):
    current_state = generate_initial_state(n)
    current_conflicts = calculate_conflicts(current_state)

    while True:
```

```
        neighbors = get_neighbors(current_state)
        random.shuffle(neighbors)  # 随机打乱邻居顺序，增加多样性

        for neighbor in neighbors:
            conflict_count = calculate_conflicts(neighbor)
            if conflict_count < current_conflicts:
                current_state = neighbor
                current_conflicts = conflict_count
                break
        else:
            # 如果没有找到更好的邻居，退出
            break

    return current_state, current_conflicts

# 随机重启爬山法（Random Restart Hill Climbing）
def random_restart_hill_climbing(n=8, max_restarts=100):
    restart_count = 0
    best_state = None
    best_conflicts = float('inf')

    while restart_count < max_restarts:
        restart_count += 1
        # print(f"Restart #{restart_count}")

        # 使用最陡上升爬山法进行搜索
        current_state, current_conflicts = steepest_ascent_hill_climbing(n)

        # 更新全局最优解
        if current_conflicts < best_conflicts:
            best_state = current_state
            best_conflicts = current_conflicts

        # 如果找到了冲突为0的解，提前终止
        if best_conflicts == 0:
            break

    return best_state, best_conflicts

# 模拟退火算法
def simulated_annealing(n=8, initial_temp=1000, cooling_rate=0.995,
max_iterations=10000):
    current_state = generate_initial_state(n)
    current_conflicts = calculate_conflicts(current_state)
    best_state = current_state[:]
    best_conflicts = current_conflicts
    temperature = initial_temp

    iteration = 0
    while iteration < max_iterations and temperature > 0.1:
        neighbors = get_neighbors(current_state)
        next_state = random.choice(neighbors)
        next_conflicts = calculate_conflicts(next_state)
```

```
        # 如果下一个状态更好（冲突更少），或者以一定概率接受较差的解
        if next_conflicts < current_conflicts or random.random() <
math.exp((current_conflicts - next_conflicts) / temperature):
            current_state = next_state
            current_conflicts = next_conflicts

        # 更新最优解
        if current_conflicts < best_conflicts:
            best_state = current_state[:]
            best_conflicts = current_conflicts

        # 降低温度
        temperature *= cooling_rate
        iteration += 1

        # 如果已经找到完美解，提前终止
        if best_conflicts == 0:
            break

    return best_state, best_conflicts

# 打印棋盘状态
def print_board(state):
    n = len(state)
    for row in range(n):
        board_row = ['Q' if col == state[row] else '.' for col in range(n)]
        print(" ".join(board_row))
    print("\n")

# # 运行最陡上升爬山法
# solution, conflicts = steepest_ascent_hill_climbing()
# print("Steepest-Ascent Hill Climbing Solution (conflicts =", conflicts, "):")
# print_board(solution)

# # 运行首选爬山法
# solution, conflicts = first_choice_hill_climbing()
# print("First-Choice Hill Climbing Solution (conflicts =", conflicts, "):")
# print_board(solution)

# # 运行随机重启爬山法
# solution, conflicts = random_restart_hill_climbing()
# print("Random Restart Hill Climbing Solution (conflicts =", conflicts, "):")
# print_board(solution)

# # 运行模拟退火算法
# solution, conflicts = simulated_annealing()
# print("Simulated Annealing Solution (conflicts =", conflicts, "):")
# print_board(solution)

# 用于计时的辅助函数
def time_function(func, *args):
    start_time = time.time()
```

```
    for _ in range(1000):
        result = func(*args)
    end_time = time.time()
    return result, end_time - start_time

# 比较三种方法的执行时间
def compare_methods():
    # 1. 最陡上升爬山法
    result, time_1 = time_function(steepest_ascent_hill_climbing)
    print(f"steepest_ascent_hill_climbing Time: {time_1:.4f} seconds")

    # 2. 首选爬山法
    result, time_2 = time_function(first_choice_hill_climbing)
    print(f"first_choice_hill_climbing Time: {time_2:.4f} seconds")

    # 3. 随机重启爬山法
    result, time_3 = time_function(random_restart_hill_climbing)
    print(f"random_restart_hill_climbing Time: {time_3:.4f} seconds")

     # 4. 模拟退火算法
    result, time_4 = time_function(simulated_annealing)
    print(f"simulated_annealing Time: {time_4:.4f} seconds")

# 运行比较
compare_methods()
```

结果分析

- 对于八数码问题，爬山法（最陡上升和首选爬山法）基本不能正确解决，几乎全部陷入局部最优解；而随机重启爬山法随着重启次数的增加，其求解正确率也在增加，重启次数100次时正确率约为25%，重启次数达到1000次正确率稳定在90%以上；模拟退火算法的正确率也是受初始温度和温度下降率影响，参数合适的情况下，正确率也在90%以上。
- 对于八皇后问题，四种算法都能够很好的解决，正确率方面都是100%，但是运行时间（搜索耗散）上，爬山法最小，随机重启稍大，模拟退火最大。