南開大學
Nankai University

计算机学院

算法导论设计报告

多景点游玩规划设计

姓名：高景珩

学号：2310648

专业：计算机科学与技术

2025 年 6 月 10 日

目录

1 实验背景

在端午节假期，由于假期时间短，可以游玩的景点多，基于这个情况，我设计了一个相关的实验，通过算法导论完成下面的实验。

其实，旅游行程规划是日常生活中常见的优化问题，尤其在时间和预算有限的情况下，如何选择景点并合理安排游玩顺序以获得最佳体验具有实际意义。本实验以景点规划为背景，结合《算法导论》中学习的算法知识，设计一个通用化的多景点旅游行程规划系统。问题需考虑景点的空间位置、门票费用、游玩时间、期待程度、开放时间以及每天的可用时间，目标是最大化总期待程度，同时满足时间和预算约束。

本实验综合运用了图算法（如 Dijkstra 算法、旅行商问题近似解法）、贪心算法、动态规划、分治策略和网络流算法的相关思想，解决从景点选择到路径优化的多阶段问题。通过建模和算法实现，不仅能够解决实际的旅游规划需求，还能展示多种算法在复杂优化问题中的协同应用，为城市旅游、资源调度等领域提供参考。

2 抽象题目条件

本实验将旅游行程规划问题抽象为一个多约束优化问题，目标是最大化总期待程度，满足预算和时间约束。问题描述如下：

- **景点集合**：给定 N 个景点，每个景点 i 包含以下属性：
  - 名称 $place_i$：字符串，表示景点名称。
  - 坐标 (x_i, y_i)：二维平面位置（单位：公里）。
  - 门票价格 M_i：非负浮点数（单位：元）。
  - 游玩时间 t_i：正浮点数（单位：小时）。
  - 期待程度 h_i：浮点数，范围 $[1, 10]$，表示景点吸引力。
  - 开放时间 $[time_{begin,i}, time_{end,i}]$：浮点数（24 小时制，单位：小时）。
- **旅行天数**：D 天，每天的可用时间为 $[day_{begin,j}, day_{end,j}]$（24 小时制，单位：小时）。
- **预算**：总预算 B 元，包含门票费用和交通费用。
- **交通模型**：
  - 景点间距离：基于坐标 (x_i, y_i) 和 (x_j, y_j)，计算欧几里得距离 $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.
  - 交通时间：速度 $v = 30$ 公里/小时，时间 $t_{ij} = d_{ij}/v$ 小时。
  - 交通费用：单位距离费用 $c_d = 2.3$ 元/公里，费用 $c_{ij} = d_{ij} \cdot c_d$ 元。
- **起点**：每天从酒店（坐标 $(0, 0)$）出发并返回。
- **目标**：选择景点子集并规划每天的游玩顺序，最大化总期待程度 $\sum h_i$，满足：
  - 总费用（$\sum M_i + \sum c_{ij} \leq B$).
  - 每天总时间（游玩时间 $\sum t_i$+ 交通时间 $\sum t_{ij} \leq day_{end,j} - day_{begin,j}$).
  - 每个景点的游玩时间在 $[time_{begin,i}, time_{end,i}]$ 内。
  - 每个景点最多访问一次。

3 输入输出设置

3.1 输入

- **第一行**：
  - N：景点数量（正整数）。
  - D：旅行天数（正整数）。
  - B：总预算（非负浮点数，单位：元）。
- **接下来 N 行**：每行描述一个景点 i：
  - $place_i$：景点名称（字符串，无空格）。
  - M_i：门票价格（非负浮点数，单位：元）。
  - t_i：游玩时间（正浮点数，单位：小时）。
  - h_i：期待程度（浮点数，范围 [1, 10]）。
  - $time_{begin,i}$：开门时间（浮点数，24 小时制）。
  - $time_{end,i}$：关门时间（浮点数，24 小时制）。
  - x_i, y_i：坐标（浮点数，单位：公里）。
- **接下来 D 行**：每行描述一天的可用时间：
  - $day_{begin,j}$：第 j 天开始时间（浮点数，24 小时制）。
  - $day_{end,j}$：第 j 天结束时间（浮点数，24 小时制）。
- **交通参数**（隐式）：
  - 速度 $v = 30$ 公里/小时。
  - 单位距离费用 $c_d = 2.3$ 元/公里。

3.2 输出

- **行程计划**：为 D 天规划每天的游玩顺序，包括：
  - 每天访问的景点子集（按顺序输出 $place_i$）。
  - 每天的交通路线（从酒店到景点，再返回酒店）。
  - 每天的时间表（开始时间、游玩时间、交通时间、结束时间）。
  - 总期待程度 $\sum h_i$。
  - 总费用（门票费用 $\sum M_i$+ 交通费用 $\sum c_{ij}$).
- **约束验证**：
  - 总费用不超过 B.
  - 每天总时间不超过 $day_{end,j} - day_{begin,j}$.
  - 景点游玩时间在 $[time_{begin,i}, time_{end,i}]$ 内。
  - 每个景点最多访问一次。

4 程序设计

根据题目要求设计一个满足时间窗口、预算约束和兴趣最大化的行程规划程序，解决思路通过综合运用图结构、贪心选择、分治策略和动态规划优化，构建了一个高效且实用的算法框架。程序首先以景点和酒店（设为坐标原点）作为节点，构建一个完全图，其中边权基于欧几里得距离，用于计算交通时间和费用。这一图结构为后续的路径规划提供了基础，确保距离和时间计算的高效性和准确性。同时，程序通过预计算全局距离矩阵，避免重复计算，进一步提升性能。

核心算法采用分治策略，将多天行程规划分解为每天的独立子问题，每天的任务是在给定时间窗口和剩余预算内选择最优的景点子集并确定访问顺序。为此，程序引入动态规划来处理每天的景点选择，通过状态压缩记录已访问景点的子集状态，并以子集的最后一个景点作为阶段变量，计算最早完成时间，以此判断时间可行性和预算约束。在状态转移过程中，程序枚举当前子集和下一个待访问景点，结合时间窗口约束和交通时间，优化选择最大化兴趣值的路径。

为了进一步提高效率，程序在动态规划选出景点子集后，采用改进的最近邻算法（nearest neighbor）进行路径排序，优先选择距离较近且时间窗口较早关闭的景点，同时通过时间窗口检查确保路径的可行性。这种贪心策略有效减少交通时间，优化每日行程的访问顺序。此外，考虑到实际应用场景，程序限制每天最多访问 5 个景点，并在动态规划后通过回溯重构最优路径，确保结果满足预算和时间约束，同时最大化总兴趣值。

最终，程序输出每一天的详细行程，包括访问顺序、起止时间、游玩和交通时长、费用以及兴趣值，清晰呈现规划结果。这种设计通过图结构建模问题，分治法分解多天任务，动态规划优化景点选择，贪心算法改进路径排序，综合实现了高效、灵活且贴合实际需求的行程规划方案，兼顾计算效率与用户体验。

5 代码实现与测试

按照上面的思路，得到程序最后的代码如下：

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <iomanip>
#include <algorithm>
#include <climits>
#include <bitset>
using namespace std;

// 全局常量定义
const double SPEED = 30.0;          // 旅行速度，单位：公里/小时
const double COST_PER_KM = 2.3;     // 每公里交通费用，单位：元/公里
const double INF = 1e9;             // 无穷大，用于初始化

// 景点结构体：存储景点的基本信息
struct Spot {
    string name;            // 景点名称
    double ticket;          // 门票价格
    double play_time;       // 游玩所需时间（小时）
    double interest;        // 景点的期待值/吸引力
```

```
21      double open_time;       // 开放时间（24小时制）
22      double close_time;      // 关闭时间（24小时制）
23      double x, y;            // 景点的地理坐标
24      bool visited = false;   // 是否已经游玩过
25  };
26
27  // 每日时间窗口结构体
28  struct Day {
29      double start_time;      // 一天的开始时间
30      double end_time;        // 一天的结束时间
31  };
32
33  // 路线结构体：存储规划好的行程信息
34  struct Route {
35      vector<int> spot_indices;  // 景点访问顺序（存储索引）
36      double start_time;         // 行程开始时间
37      double end_time;           // 行程结束时间
38      double play_duration;      // 总游玩时间
39      double travel_duration;    // 总交通时间
40      double cost;               // 总花费
41      double interest;           // 总期待值
42  };
43
44  /*
45   * 计算两点之间的欧几里得距离
46   * @param x1, y1: 第一个点的坐标
47   * @param x2, y2: 第二个点的坐标
48   * @return: 两点间的直线距离
49   */
50  double euclidean_distance(double x1, double y1, double x2, double y2) {
51      return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
52  }
53
54  /*
55   * 根据距离计算交通时间
56   * @param dist: 距离（公里）
57   * @return: 所需时间（小时）
58   */
59  double travel_time(double dist) {
60      return dist / SPEED;
61  }
62
63  /*
64   * 计算交通费用
65   * @param dist: 距离（公里）
66   * @return: 费用（元）
67   */
68  double travel_cost(double dist) {
69      return dist * COST_PER_KM;
```

```
}

// 全局距离矩阵
vector<vector<double>> global_dist;

/*
 * 初始化全局距离矩阵
 * 计算所有景点之间以及起点（酒店）到各景点的距离
 * @param spots: 景点列表
 */
void init_global_dist(const vector<Spot>& spots) {
    int N = spots.size();
    global_dist.assign(N + 1, vector<double>(N + 1));

    // 计算起点（索引0）到所有景点的距离
    for (int i = 0; i < N; ++i) {
        global_dist[0][i + 1] = global_dist[i + 1][0] =
            euclidean_distance(0, 0, spots[i].x, spots[i].y);
    }

    // 计算景点之间的距离
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            global_dist[i + 1][j + 1] =
                euclidean_distance(spots[i].x, spots[i].y, spots[j].x, spots[j].y);
        }
    }
}

/*
 * 检查给定路线是否满足时间窗口约束
 * @param spots: 景点列表
 * @param order: 访问顺序
 * @param day_start: 一天的开始时间
 * @param day_end: 一天的结束时间
 * @return: 是否可行
 */
bool check_time_windows(const vector<Spot>& spots, const vector<int>& order,
    double day_start, double day_end) {
    double current_time = day_start;
    double prev_x = 0, prev_y = 0;

    // 遍历每个景点，检查时间约束
    for (int idx : order) {
        double dist = euclidean_distance(prev_x, prev_y, spots[idx].x, spots[idx].y);
        current_time += travel_time(dist);

        // 检查是否能在景点关闭前到达
        if (current_time > spots[idx].close_time) {
```

```
119             return false;
120         }
121 
122         // 如果早到，需要等待开放
123         if (current_time < spots[idx].open_time) {
124             current_time = spots[idx].open_time;
125         }
126 
127         current_time += spots[idx].play_time;
128 
129         // 检查是否能在景点关闭前游玩完
130         if (current_time > spots[idx].close_time) {
131             return false;
132         }
133 
134         prev_x = spots[idx].x;
135         prev_y = spots[idx].y;
136     }
137 
138     // 检查是否能在规定时间内返回
139     double dist_back = euclidean_distance(prev_x, prev_y, 0, 0);
140     current_time += travel_time(dist_back);
141     return current_time <= day_end;
142 }
143 
144 /*
145  * 使用最近邻算法构建初始路线
146  * @param spots: 景点列表
147  * @param indices: 可选景点的索引
148  * @param day_start: 开始时间
149  * @param day_end: 结束时间
150  * @return: 可行的访问顺序
151  */
152 vector<int> nearest_neighbor(const vector<Spot>& spots, const vector<int>& indices,
153     double day_start, double day_end) {
154     int n = indices.size();
155     vector<bool> visited(n, false);
156     vector<int> order;
157     double current_time = day_start;
158     double prev_x = 0, prev_y = 0;
159 
160     while (order.size() < n) {
161         double min_dist = INF;
162         int next = -1;
163         double earliest_close = INF;
164 
165         // 寻找最近且满足时间窗口的未访问景点
166         for (int i = 0; i < n; ++i) {
167             if (!visited[i]) {
```

```
168                 double dist = euclidean_distance(prev_x, prev_y,
169                     spots[indices[i]].x, spots[indices[i]].y);
170                 double arrive = current_time + travel_time(dist);
171                 double finish = max(arrive, spots[indices[i]].open_time) +
172                     spots[indices[i]].play_time;
173
174                 if (arrive <= spots[indices[i]].close_time && finish <= day_end) {
175                     if (spots[indices[i]].close_time < earliest_close ||
176                         (spots[indices[i]].close_time == earliest_close && dist <
                             min_dist)) {
177                         min_dist = dist;
178                         earliest_close = spots[indices[i]].close_time;
179                         next = i;
180                     }
181                 }
182             }
183         }
184
185         if (next == -1) break;
186
187         visited[next] = true;
188         order.push_back(indices[next]);
189
190         double dist = euclidean_distance(prev_x, prev_y,
191             spots[indices[next]].x, spots[indices[next]].y);
192         current_time = max(current_time + travel_time(dist),
193             spots[indices[next]].open_time) + spots[indices[next]].play_time;
194         prev_x = spots[indices[next]].x;
195         prev_y = spots[indices[next]].y;
196     }
197
198     // 验证并优化路线
199     if (!order.empty() && !check_time_windows(spots, order, day_start, day_end)) {
200         for (int i = order.size(); i > 0; --i) {
201             vector<int> partial_order(order.begin(), order.begin() + i);
202             if (check_time_windows(spots, partial_order, day_start, day_end)) {
203                 return partial_order;
204             }
205         }
206         return {};
207     }
208
209     return order;
210 }
211
212 /*
213  * 核心路线规划函数
214  * 使用动态规划和贪心策略进行多天行程规划
215  * @param N: 景点数量
```

```
 * @param D: 旅行天数
 * @param total_budget: 总预算
 * @param spots: 景点列表
 * @param days: 每天的时间窗口
 * @return: 规划好的路线列表
 */
vector<Route> plan_routes(int N, int D, double total_budget,
    vector<Spot>& spots, vector<Day>& days) {
    vector<Route> routes(D);
    double budget_used = 0.0;

    // 初始化距离矩阵
    init_global_dist(spots);

    // 对每一天进行规划
    for (int d = 0; d < D; ++d) {
        double day_start = days[d].start_time, day_end = days[d].end_time;

        // 筛选当天可行的景点
        vector<int> valid_indices;
        for (int i = 0; i < N; ++i) {
            if (!spots[i].visited &&
                spots[i].ticket <= total_budget - budget_used &&
                spots[i].open_time <= day_end &&
                spots[i].close_time >= day_start) {
                valid_indices.push_back(i);
            }
        }

        int M = valid_indices.size();
        if (M == 0) continue;

        // 预处理状态信息
        vector<double> ticket_sum(1 << M, 0), interest_sum(1 << M, 0),
            play_sum(1 << M, 0);
        for (int mask = 1; mask < (1 << M); ++mask) {
            for (int i = 0; i < M; ++i) {
                if (mask & (1 << i)) {
                    ticket_sum[mask] += spots[valid_indices[i]].ticket;
                    interest_sum[mask] += spots[valid_indices[i]].interest;
                    play_sum[mask] += spots[valid_indices[i]].play_time;
                }
            }
            if (ticket_sum[mask] + budget_used > total_budget) {
                ticket_sum[mask] = INF;
                interest_sum[mask] = -1;
                play_sum[mask] = INF;
            }
        }
```

```
265
266         // 动态规划状态数组
267         vector<vector<double>> dp(1 << M, vector<double>(M, INF));
268         vector<vector<int>> parent(1 << M, vector<int>(M, -1));
269
270         // 计算每个景点的最早到达和最晚离开时间
271         vector<double> earliest_arrival(M), latest_departure(M);
272         for (int i = 0; i < M; ++i) {
273             earliest_arrival[i] = day_start +
274                 travel_time(global_dist[0][valid_indices[i] + 1]);
275             latest_departure[i] = spots[valid_indices[i]].close_time -
276                 spots[valid_indices[i]].play_time;
277         }
278
279         // 初始化单个景点的状态
280         for (int i = 0; i < M; ++i) {
281             double arrive = earliest_arrival[i];
282             double start_play = max(arrive, spots[valid_indices[i]].open_time);
283             double finish_play = start_play + spots[valid_indices[i]].play_time;
284             double travel_cost_to_spot = travel_cost(global_dist[0][valid_indices[i]
                    + 1]);
285             double travel_cost_back = travel_cost(global_dist[valid_indices[i] +
                    1][0]);
286
287             if (finish_play <= spots[valid_indices[i]].close_time &&
288                 finish_play + travel_time(global_dist[valid_indices[i] + 1][0]) <=
                        day_end &&
289                 ticket_sum[1 << i] + travel_cost_to_spot + travel_cost_back <=
290                 total_budget - budget_used) {
291                 dp[1 << i][i] = finish_play;
292                 parent[1 << i][i] = -1;
293             }
294         }
295
296         // 动态规划主循环
297         const int MAX_K = 5;  // 每天最多访问的景点数
298         for (int k = 1; k <= min(M, MAX_K); ++k) {
299             for (int mask = 1; mask < (1 << M); ++mask) {
300                 if (bitset<32>(mask).count() != k || interest_sum[mask] < 0) continue;
301
302                 // 检查当前组合是否可行
303                 vector<int> subset;
304                 for (int i = 0; i < M; ++i) {
305                     if (mask & (1 << i)) subset.push_back(valid_indices[i]);
306                 }
307                 if (!check_time_windows(spots, subset, day_start, day_end)) {
308                     continue;
309                 }
310
```

```
            // 尝试添加新的景点
            for (int u = 0; u < M; ++u) {
                if (!(mask & (1 << u)) || dp[mask][u] == INF) continue;

                for (int v = 0; v < M; ++v) {
                    if (mask & (1 << v)) continue;

                    int next_mask = mask | (1 << v);
                    double travel_t = travel_time(
                        global_dist[valid_indices[u] + 1][valid_indices[v] + 1]);
                    double arrive = dp[mask][u] + travel_t;

                    if (arrive > spots[valid_indices[v]].close_time) continue;

                    double start_play = max(arrive,
                        spots[valid_indices[v]].open_time);
                    double finish_play = start_play +
                        spots[valid_indices[v]].play_time;
                    double travel_cost_to_spot = travel_cost(
                        global_dist[0][valid_indices[u] + 1]);
                    double travel_cost_next = travel_cost(
                        global_dist[valid_indices[u] + 1][valid_indices[v] + 1]);
                    double travel_cost_back = travel_cost(
                        global_dist[valid_indices[v] + 1][0]);
                    double cost = ticket_sum[next_mask] + travel_cost_to_spot +
                        travel_cost_next + travel_cost_back;

                    if (finish_play <= day_end && cost + budget_used <=
                        total_budget &&
                        finish_play < dp[next_mask][v]) {
                        dp[next_mask][v] = finish_play;
                        parent[next_mask][v] = u;
                    }
                }
            }
        }
    }

    // 选择最优解
    double best_interest = -1, best_cost = INF, best_end_time = INF;
    int best_mask = 0, best_last = -1;

    for (int mask = 1; mask < (1 << M); ++mask) {
        if (interest_sum[mask] < 0 || bitset<32>(mask).count() > MAX_K) continue;

        vector<int> subset;
        for (int i = 0; i < M; ++i) {
            if (mask & (1 << i)) subset.push_back(valid_indices[i]);
        }
```

```
357
358             if (!check_time_windows(spots, subset, day_start, day_end)) continue;
359
360             for (int last = 0; last < M; ++last) {
361                 if (dp[mask][last] == INF) continue;
362
363                 double return_travel_t = travel_time(
364                     global_dist[valid_indices[last] + 1][0]);
365                 double finish_time = dp[mask][last] + return_travel_t;
366
367                 if (finish_time <= day_end) {
368                     double travel_cost_to_spot = travel_cost(
369                         global_dist[0][valid_indices[0] + 1]);
370                     double travel_cost_back = travel_cost(
371                         global_dist[valid_indices[last] + 1][0]);
372                     double cost = ticket_sum[mask];
373
374                     for (int i = 1; i < subset.size(); ++i) {
375                         cost += travel_cost(
376                             global_dist[subset[i - 1] + 1][subset[i] + 1]);
377                     }
378                     cost += travel_cost_to_spot + travel_cost_back;
379
380                     if (cost + budget_used <= total_budget &&
381                         interest_sum[mask] > best_interest) {
382                         best_interest = interest_sum[mask];
383                         best_cost = cost;
384                         best_end_time = finish_time;
385                         best_mask = mask;
386                         best_last = last;
387                     }
388                 }
389             }
390         }
391
392         // 如果找不到可行解，继续下一天
393         if (best_interest < 0) {
394             continue;
395         }
396
397         // 重建最优路径
398         vector<int> route_spots;
399         int cur_mask = best_mask, cur_pos = best_last;
400         while (cur_pos != -1) {
401             route_spots.push_back(valid_indices[cur_pos]);
402             int prev_pos = parent[cur_mask][cur_pos];
403             cur_mask ^= (1 << cur_pos);
404             cur_pos = prev_pos;
405         }
```

```
406         reverse(route_spots.begin(), route_spots.end());
407
408         // 使用最近邻算法优化路线
409         route_spots = nearest_neighbor(spots, route_spots, day_start, day_end);
410         if (route_spots.empty()) {
411             continue;
412         }
413
414         // 标记已访问的景点
415         for (int idx : route_spots) {
416             spots[idx].visited = true;
417         }
418
419         // 计算路线详细信息
420         double play_duration = 0, travel_duration = 0;
421         double current_time = day_start, prev_x = 0, prev_y = 0;
422         double cost = 0;
423
424         for (int idx : route_spots) {
425             double dist = euclidean_distance(prev_x, prev_y, spots[idx].x,
                    spots[idx].y);
426             travel_duration += travel_time(dist);
427             current_time = max(current_time + travel_time(dist),
                    spots[idx].open_time);
428             play_duration += spots[idx].play_time;
429             current_time += spots[idx].play_time;
430             cost += spots[idx].ticket;
431             if (prev_x != 0 || prev_y != 0) {
432                 cost += travel_cost(dist);
433             }
434             prev_x = spots[idx].x;
435             prev_y = spots[idx].y;
436         }
437
438         travel_duration += travel_time(euclidean_distance(prev_x, prev_y, 0, 0));
439         cost += travel_cost(euclidean_distance(0, 0,
440             spots[route_spots[0]].x, spots[route_spots[0]].y));
441         cost += travel_cost(euclidean_distance(prev_x, prev_y, 0, 0));
442         best_end_time = current_time +
443             travel_time(euclidean_distance(prev_x, prev_y, 0, 0));
444
445         // 保存当天的路线
446         routes[d] = { route_spots, day_start, best_end_time, play_duration,
447                       travel_duration, cost, best_interest };
448         budget_used += cost;
449     }
450
451     return routes;
452 }
```

```
453
454 /*
455  * 打印规划结果
456  * @param routes: 规划好的路线列表
457  * @param spots: 景点列表
458  */
459 void print_routes(const vector<Route>& routes, const vector<Spot>& spots) {
460     double total_cost = 0, total_interest = 0;
461
462     cout << fixed << setprecision(2);
463     cout << "\n========== 行程规划结果 ==========" << endl;
464
465     for (int d = 0; d < routes.size(); ++d) {
466         cout << "第" << d + 1 << "天行程: " << endl;
467
468         if (routes[d].spot_indices.empty()) {
469             cout << "无可行景点行程" << endl;
470         }
471         else {
472             cout << "行程路线: 酒店 -> ";
473             for (int i = 0; i < routes[d].spot_indices.size(); ++i) {
474                 cout << spots[routes[d].spot_indices[i]].name;
475                 if (i != routes[d].spot_indices.size() - 1) {
476                     cout << " -> ";
477                 }
478             }
479             cout << " -> 酒店" << endl;
480             cout << "起始时间: " << routes[d].start_time << " 时" << endl;
481             cout << "结束时间: " << routes[d].end_time << " 时" << endl;
482             cout << "游玩时间: " << routes[d].play_duration << " 小时" << endl;
483             cout << "交通时间: " << routes[d].travel_duration << " 小时" << endl;
484             cout << "当日花费: " << routes[d].cost << " 元" << endl;
485             cout << "期待总值: " << routes[d].interest << endl;
486             total_cost += routes[d].cost;
487             total_interest += routes[d].interest;
488         }
489         cout << "----------------------------------" << endl;
490     }
491     cout << "\n总预算消耗: " << total_cost << " 元" << endl;
492     cout << "总期待程度: " << total_interest << endl;
493     cout << "==================================" << endl;
494 }
495
496 /*
497  * 主函数
498  * 处理输入并调用规划函数
499  */
500 int main() {
501     int N, D;
```

```
    double B;

    cout << "请输入景点数量 N、旅行天数 D、总预算 B（元）：\n";
    cin >> N >> D >> B;

    vector<Spot> spots(N);
    cout << "请输入每个景点的信息（名称 票价 游玩时间 期待 开始时间 结束时间 x
        y）：\n";
    for (int i = 0; i < N; ++i) {
        cin >> spots[i].name >> spots[i].ticket >> spots[i].play_time
            >> spots[i].interest >> spots[i].open_time >> spots[i].close_time
            >> spots[i].x >> spots[i].y;
    }

    vector<Day> days(D);
    cout << "请输入每一天的可用时间（开始时间 结束时间）：\n";
    for (int i = 0; i < D; ++i) {
        cin >> days[i].start_time >> days[i].end_time;
    }

    // 调用路线规划函数
    vector<Route> routes = plan_routes(N, D, B, spots, days);

    // 打印规划结果
    print_routes(routes, spots);

    return 0;
}
```

接下来对代码进行几次简单的测试，得到以下结果：

- 针对多天多景点的测试，得到以下测试结果：

```
请输入景点数量 N、旅行天数 D、总预算 B（元）：
5 2 500
请输入每个景点的信息（名称 票价 游玩时间 期待 开始时间 结束时间 x y）：
A 50 1.5 80 8 18 1 1
B 40 1 70 8 18 2 2
C 30 2 60 8 18 3 3
D 20 1.5 50 8 18 4 4
E 10 1 30 8 18 5 5
请输入每一天的可用时间（开始时间 结束时间）：
9 18
9 18

========== 行程规划结果 ==========
第1天行程：
行程路线：酒店 -> A -> B -> C -> D -> E -> 酒店
起始时间：9.00 时
结束时间：16.47 时
游玩时间：7.00 小时
交通时间：0.47 小时
当日花费：182.53 元
期待总值：290.00
--------------------------------
第2天行程：
无可行景点行程
--------------------------------

总预算消耗：182.53 元
总期待程度：290.00
================================
```

图 5.1: 测试样例 1

- 针对预算不足的测试，得到结果如下：

```
请输入景点数量 N、旅行天数 D、总预算 B（元）：
3 1 40
请输入每个景点的信息（名称 票价 游玩时间 期待 开始时间 结束时间 x y）：
A 50 2 80 8 18 1 1
B 60 1.5 90 8 18 2 2
C 30 1 50 8 18 3 3
请输入每一天的可用时间（开始时间 结束时间）：
8 20

========== 行程规划结果 ==========
第1天行程：
无可行景点行程
----------------------------------

总预算消耗：0.00 元
总期待程度：0.00
==================================
```

图 5.2: 测试样例 2

- 针对参观时间不达标的测试，得到的结果如下：

```
请输入景点数量 N、旅行天数 D、总预算 B（元）：
3 1 100
请输入每个景点的信息（名称 票价 游玩时间 期待 开始时间 结束时间 x y）：
A 20 2 70 13 14 1 1
B 20 2 60 15 16 2 2
C 20 2 50 17 18 3 3
请输入每一天的可用时间（开始时间 结束时间）：
8 12

========== 行程规划结果 ==========
第1天行程：
无可行景点行程
----------------------------------

总预算消耗：0.00 元
总期待程度：0.00
==================================
```

图 5.3: 测试样例 3

- 最后，为了模拟实际情况与数据规模（一般一个旅游城市有几十个景点，而游玩的时间只有几天，不可能全部游玩），我设计了 20 个景点，3 天，1000 元预算的测试样例，并且为了与实际相符，我规定每天游玩的景点上限个数为 5（人的精力有限），最终测试得到如下的安排，实际上讲，这个程序已经能为实际情况给出一定的参考：

```
请输入景点数量 N、旅行天数 D、总预算 B（元）：
20 3 1000
请输入每个景点的信息（名称 票价 游玩时间 期待 开始时间 结束时间 x y）：
A 50 2 80 8 18 1 1
B 60 1.5 75 9 17 2 2
C 40 1 60 10 16 3 3
D 55 1.8 78 8 15 4 4
E 45 1.2 65 11 19 5 1
F 70 2 90 9 18 6 5
G 30 1 50 10 14 1 6
H 65 1.5 82 10 18 7 7
I 55 2 75 8 16 8 2
J 40 1 60 9 17 2 8
K 50 1.5 70 10 15 5 5
L 45 1.2 68 9 16 3 6
M 60 1.8 80 8 14 6 3
N 35 1 55 10 18 4 7
O 50 1.5 72 9 16 7 1
P 40 1 65 8 17 1 8
Q 55 2 77 9 15 5 3
R 65 1.5 85 10 18 6 6
S 50 1.3 70 8 14 3 4
T 45 1 60 9 17 2 5
请输入每一天的可用时间（开始时间 结束时间）：
10 20
8 18
8 16
```

图 5.4: 测试样例 4-输入

```
========== 行程规划结果 ==========
第1天行程：
行程路线：酒店 -> C -> B -> F -> R -> H -> 酒店
起始时间：10.00 时
结束时间：18.27 时
游玩时间：7.50 小时
交通时间：0.77 小时
当日花费：352.83 元
期待总值：392.00
----------------------------------
第2天行程：
行程路线：酒店 -> M -> D -> O -> P -> A -> 酒店
起始时间：8.00 时
结束时间：17.13 时
游玩时间：8.10 小时
交通时间：1.03 小时
当日花费：325.89 元
期待总值：375.00
----------------------------------
第3天行程：
行程路线：酒店 -> K -> L -> I -> 酒店
起始时间：8.00 时
结束时间：15.26 时
游玩时间：4.70 小时
交通时间：0.80 小时
当日花费：205.10 元
期待总值：333.00
----------------------------------

总预算消耗：883.82 元
总期待程度：1100.00
==================================
```

图 5.5: 测试样例 4-结果

测试到这个程度，实验设计基本完成，已经能解决“选择困难症”或“计划拖延者”的一些难题。

6 实验总结与心得

通过本次实验，我成功设计并实现了一个旅游行程规划系统。该系统能够综合考虑景点的空间位置、门票费用、游玩时间、期待程度、开放时间以及每天的可用时间等因素，通过动态规划、贪心算法和图算法等技术手段，最大化总期待程度，同时满足时间和预算约束。实验中，我对多种测试样例进行了验证，包括多天多景点、预算不足、参观时间不达标以及大规模数据的模拟场景。结果表明，该系统能够在不同情况下生成合理且高效的行程规划，为实际旅游规划提供了一定的参考价值。未来，我可以进一步优化该系统，例如接入实时地图数据以获取更准确的交通信息，或者开发一个前端网站，将该系统实例化为一个用户友好的在线旅游规划工具，从而更好地服务于实际应用。

在本学期的算法学习中，我深刻体会到了算法的多样性和强大功能。通过本次实验，我将《算法导论》中学到的图算法、动态规划、贪心算法等知识应用到实际问题中，进一步加深了对这些算法的理解。例如，动态规划在解决多阶段决策问题时的高效性和贪心算法在局部优化中的简洁性，都让我感受到算法设计的魅力。同时，我也意识到算法的选择和组合对于解决复杂问题的重要性。在实验过程中，我不断尝试不同的算法组合，最终找到了一种既能满足约束条件又能优化目标的解决方案。这不仅提升了我的编程能力，也让我学会了如何根据问题的特点灵活运用所学的算法知识。此外，通过本次实验，我更加明白了理论与实践相结合的重要性，只有将所学知识应用到实际问题中，才能真正发挥其价值。