第 9 章　分支限界法

1. 给定 n 种物品和一个背包。物品 i 的重量是 w_i，其价值为 v_i，$i=1$，…，n，背包容量为 c。问应该如何选择装入背包的物品，使得装入背包中物品的总价值最大？使用队列式分支限界法，采用先进先出搜索求解 0-1 背包问题。

C/C++代码：

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_N100

//定义一个结构体表示节点
typedef struct {
    int level; // 当前物品的层级
    int profit; // 当前总价值
    int weight; // 当前总重量
    int bound; // 当前节点的上界
    int items[MAX_N]; // 记录每个物品是否选择
} Node;

//物品数量
int n;
//物品重量和价值
int weights[MAX_N], values[MAX_N];
//背包容量
int capacity;

//上界计算函数
int bound(Node node) {
    if (node.weight > capacity) {
        return 0;   // 超出容量，上界为 0
    }

    int bound_value = node.profit;
    int total_weight = node.weight;
    int j = node.level;
```

```
        // 贪心方式将物品装入背包
        while (j < n && total_weight + weights[j] <= capacity) {
            total_weight += weights[j];
            bound_value += values[j];
            j++;
        }

        // 如果仍有剩余容量, 则按单位价值比例装入部分物品
        if (j < n) {
            bound_value += (capacity - total_weight) * (values[j] / (float)
weights[j]);
        }

        return bound_value;
    }

    //队列式分支限界法
    void knapsack_branch_and_bound() {
        // 初始化队列
        Node queue[MAX_N * 2];
        int front =0, rear = 0;

        // 创建根节点: 没有选择任何物品
        Node root = {0, 0, 0, 0, {0}};
        root.bound = bound(root);
        queue[rear++] = root;

        int max_profit =0;
        int best_items[MAX_N] = {0};

        // 广度优先搜索 (队列方式)
        while (front < rear) {
            Node u = queue[front++]; // 出队

            if (u.level == n) {  // 到达叶子节点
                if (u.profit > max_profit) {
                    max_profit = u.profit;
                    for (int i = 0; i < n; i++) {
                        best_items[i] = u.items[i];
```

```
            }
        }
        continue;
    }

    // 不选择当前物品
    Node v1 = u;
    v1.level++;
    v1.bound = bound(v1);
    if (v1.bound > max_profit) {
        queue[rear++] = v1; // 入队
    }

    // 选择当前物品(如果容量允许)
    if (u.weight + weights[u.level] <= capacity) {
        Node v2 = u;
        v2.level++;
        v2.profit += values[u.level];
        v2.weight += weights[u.level];
        v2.items[u.level] =1;   // 选择当前物品
        if (v2.profit > max_profit) {
            max_profit = v2.profit;
            for (int i = 0; i < n; i++) {
                best_items[i] = v2.items[i];
            }
        }
        v2.bound = bound(v2);
        if (v2.bound > max_profit) {
            queue[rear++] = v2; // 入队
        }
    }
}

// 输出结果
printf("% d\n", max_profit);
for (int i = 0; i < n; i++) {
    printf("% d ", best_items[i]);
}
printf("\n");
```

```
}

int main() {
    // 输入数据
    scanf("%d", &n);   // 物品数量
    int weights_input[MAX_N], values_input[MAX_N];
    for (int i = 0; i < n; i++) {
        scanf("%d", &weights_input[i]);   // 物品重量
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", &values_input[i]);   // 物品价值
    }
    scanf("%d", &capacity);   // 背包容量
    // 将输入的数据赋值给全局数组
    for (int i = 0; i < n; i++) {
        weights[i] = weights_input[i];
        values[i] = values_input[i];
    }
    // 调用分支限界法求解
    knapsack_branch_and_bound();
    return 0;
}
```

Python 代码：

```
from collections import deque

def knapsack_branch_and_bound(n, weights, values, capacity):
    # 上界计算函数
    def bound(level, profit, weight):
        if weight > capacity:    # 超出容量，上界为 0
            return 0
        bound_value = profit    # 当前价值
        totweight = weight      # 当前重量
        j = level
        # 贪心加入完整物品
        while j < n and totweight + weights[j] <= capacity:
            totweight += weights[j]
            bound_value += values[j]
            j +=1
        # 如果还有剩余容量，按单位价值加入部分物品
```

```
        if j < n:
            bound_value += (capacity - totweight) * (values[j] / weights[j])
        return bound_value

    # 节点类
    class Node:
        def __init__(self, level, profit, weight, items):
            self.level = level
            self.profit = profit
            self.weight = weight
            self.items = items.copy()  # 复制选择方案
            self.bound = bound(level, profit, weight)  # 计算上界

    # 初始化队列和最优解
    queue = deque()
    root = Node(0, 0, 0, [0] * n)  # 根节点: 未选择任何物品
    queue.append(root)
    max_profit =0
    best_items = [0] * n

    # FIFO 搜索
    while queue:
        u = queue.popleft()# 出队

        if u.level == n:   # 叶子节点
            if u.profit > max_profit:
                max_profit = u.profit
                best_items = u.items
            continue

        # 子节点 1: 不选择当前物品
        v1 = Node(u.level +1, u.profit, u.weight, u.items)
        if v1.bound > max_profit:   # 上界大于当前最优解才入队
            queue.append(v1)

        # 子节点 2: 选择当前物品(如果容量允许)
        if u.weight + weights[u.level] <= capacity:
            v2 = Node(u.level +1, u.profit + values[u.level], u.weight +
weights[u.level], u.items)
```

```
                v2.items[u.level] =1  # 标记选择
                if v2.profit > max_profit:   # 更新最优解
                    max_profit = v2.profit
                    best_items = v2.items
                if v2.bound > max_profit:   # 上界大于当前最优解才入队
                    queue.append(v2)
        return max_profit, best_items

    if __name__ == "__main__":
        # 读取输入
        n =int(input())
        weights =list(map(int, input().split()))
        values =list(map(int, input().split()))
        capacity =int(input())
        # 求解
        max_profit, best_items = knapsack_branch_and_bound(n, weights,
values, capacity)
        # 输出结果
        print(max_profit)
        print(' '.join(map(str, best_items)))
```

2. 给定 n 种物品和一个背包。物品 i 的重量是 w_i，其价值为 v_i，$i=1, \cdots, n$，背包容量为 c。问应该如何选择装入背包的物品，使得装入背包中物品的总价值最大？使用优先队列式分支限界法，求解 0-1 背包问题。

C/C++代码：

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_N100
#define MAX_CAPACITY1000

//定义节点结构体
typedef struct {
    int level; // 当前物品的层级
    int profit; // 当前总价值
    int weight; // 当前总重量
    int bound; // 当前节点的上界
    int items[MAX_N]; // 记录每个物品是否选择
} Node;
```

```
//物品数量和背包容量
int n, c;
//物品重量和价值
int w[MAX_N], v[MAX_N];

//最大堆相关函数
Node heap[MAX_N * 2];
int heapSize =0;

void swap(Node * a, Node * b) {
    Node temp = * a;
    * a = * b;
    * b = temp;
}

void heapifyUp(int i) {
    while (i > 0 && heap[i].bound > heap[(i - 1) / 2].bound) {
        swap(&heap[i], &heap[(i -1) / 2]);
        i = (i -1) / 2;
    }
}

void heapifyDown(int i) {
    int largest = i;
    int left =2 *  i + 1;
    int right =2 *  i + 2;
    if (left < heapSize && heap[left].bound > heap[largest].bound) {
        largest = left;
    }
    if (right < heapSize && heap[right].bound > heap[largest].bound) {
        largest = right;
    }
    if (largest ! = i) {
        swap(&heap[i], &heap[largest]);
        heapifyDown(largest);
    }
}

void pushHeap(Node node) {
```

```
    heap[heapSize] = node;
    heapifyUp(heapSize);
    heapSize++;
}

Node popHeap() {
    Node root = heap[0];
    heapSize--;
    heap[0] = heap[heapSize];
    heapifyDown(0);
    return root;
}

//计算上界函数
int bound(Node node) {
    if (node.weight > c) {
        return 0;
    }
    int bound_value = node.profit;
    int total_weight = node.weight;
    int j = node.level;
    // 贪心方式将物品装入背包
    while (j < n && total_weight + w[j] <= c) {
        total_weight += w[j];
        bound_value += v[j];
        j++;
    }
    // 如果仍有剩余容量，则按单位价值比例装入部分物品
    if (j < n) {
        bound_value += (c - total_weight) * (v[j] / (float)w[j]);
    }
    return bound_value;
}

//分支限界法求解 0-1 背包问题
void knapsack_branch_and_bound() {
    int maxProfit =0;
    int bestItems[MAX_N] = {0};
```

```
// 初始化根节点
Node root = {0, 0, 0, bound(root), {0}};
pushHeap(root);
while (heapSize > 0) {
    Node u = popHeap(); // 取出上界最大的节点
    if (u.level == n) {  // 叶子节点
        if (u.profit > maxProfit) {
            maxProfit = u.profit;
            for (int i = 0; i < n; i++) {
                bestItems[i] = u.items[i];
            }
        }
        continue;
    }
    // 不选择当前物品
    Node v1 = u;
    v1.level++;
    v1.bound = bound(v1);
    if (v1.bound > maxProfit) {
        pushHeap(v1);
    }
    // 选择当前物品(如果容量允许)
    if (u.weight + w[u.level] <= c) {
        Node v2 = u;
        v2.level++;
        v2.profit += v[u.level];
        v2.weight += w[u.level];
        v2.items[u.level] =1;   // 选择当前物品
        if (v2.profit > maxProfit) {
            maxProfit = v2.profit;
            for (int i = 0; i < n; i++) {
                bestItems[i] = v2.items[i];
            }
        }
        v2.bound = bound(v2);
        if (v2.bound > maxProfit) {
            pushHeap(v2);
        }
    }
```

```
        }

        // 输出结果
        printf("% d\n", maxProfit);
        for (int i = 0; i < n; i++) {
            printf("% d ", bestItems[i]);
        }
        printf("\n");
    }

    int main() {
        // 输入数据
        scanf("% d", &n);   // 物品数量
        for (int i = 0; i < n; i++) {
            scanf("% d", &w[i]);   // 物品重量
        }
        for (int i = 0; i < n; i++) {
            scanf("% d", &v[i]);   // 物品价值
        }
        scanf("% d", &c);   // 背包容量
        // 求解
        knapsack_branch_and_bound();
        return 0;
    }
```

Python 代码：

```
    import heapq

    def knapsack(n, w, v, c):
        # 计算上界函数
        def bound(level, profit, weight):
            if weight > c:
                return 0
            bound_value = profit
            totweight = weight
            # 按单位价值排序剩余物品
            remaining =sorted([(v[i]/w[i], w[i], v[i]) for i in range(level,
n)], reverse=True)
            for unit_val, wt, val in remaining:
                if totweight + wt <= c:
```

```
            totweight += wt
            bound_value += val
        else:
            bound_value += (c - totweight) *  unit_val
            break
    return bound_value

# 节点类
class Node:
    def __init__(self, level, profit, weight, items):
        self.level = level
        self.profit = profit
        self.weight = weight
        self.items = items.copy()
        self.bound = bound(level, profit, weight)

    def __lt__(self, other):
        return self.bound > other.bound  # 最大堆

# 初始化
maxProfit =0
bestItems = [0] *  n
pq = []# 优先队列(最大堆)
root = Node(0, 0, 0, [0] *  n)
heapq.heappush(pq, root)

while pq:
    u = heapq.heappop(pq)# 取出上界最大的节点

    if u.level == n:   # 叶子节点
        if u.profit > maxProfit:
            maxProfit = u.profit
            bestItems = u.items
        continue

    # 子节点: 不选择当前物品
    v1 = Node(u.level +1, u.profit, u.weight, u.items)
    if v1.bound > maxProfit:
        heapq.heappush(pq, v1)
```

```
            # 子节点: 选择当前物品(如果重量允许)
            if u.weight + w[u.level] <= c:
                v2 = Node(u.level +1, u.profit + v[u.level], u.weight + w[u.
level], u.items)
                v2.items[u.level] =1
                if v2.profit > maxProfit:    # 更新最优解
                    maxProfit = v2.profit
                    bestItems = v2.items
                if v2.bound > maxProfit:
                    heapq.heappush(pq, v2)

        return maxProfit, bestItems

    if __name__ == "__main__":
        # 读取输入
        n =int(input())
        w =list(map(int, input().split()))
        v =list(map(int, input().split()))
        c =int(input())

        # 求解
        maxProfit, bestItems = knapsack(n, w, v, c)

        # 输出结果
        print(maxProfit)
        print('  '.join(map(str, bestItems)))
```

3. 某售货员要到若干城市去推销商品，已知各城市之间的距离，他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程最短。下图中，圆圈中的数字表示城市的编号，共有 4 个城市；连线上的数字表示城市之间的距离，例如城市 1 和城市 2 之间的距离是 30。

C/C++代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAX_N20

//定义一个节点结构，包含下界、当前成本和路径
typedef struct {
```

```
    int bound; // 下界
    int cost; // 当前成本
    int path[MAX_N]; // 当前路径
    int path_len; // 当前路径的长度
} Node;

//优先队列(最小堆)相关变量
Node heap[MAX_N * 100];   // 堆的最大容量
int heapSize =0;

void swap(Node * a, Node * b) {
    Node temp = * a;
    * a = * b;
    * b = temp;
}

void heapifyUp(int i) {
    while (i > 0 && heap[i].bound < heap[(i - 1) / 2].bound) {
        swap(&heap[i], &heap[(i -1) / 2]);
        i = (i -1) / 2;
    }
}

void heapifyDown(int i) {
    int smallest = i;
    int left =2 *  i + 1;
    int right =2 *  i + 2;

    if (left < heapSize && heap[left].bound < heap[smallest].bound) {
        smallest = left;
    }

    if (right < heapSize && heap[right].bound < heap[smallest].bound) {
        smallest = right;
    }

    if (smallest != i) {
        swap(&heap[i], &heap[smallest]);
        heapifyDown(smallest);
```

```
    }
}

void pushHeap(Node node) {
    heap[heapSize] = node;
    heapifyUp(heapSize);
    heapSize++;
}

Node popHeap() {
    Node root = heap[0];
    heapSize--;
    heap[0] = heap[heapSize];
    heapifyDown(0);
    return root;
}

//计算当前路径的下界
int calculate_bound(Node node, int n, int dist[MAX_N][MAX_N]) {
    int bound = node.cost;
    int visited[MAX_N] = {0};
    for (int i = 0; i < node.path_len; i++) {
        visited[node.path[i]] =1;
    }

    // 贪心计算剩余城市的最小出边
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            int min_edge = INT_MAX;
            for (int j = 0; j < n; j++) {
                if (dist[i][j] != -1) {
                    min_edge = (min_edge < dist[i][j]) ? min_edge : dist[i][j];
                }
            }
            bound += min_edge;
        }
    }

    // 最后一个城市到起点的最小边
```

```
        int last_city = node.path[node.path_len -1];
        if (node.path_len < n) {
            int min_edge = INT_MAX;
            for (int j = 0; j < n; j++) {
                if (!visited[j] && dist[last_city][j] != -1) {
                    min_edge = (min_edge < dist[last_city][j]) ? min_edge :
dist[last_city][j];
                }
            }
            bound += min_edge;
        }

        return bound;
    }

    // TSP 分支限界法求解最短路径
    int tsp_branch_and_bound(int n, int dist[MAX_N][MAX_N]) {
        int min_cost = INT_MAX;

        // 初始化根节点
        Node root = {0, 0, {0}, 1};   // 起点是 0，路径只包含起点
        root.bound = calculate_bound(root, n, dist);
        pushHeap(root);

        while (heapSize > 0) {
            Node u = popHeap();

            // 如果当前下界大于最优解，剪枝
            if (u.bound >= min_cost) {
                continue;
            }

            // 如果路径包含所有城市，计算回到起点的成本
            if (u.path_len == n) {
                int total_cost = u.cost + dist[u.path[n -1]][u.path[0]];
                if (total_cost < min_cost) {
                    min_cost = total_cost;
                }
                continue;
```

```
        }

        // 扩展未访问的城市
        int curr_city = u.path[u.path_len -1];
        for (int next_city = 0; next_city < n; next_city++) {
            // 检查是否已经访问过这个城市
            int already_visited =0;
            for (int i = 0; i < u.path_len; i++) {
                if (u.path[i] == next_city) {
                    already_visited =1;
                    break;
                }
            }

            if (!already_visited) {
                Node v = u;
                v.cost += dist[curr_city][next_city];
                if (v.cost >= min_cost) {
                    continue;   // 剪枝
                }

                v.path[v.path_len] = next_city;
                v.path_len++;

                v.bound = calculate_bound(v, n, dist);
                if (v.bound < min_cost) {
                    pushHeap(v);
                }
            }
        }
    }

    return min_cost;
}

int main() {
    int n;
    scanf("% d", &n);   // 输入城市数量
    int dist[MAX_N][MAX_N];
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &dist[i][j]);
        }
    }
    // 求解并输出最短路径的总成本
    int result = tsp_branch_and_bound(n, dist);
    printf("%d\n", result);
    return 0;
}
```

Python 代码：

```
import heapq

def tsp_branch_and_bound(n, dist):
    # 计算下界函数
    def calculate_bound(node):
        cost = node[1]  # 当前路径成本
        visited =set(node[2])  # 已访问城市
        bound = cost
        # 对每个未访问城市，加入其最小出边
        for i in range(n):
            if i not in visited:
                min_edge =float('inf')
                for j in range(n):
                    if dist[i][j] != -1:  # 排除主对角线
                        min_edge =min(min_edge, dist[i][j])
                bound += min_edge
        # 当前城市到起点的距离
        last_city = node[2][-1]
        if len(visited) < n:
            min_edge =float('inf')
            for j in range(n):
                if j not in visited and dist[last_city][j] != -1:
                    min_edge =min(min_edge, dist[last_city][j])
            bound += min_edge
        return bound

    # 初始化
    min_cost =float('inf')
```

```
    pq = []# 优先队列: (bound, cost, path)
    root = (0, 0, [0])  # 下界, 当前成本, 路径
    heapq.heappush(pq, (calculate_bound(root), root[1], root[2]))

    while pq:
        bound, cost, path = heapq.heappop(pq)

        if bound >= min_cost:   # 剪枝
            continue

        if len(path) == n:   # 到达所有城市
            total_cost = cost + dist[path[-1]][0]  # 回到起点
            if total_cost < min_cost:
                min_cost = total_cost
            continue

        # 扩展未访问城市
        curr_city = path[-1]
        for next_city in range(n):
            if next_city not in path:
                new_cost = cost + dist[curr_city][next_city]
                if new_cost >= min_cost:   # 剪枝
                    continue
                new_path = path + [next_city]
                new_node = (0, new_cost, new_path)
                new_bound = calculate_bound(new_node)
                if new_bound < min_cost:
                    heapq.heappush(pq, (new_bound, new_cost, new_path))

    return min_cost

if __name__ == "__main__":
    # 读取输入
    n =int(input())
    dist = []
    for _ in range(n):
        row =list(map(int, input().split()))
        dist.append(row)
    # 求解
```

```
    result = tsp_branch_and_bound(n, dist)
    # 输出结果
    print(result)
```

4. 对于给出的电路图和指定大小的电路板，需要将电路在电路板上实现。所谓电路板，可以看作是一个 $n \times n$ 的格子图（如下图所示）。用户给定的电路由若干电路原件组成，每一个电路原件可能会占用一个或多个格子。这里，我们将被电路原件占据的格子分为两类。

第一类只是纯粹占据了这个格子，之后这个格子不会再被使用，也不会从被占据的位置连出去任何的电路线。这样的格子被我们视作是电路板上的障碍物。

还有一类格子，我们称为是电路原件的接口，上面虽然被电路原件占用，但是仍有可能从其中连出去一些电路线到其它的电路原件上，从而形成电路。

布线时，电路只能沿着直线或直角布线。现在，希望找到一个可行方案，使得路径的总长度最短。

C/C++代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

//定义节点结构体
typedef struct {
    int total_est_cost; // 估计总成本 (f = cost + heuristic)
    int cost; // 当前路径长度
    int x, y; // 当前位置
} Node;

//定义优先队列结构体
typedef struct {
    Node*  nodes;
    int size;
    int capacity;
} PriorityQueue;

//初始化优先队列
PriorityQueue*  create_pq(int capacity) {
    PriorityQueue*  pq = (PriorityQueue* )malloc(sizeof(PriorityQueue));
    pq->nodes = (Node* )malloc(capacity *  sizeof(Node));
    pq->size =0;
    pq->capacity = capacity;
    return pq;
```

```
    }

    //释放优先队列内存
    void free_pq(PriorityQueue* pq) {
        free(pq->nodes);
        free(pq);
    }
    //交换两个节点
    void swap(Node* a, Node* b) {
        Node temp = * a;
        * a = * b;
        * b = temp;
    }

    //堆化(向下调整)
    void heapify_down(PriorityQueue* pq, int index) {
        int smallest = index;
        int left =2 * index + 1;
        int right =2 * index + 2;

        if (left < pq->size && pq->nodes[left].total_est_cost < pq->nodes
[smallest].total_est_cost)
            smallest = left;
        if (right < pq->size && pq->nodes[right].total_est_cost < pq->nodes
[smallest].total_est_cost)
            smallest = right;

        if (smallest != index) {
            swap(&pq->nodes[index], &pq->nodes[smallest]);
            heapify_down(pq, smallest);
        }
    }

    //堆化(向上调整)
    void heapify_up(PriorityQueue* pq, int index) {
        while (index > 0) {
            int parent = (index -1) / 2;
            if (pq->nodes[parent].total_est_cost > pq->nodes[index].total_
est_cost) {
```

```
            swap(&pq->nodes[parent], &pq->nodes[index]);
            index = parent;
        }else {
            break;
        }
    }
}

//入队
void pq_push(PriorityQueue* pq, int total_est_cost, int cost, int x, int
y) {
    if (pq->size == pq->capacity) {
        pq->capacity * =2;
        pq->nodes = (Node* )realloc(pq->nodes, pq->capacity * sizeof
(Node));
    }
    Node node = {total_est_cost, cost, x, y};
    pq->nodes[pq->size] = node;
    heapify_up(pq, pq->size);
    pq->size++;
}

//出队
Node pq_pop(PriorityQueue* pq) {
    Node root = pq->nodes[0];
    pq->nodes[0] = pq->nodes[pq->size - 1];
    pq->size--;
    if (pq->size > 0)
        heapify_down(pq, 0);
    return root;
}

//检查队列是否为空
bool pq_empty(PriorityQueue* pq) {
    return pq->size == 0;
}

//启发式函数: 曼哈顿距离
int heuristic(int x, int y, int end_x, int end_y) {
```

```
    return abs(x - end_x) + abs(y - end_y);
}

//检查位置是否合法(使用指针形式传递二维数组)
bool is_valid(int x, int y, int n, int** grid) {
    return x >= 0 && x < n && y >= 0 && y < n && grid[x][y] == 0;
}

//电路布线函数
int circuit_wiring(int n, int** grid, int start_x, int start_y, int end
_x, int end_y) {
    // 定义四个移动方向: 上、下、左、右
    int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    // 初始化优先队列
    PriorityQueue* pq = create_pq(1000); // 初始容量设为 1000

    // 初始化 visited 数组
    bool** visited = (bool** )malloc(n * sizeof(bool* ));
    for (int i = 0; i < n; i++) {
        visited[i] = (bool* )calloc(n, sizeof(bool));
    }

    // 起点入队
    int start_heuristic = heuristic(start_x, start_y, end_x, end_y);
    pq_push(pq, start_heuristic, 0, start_x, start_y);
    visited[start_x][start_y] =true;

    while (!pq_empty(pq)) {
        Node node = pq_pop(pq);
        int total_est_cost = node.total_est_cost;
        int cost = node.cost;
        int x = node.x;
        int y = node.y;

        // 到达终点, 返回路径长度(格子数 = 边数 + 1)
        if (x == end_x && y == end_y) {
            free_pq(pq);
            for (int i = 0; i < n; i++)
```

```
                free(visited[i]);
            free(visited);
            return cost + 1;
        }

        // 扩展相邻的四个方向
        for (int i = 0; i < 4; i++) {
            int nx = x + directions[i][0];
            int ny = y + directions[i][1];
            if (is_valid(nx, ny, n, grid) && !visited[nx][ny]) {
                visited[nx][ny] =true;
                int new_cost = cost +1;
                int new_total_est_cost = new_cost + heuristic(nx, ny, end
_x, end_y);
                pq_push(pq, new_total_est_cost, new_cost, nx, ny);
            }
        }
    }
    free_pq(pq);
    for (int i = 0; i < n; i++)
        free(visited[i]);
    free(visited);
    return -1; // 无解
}

int main() {
    int n;
    scanf("% d", &n);
    // 动态分配二维数组 grid
    int* *  grid = (int* * )malloc(n *  sizeof(int* ));
    for (int i = 0; i < n; i++) {
        grid[i] = (int* )malloc(n *  sizeof(int));
        for (int j = 0; j < n; j++) {
            scanf("% d", &grid[i][j]);
        }
    }
    // 读取起点和终点坐标
    int x1, y1, x2, y2;
    scanf("% d % d % d % d", &x1, &y1, &x2, &y2);
```

```
    // 转换为 0-based 索引
    int start_x = x1 -1;
    int start_y = y1 -1;
    int end_x = x2 -1;
    int end_y = y2 -1;
    // 求解
    int result = circuit_wiring(n, grid, start_x, start_y, end_x, end_y);
    // 输出结果
    printf("% d\n", result);
    // 释放 grid 内存
    for (int i = 0; i < n; i++)
        free(grid[i]);
    free(grid);
    return 0;
}
```

Python 代码：

```
import heapq

def circuit_wiring(n, grid, start, end):
    # 定义四个移动方向：上、下、左、右
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # 启发式函数：曼哈顿距离
    def heuristic(x, y):
        return abs(x - end[0]) + abs(y - end[1])

    # 检查位置是否合法
    def is_valid(x, y):
        return 0 <= x < n and 0 <= y < n and grid[x][y] == 0

    # 初始化优先队列：(估计总成本，当前成本，当前位置)
    pq = []
    heapq.heappush(pq, (heuristic(start[0], start[1]), 0, start))
    visited =set()
    visited.add(start)

    while pq:
        # 取出估计总成本最小的节点
        total_est_cost, cost, (x, y) = heapq.heappop(pq)
```

```
            # 到达终点，返回路径长度(格子数 = 边数 + 1)
            if (x, y) == end:
                return cost + 1  # 格子数

            # 扩展相邻的四个方向
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if is_valid(nx, ny) and (nx, ny) not in visited:
                    visited.add((nx, ny))
                    new_cost = cost +1
                    total_est_cost = new_cost + heuristic(nx, ny)
                    heapq.heappush(pq, (total_est_cost, new_cost, (nx, ny)))

    return -1  # 无解
if __name__ == "__main__":
    # 读取输入
    n =int(input())
    grid = []
    for _ in range(n):
        row =list(map(int, input().split()))
        grid.append(row)
    # 读取起点和终点坐标，并转换为 0-based 索引
    x1, y1, x2, y2 =map(int, input().split())
    start = (x1 -1, y1 - 1)  # 转换为 0-based
    end = (x2 -1, y2 - 1)
    # 求解
    result = circuit_wiring(n, grid, start, end)
    # 输出结果
    print(result)
```