# 基于动态规划和博弈论的沙漠穿越决策优化模型

## 摘要

本文针对“穿越沙漠”这一典型的资源约束和多变环境下的动态决策问题，建立了一系列由简入繁、层层递进的数学模型。该问题的核心在于为不同信息和交互条件下的决策主体，寻求长期收益最大化的最优策略。

**针对问题一**，我们首先把第一关和第二关的地图进行简化，构建起点、村庄、矿山、终点之间的最短路径图。在天气状况完全已知的确定性环境下，我们将单人最优路径问题建模为一个**多阶段动态规划模型**。该模型以玩家每日的时间, 地点, 水, 食物作为状态变量，通过构建**状态转移方程**来精确描述玩家在已知天气下的行动（如停留、移动、挖矿）对状态的改变。我们以最终剩余资金最大化为优化目标，通过**逆向求解**的方式，为整个时空状态空间计算出最优价值，从而能够精确求解出最优的行动方案。最终得到的第一关剩余最大资金为 **10230 元**，第二关剩余最大资金为 **12730 元**。

**针对问题二**，在未来天气未知的随机环境下，我们将问题抽象为一个**马尔可夫决策过程**。其解不再是固定的行动路径，而是状态依赖的最优策略。考虑到包含资源维度的状态空间巨大，直接求解动作价值函数会导致计算不可行，我们采用**价值迭代算法**，以及**有限期逆向推导**等技巧，对状态价值函数进行求解。该方法以**贝尔曼最优方程**为理论核心，最终在决策时，通过前瞻一步的方式从最优价值函数中直接导出每日的动态最优策略，既保证了决策的动态最优性，又有效规避了存储巨大 Q 表的难题。

**针对问题三**，在引入多玩家的博弈环境下，我们分析了其强烈的冲突惩罚机制，将问题定性为**非合作博弈**，其核心在于冲突规避。对于第一小问的静态博弈，我们指出其解为**纳什均衡**。我们探讨了两种均衡形式：1) 通过**迭代最佳响应算法**，模拟理性玩家之间互相揣测的过程，求解各玩家的**纯策略纳什均衡**；2) 为解决对称博弈中的“协调困境”，我们进一步引入**混合策略纳什均衡**，将玩家的行动策略定义为行动的概率分布，使其更具普适性。对于第二小问的动态博弈，我们构建了一个多智能体决策模型，其中每个玩家通过预测其他玩家的行为来动态调整自身策略，以规避冲突、最大化个人收益。

最终，本文通过构建从精确动态规划到近似动态规划，再到非合作博弈均衡分析的完整模型体系，为“穿越沙漠”问题在不同层次的复杂情境下，提供了兼具理论深度与实践可行性的最优策略方案。

**关键字**：动态规划　马尔可夫决策　价值迭代　纳什均衡

# 一、 问题背景与重述

## 1.1 问题背景

该题目的背景基于一个“穿越沙漠”的小游戏，核心设定如下：

玩家凭借一张地图，从起点出发，通过合理规划路线、安排行动（停留、行走或挖矿）、资源购买，在限定天数之前到达终点，并使剩余资金最大化。游戏的核心规则包括：

- 资源管理

(1) 玩家需携带水和食物两种资源（以“箱”为单位），在到达终点之前，水和食物都不能消耗完且携带总量不能超过负重上限。

(2) 由于天气（晴朗、高温或沙暴）和行动（停留、行走或挖矿）不同，每天的资源消耗量也可能不同：在晴朗、高温和沙暴天气下停留一天所需的物资量为各自的基础消耗量，行走和挖矿的资源消耗分别为各自基础消耗量的 2 倍和 3 倍。

- 特殊区域互动

(1) 起点：第 0 天可按基准价购买资源，仅 1 次机会；

(2) 矿山：停留可挖矿（到达当天不可挖），挖矿获基础收益，沙暴日可挖矿；

(3) 村庄：可按基准价 2 倍购买资源；

(4) 终点：剩余资源按基准价一半退款。

- 天气与行动约束

沙暴日必须停留，非沙暴日可移动至相邻区域或停留。

## 1.2 问题重述

本题中，每个问题具有不同的游戏设定，下面一一阐述题意。

**问题一：**单玩家事先知道游戏时间（30 天）内每天的天气情况，负重上限为 1200 千克，初始资金 10000 元，挖矿一天得到的基础收益为 1000 元，水和食物的单位质量分别为 3 千克、2 千克，基准价格分别为 5 元、10 元，物资的基础消耗量在晴朗天气下分别为 5 箱、7 箱；在高温天气下分别为 5 箱、7 箱；在沙暴天气下分别为 5 箱、7 箱。基于以上游戏设定，求解第一关和第二关。

**问题二：**单玩家仅知道当天的天气情况，需在规定截止日期内到达终点。玩家负重上限为 1200 千克，初始资金 10000 元，挖矿一天得到的基础收益为 200 元，水和食物的单位质量分别为 3 千克、2 千克，基准价格分别为 5 元、10 元，物资的基础消耗量在

晴朗天气下分别为 3 箱、4 箱；在高温天气下分别为 9 箱、9 箱；在沙暴天气下分别为 10 箱、10 箱。基于以上游戏设定，讨论第三关和第四关。

**问题三：**多玩家有相同的初始资金，且同时从起点出发，需在规定截止日期内到达终点。n 名玩家互动规则如下：

- 若某天任意 $k(2 \leq k \leq n)$ 名玩家均从区域 A 行走到区域 B(A≠B)，则每人消耗的资源数量均为基础消耗量的 $2k$ 倍；
- 若某天任意 $k(2 \leq k \leq n)$ 名玩家在同一矿山挖矿，则每人消耗的资源数量均为基础消耗量的 3 倍，且每人一天可通过挖矿获得的资金是基础收益的 $\frac{1}{k}$；
- 若某天任意 $k(2 \leq k \leq n)$ 名玩家在同一村庄购买资源，每箱价格均为基准价格的 4 倍；
- 其他情况下消耗资源数量与资源价格与单人游戏相同。

基于上述规则，第五关和第六关的游戏设定有些许不同：

(1) 第五关：$n = 2$，在整个游戏时段内每天天气状况事先全部已知，限定到达终点的时间为 10 天，每名玩家的行动方案需在第 0 天确定且此后不能更改。负重上限为 1200 千克，初始资金 10000 元，挖矿一天得到的基础收益为 200 元，水和食物的单位质量分别为 3 千克、2 千克，基准价格分别为 5 元、10 元，物资的基础消耗量在晴朗天气下分别为 3 箱、4 箱；在高温天气下分别为 9 箱、9 箱；在沙暴天气下分别为 10 箱、10 箱。

(2) 第六关：$n = 3$，所有玩家仅知道当天的天气状况，从第 1 天起，每名玩家在当天行动结束后均知道其余玩家当天的行动方案和剩余的资源数量，随后确定各自第二天的行动方案。负重上限为 1200 千克，初始资金 10000 元，挖矿一天得到的基础收益为 1000 元，水和食物的单位质量分别为 3 千克、2 千克，基准价格分别为 5 元、10 元，物资的基础消耗量在晴朗天气下分别为 3 箱、4 箱；在高温天气下分别为 9 箱、9 箱；在沙暴天气下分别为 10 箱、10 箱。

# 二、 问题分析

## 2.1 问题一分析

### 2.1.1 第一关分析

直观地分析第一关的地图, 在第一关的地图中，除起点和终点外，还分布有一个矿山和一个村庄。在不考虑天气情况的条件下，起点到终点所需最短时间为 3 天，但是由于行走的 3 天都在消耗物资，玩家并未获得收益；而无论在哪种天气条件下，在矿山挖矿一天都会赚取不少的收益，因此最佳选择显然是尽量多地到矿山挖矿赚取收益。在游戏中，玩家应尽量减少行走路径过长带来的物资损失，因此我们可以构建起点、终点、

村庄和矿山之间的最短路径图，玩家进行游戏时，只按照最短路径地图中行走。由于每天的天气情况已知，且每次做出行动时相应的状态变化可求，那么我们可以写出状态转移方程，构建玩家行动的动态规划模型，利用动态规划方法求解，这样就获得了玩家在游戏过程中的状态和结束游戏时的资金剩余量，模型建立成功。

#### 2.1.2 第二关分析

相比于第一关，第二关的地图中增加了一个矿场和一个村庄，其余游戏设定与第一关类似，因此思路与第一关也类似，即构建玩家行动的动态规划模型，求解玩家在游戏中的状态和结束游戏时的资金剩余量。

### 2.2 问题二分析

#### 2.2.1 第三关分析

在第三关中，玩家仅知道当天的天气状况，但已知 10 天内不会出现沙暴天气，也就是说，玩家只能根据当前的状态做出决策。所以问题不再是一个静态的最优路径，而是变成了一个动态的最优决策。玩家根据最优策略，做出自己的行动。显然，玩家的下一步的状态只由当前状态决定，与历史状态无关，因此我们选择利用马尔科夫决策对问题进行求解。首先我们可以定义一个奖励函数，即从当前状态执行动作特定转移到一个新状态后，获得的即时奖励。然后定义一个包含状态空间、动作空间、转移概率和奖励函数的四元组，构建出 MDP 模型。接着，使用价值迭代方法[1] 并引入有限期逆行推导等技巧，计算状态值函数，即从某一状态开始，使用一种策略得到的期望回报值。得到状态值函数，我们就得到了最优策略分布，从而可以确定一般情况下玩家应作出的策略，模型建立成功。

#### 2.2.2 第四关分析

第四关地图有所改变，增加了一座矿山和一个村庄，挖矿的基础收益为 1000 元，同时游戏期限改为了 30 天，且 30 天内较少出现沙暴天气。游戏设定与第三关类似，因此依然可以像第三关一样，构建 MDP 模型，并计算出状态值函数表，得到最优策略分布，最后确定了最优策略。

### 2.3 问题三分析

#### 2.3.1 第五关分析

第五关与第三关类似，是一个挖矿收益少、竞争惩罚高、时间紧张的地图。第五关还有一个重要特征是玩家在游戏开始前提交方案，意味着这是一个静态问题，并且决策

空间与提交的路径有关。于是，我们通过人为限缩决策空间，筛选出适当大小的低碰撞路径组合，在缩小了的决策空间中通过无差别性列出方程解纳什均衡[2][3]，得到每名玩家的最优反应，近似模拟真正的最优反应。

### 2.3.2 第六关分析

第六关除了复杂度较高，还有着三名玩家。相比于第五关，第六关更强调实时决策，于是我们可以把决策空间缩小到每一天的行动决策构成的子博弈中。我们通过适当的缩减村庄购买物品的情况来降低复杂度，从而用三阶张量存储价值函数来描述不同玩家在其他玩家做出某决策的情况下的收益，进而求得每个玩家在子博弈的纳什均衡下的最优反应。

## 三、 模型假设

为了建立更精确的数学模型，本文根据实际情况建立了一些合理的假设以及条件约束。具体的假设如下所示：

**假设 1：** 玩家的所有行动（如移动、挖矿、购买）都在一天内完成，资源消耗和状态更新发生在当天的结束时刻。起点和村庄的资源库存是无限的，只要玩家资金充足，就可以无限量购买，不会出现缺货的情况。

**假设 2：** 地图是无向图，即区域 A 与区域 B 相邻，则从 A 到 B 和从 B 到 A 的移动均被允许且消耗时间相同（1 天）。假设玩家对地图信息完全掌握。

**假设 3：** 水和食物在被购买、消耗和计算时，均以“箱”为最小整数单位进行，不可分割。

**假设 4：** 在起点、村庄和终点的所有交易都是瞬时完成的，不存在交易延迟或无法交易的情况。

**假设 5：** 第二问的天气满足某种概率分布。

**假设 6：** 正在开采矿山的玩家数不变的情况下，矿山的收益是固定的，不会因为开采时间变长而递减，也不存在挖完的可能性。

**假设 7：** 在第四关中，对于“较少出现沙暴气候”这一模糊描述，我们将其量化为一个具体的低概率。为便于模型求解，我们假设在任何未知的一天，$P(\text{沙暴}) = 0.1$。同时假设“晴朗”和“高温”平分剩余的概率，即 $P(\text{晴朗}) = 0.45$,$P(\text{高温}) = 0.45$。

**假设 8：** 在多玩家博弈模型中，所有玩家都是理性的，且每个玩家的目标都是最大化自身的最终收益。我们假设每个玩家在做决策时，会认为其他玩家也遵循同样的理性原则。玩家之间在游戏开始时不能相互讨论，也不存在明确的通讯，所有决策均基于公开信息（所有人的位置和资源）独立做出。

# 四、 模型的建立和求解

首先，我们将游戏中的设定符号化：

$d$ 表示天数，$d \in \{0, 1, \ldots, d_{max}\}$，其中 $d_{max} = 30$ 为截止日期，游戏从 $d = 0$ 开始。

$p$ 表示玩家所处的区域，$p \in \{1, 2, \ldots, 27\}$，代表地图上的区域编号。

$w_d$、$f_d$ 分别表示第 $d$ 天剩余水量、第 $d$ 天剩余食物量，单位均为箱。

$m_w$、$m_f$ 分别表示水和食物的单位质量，单位均为 $kg$。

$c_w$、$c_f$ 分别表示水和食物的基准价格，单位均为元/箱。

$M$ 表示初始资金，$M = 10000$，单位为元。

$W$ 表示负重上限，$W = 1200$，单位为 $kg$。

$x_d$ 表示第 $d$ 天的天气情况，$x_d \in \{0, 1, 2\}$，其中 1 表示晴朗，2 表示高温，3 表示沙暴。

$y_d$ 表示玩家第 $d$ 天的行动，$y_d \in \{1, 2, 3, 4\}$，其中 1 表示停留，2 表示行走，3 表示挖矿，4 表示购买。

接着将约束条件描述成数学表达式：

- **时间上限约束**

  玩家必须在截止日期前到达终点：

$$\exists d^* \in \{1, \ldots, d_{\text{max}}\} \text{ s.t. } p_{d^*} = p_{\text{end}}$$

  其中 $d^*$ 是到达终点的天数，$p_{\text{end}}$ 表示终点。

- **资源维持约束**

  在游戏结束（到达终点）之前，玩家的水和食物储备不能耗尽。对于任意一天 $d < d^*$，必须满足：

$$w_d \geq 0 \quad \text{且} \quad f_d \geq 0$$

- **负重上限约束**

  在任何一天开始时，玩家携带的水和食物的总质量不能超过负重上限 $W$。对于任意一天 $d \in \{0, 1, \ldots, d_{max}\}$，必须满足：

$$w_d \cdot m_w + f_d \cdot m_f \leq W$$

- **初始资金约束**

  第 0 天购买初始物资的总费用不能超过初始资金 $M$。设 $w_{\text{buy}}$ 和 $f_{\text{buy}}$ 为购买的水和食物数量。

$$w_{\text{buy}} \cdot c_w + f_{\text{buy}} \cdot c_f \leq M$$

  同时，初始资源量即为购买量: $w_0 = w_{\text{buy}}, f_0 = f_{\text{buy}}$。

- **沙暴日行动约束**

  沙暴日时，玩家不可行走：

$$x_d = 3 \implies y_d \neq 2$$

### 4.1 问题一

#### 4.1.1 第一关模型的建立

在游戏中，玩家应尽量减少行走路程过长带来的物资损失，因此，我们首先将地图的每个区域视作一个顶点，在有公共边界的区域间连接无向边，然后利用 BFS 算法求解，最终构建出如下所示的起点、终点、村庄和矿山之间的最短路径示意图（若两点之间路径长度相同，则只表示其中一条；玩家的最优策略不会涉及到起点到终点的直接移动，故两点之间的直接路径不再表示）：

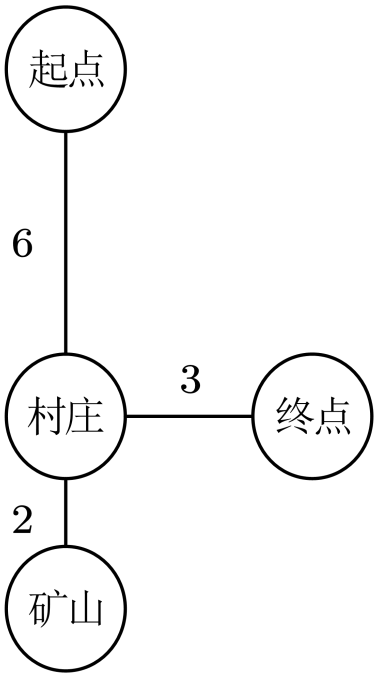


**图 1　第一关最短路径示意图**

问题一已知所有的天气情况，即所有影响决策的变量在一开始已知，设状态量 $dp(p,d,l)$ 表示玩家在 $p$ 位置，第 $d$ 天，物资拥有量为二元组 $l(w_d, f_d)$ 时的持有资金量。

当玩家选择在第 $d-1(d \geq 1)$ 天从 $p_f$ 行走到 $p$ 时，其资金拥有量可表示为：

$$dp(p,d,l) = dp(p_f, d-1, l + 2 * c_{d-1}), \quad p_f \text{与} p \text{相邻且} p_f \neq p, x_{d-1} \neq 3 \tag{1}$$

当玩家选择第 $d(d \geq 1)$ 天在村庄购买 $\Delta l$ 物资时，其资金拥有量可表示为：

$$dp(p,d,l) = dp(p,d,l-\Delta l) + price(\Delta l), \quad p \text{为村庄} \tag{2}$$

其中 $price$ 表示购买 $\Delta l$ 物资消耗的资金数。

当玩家选择在第 $d-1(d \geq 1)$ 天停留时，其资金拥有量可表示为：

$$dp(p,d,l) = dp(p,d-1,l+ct_{d-1}) \tag{3}$$

其中 $ct_d$ 表示玩家第 $d$ 天的基础消耗量。

当玩家选择第 $d-1(d \geq 1)$ 天在矿山挖矿时，其资金拥有量可表示为：

$$dp(p,d,l) = dp(p,d-1,l+3*ct_{d-1}) - profit, \quad p\text{为矿山} \tag{4}$$

其中 $profit$ 表示玩家在矿山挖矿一天获得的基础收益。

公式 (1) ~ (4) 分别刻画了玩家在采取行走、购买、停留或挖矿这四种不同行动时，其状态的演化规则，他们都满足满足无后效性、最优子结构、重叠子问题三个性质，因此可以利用动态规划方法求解。在任意一个决策点，玩家都可能面临多种行动选择，而其最终目标是使游戏结束时持有的资金最大化。根据动态规划的最优性原理，一个最优策略的子策略也必须是最优的。这意味着，为了在第 $d$ 天获得最大资金，玩家必须在第 $d-1$ 天做出能通往此最优结果的决策。基于这一思想，我们将上述分离的公式进行整合，构建出综合状态转移方程：

$$dp(p,d,l) = max \begin{cases} dp(p_f,d-1,l+2*c_{d-1}), & p_f\text{与}p\text{相邻且}p_f \neq p, x_{d-1} \neq 3 \\ dp(p,d,l-\Delta l) + price(\Delta l), & p\text{为村庄} \\ dp(p,d-1,l+ct_{d-1}) & \\ dp(p,d-1,l+3*ct_{d-1}) - profit, & p\text{为矿山} \end{cases} \tag{5}$$

最终最大资金

$$ans = \max_{1 \leq d \leq d_{\max}} \{\mathrm{dp}(p_{\mathrm{end}}, d, l)\} \tag{6}$$

#### 4.1.2 第一关模型的求解

由建立好的动态规划模型，基于公式 (5) 和 (6), 利用程序求解，得到最终最大资金为 10230 元。玩家游戏过程如下所示：


**图 2　第一关线路图**

### 4.1.3 第二关模型的建立

第二关相比于第一关，地图中增加了一个村庄和一个矿山，其余游戏设定不变，因此模型建立过程与第一关类似。

首先利用 BFS 算法，构建出第二关的最短路径示意图：

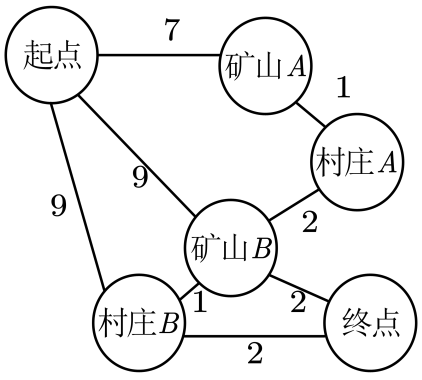


**图 3　第二关最短路径示意图**

接着构建动态规划模型，整理出的综合状态转移模型同样为公式 (5)，最终最大资金表达式为公式 (6)。

### 4.1.4 第二关模型的求解

由建立好的动态规划模型，基于公式 (5) 和 (6), 利用程序求解。

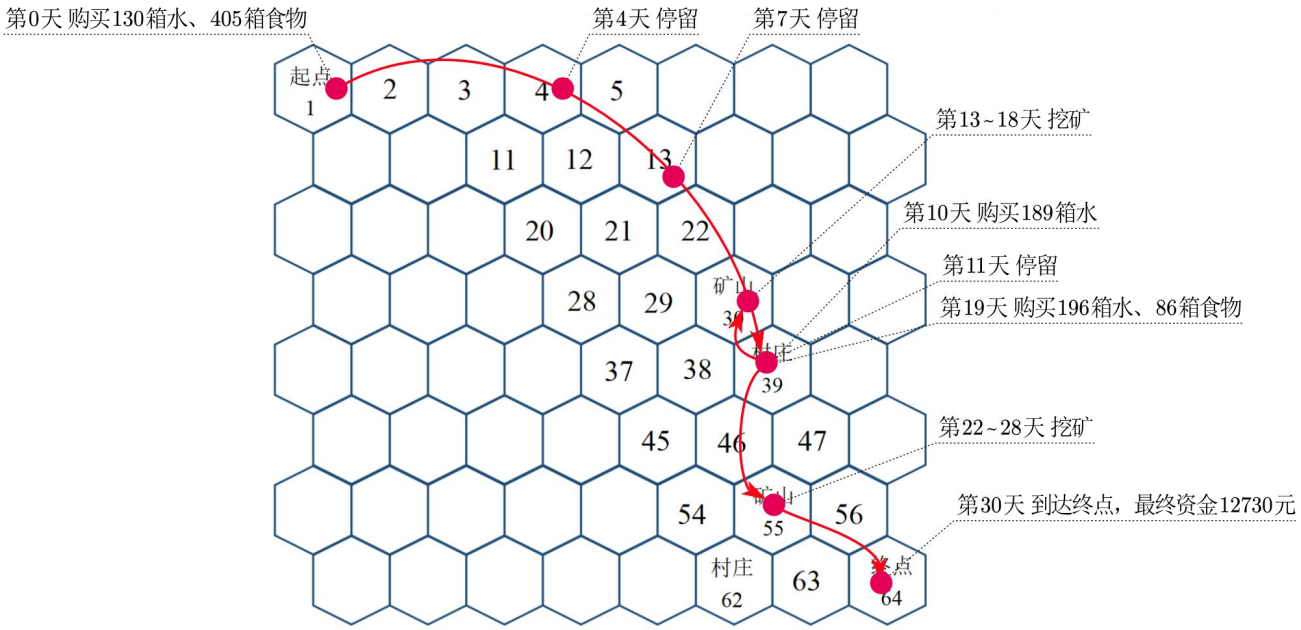


**图 4　第二关线路图**

图 4为求解得到的玩家线路图，并得出最终最大资金为 12730 元。

## 4.2 问题二

### 4.2.1 第三关模型的建立

与第一问的确定性环境不同，玩家只知道当天的天气情况，对未来的天气一无所知，据此进行决策。也就是说，下一步的决策只取决于玩家现在的位置、资源数量等状态。任何预先制定的固定路径都可能因为天气突变而变得不再最优。

所以问题要求解的不再是一个静态的最优路径，而是一个动态的最优决策策略。该策略 π 应当是一个从状态 s 到行动 a 的映射，即 π(s) = a。它能够根据玩家所处的任何一个具体状态（如所在位置、剩余资源等），给出当前最优的行动指令。这个过程本质上是一个序贯决策问题。

马尔可夫决策过程是描述此类问题的经典数学框架，它能够清晰地定义在不确定环境下，智能体（玩家）如何通过与环境交互来学习并执行最优策略，以最大化其长期累积奖励。因此，我们选择 MDP 作为解决本问题的核心模型。

马尔科夫性的定义：如果在 t 时刻的状态 St 满足如下等式，那么这个状态被称为马尔科夫状态，并且满足马尔科夫性：

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, \ldots, S_t]$$

状态 $S_t$ 包含了所有历史相关信息，若 $S_t$ 已知，则 $S_1, S_2, ..., S_{t-1}$ 都可以被抛弃。

马尔科夫状态的一个序列称为马尔科夫过程，在游戏中，玩家在地图中的状态满足马尔可夫性质：即未来只与当前状态有关，与过去如何到达当前状态无关，将来和过去是条件独立的。因此，本问题可抽象为一个马尔科夫决策过程（下称 MDP）。从而进行求解。

一个 MDP 由元组 $(S, A, P, R, )$ 定义：

(1) 状态空间 $S$：我们定义玩家的状态 $s$ 为一个包含其核心信息的元组 $s = (p, w_d, f_d)$，其中 $p$ 为当前区域编号，$w_d$ 和 $f_d$ 分别为剩余的水和食物箱数。

(2) 动作空间 $A(s)$：在状态 $s$ 下，玩家可采取的行动集合，包括原地停留、前往相邻区域、在矿山挖矿等。

(3) 转移概率 $P(s' \mid s, a)$：在状态 $s$ 执行动作 $a$ 后，转移到新状态 $s'$ 的概率，状态转移概率到下一各状态的和为 1。该概率由对未来天气的假设决定。例如，在第四关中，我们可做出合理假设 $P$(沙暴) $= 0.1$,$P$(晴朗) $= 0.45$,$P$(高温) $= 0.45$。

(4) 奖励函数 $R(s, a, s')$：从状态 $s$ 执行动作 $a$ 转移到 $s'$ 后获得的即时奖励。在本模型中，奖励与资金的增量直接相关。例如，挖矿的奖励为 1000 元减去当日消耗资源的成本，而其他行动的奖励则为消耗资源的负成本。但是 $R$ 函数只是当下决策对状态的奖励，对于一个长期的决策判断来说，立即回报函数 $r(s, a)$ 无法说明策略的好坏。因而还需要定义值函数来表明当前状态下策略 $\pi$ 的长期影响。

在本问题中，由于游戏中动作的引入，值函数分为了两种：状态值函数（$V$ 函数），是从状态 $s$ 开始，使用策略 $\pi$ 得到的期望回报值

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

状态动作值函数（Q 函数），是从状态 s 开始，执行动作 a，然后对接下来的状态使用策略 $\pi$ 得到的期望回报值。

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

使用价值迭代方法计算 Q 函数表 Q(s,a) 会导致表过大, 无法计算, 因此使用价值迭代方法计算 V 函数表, 进而得到最优决策分布.

(5) 衰减因子 $\gamma$：设为 1，表示未来的收益与当前收益同等重要。

### 4.2.2 第三关模型的建立

价值迭代算法的理论基石是贝尔曼最优方程。该方程为我们要求解的最优价值函数 V*(s) 提供了一个基本的存在性条件和递归结构。其基本思想为一个状态的最优价值，等于从该状态出发，采取最佳行动后所能获得的“即时奖励”与“所有可能的未来期望回报之和”的期望值。

首先，我们定义最优状态价值函数 V*(s) 为：在遵循某一策略 π 时，从状态 s 出发所能获得的最大期望累积回报。即：$V^*(s) = \max_\pi \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$，其中 r 为即时奖励。

贝尔曼最优方程指出，一个状态的最优价值 V*(s) 与其所有可能的后继状态的最优价值 V*(s') 之间，必须满足以下关系：

$$V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

$\max_{a \in A(s)}$ 表示在一个状态 s，智能体必然会选择那个能使其长期期望回报最大化的行动 a。

$\sum_{s' \in S} P(s'|s,a)[...]$：这是一个期望值计算。由于环境具有不确定性（天气随机），在执行行动 a 后，可能会转移到多个不同的后继状态 s'。该部分通过状态转移概率 $P(s'|s,a)$ 对所有可能结果的价值进行加权平均。

$R(s,a,s')$ 表示执行行动 a 并转移到状态 s' 后获得的即时奖励。在我们的模型中，它对应于资金的变化。

$\gamma V^*(s')$ 表示到达后继状态 s' 后的未来期望回报的折现值。

这个方程本身是一个最优性条件，直接求解很困难。但它为我们提供了一个迭代求解的蓝图。价值迭代算法正是将该方程的右侧部分作为一个“贝尔曼更新算子”，通过反复迭代计算，使得任意初始化的价值函数 $V_0(s)$ 能够逐步收敛至唯一的最优解 V*(s)。

### 4.3 问题三

#### 4.3.1 第五关模型的建立

问题三的 (1) 中的游戏模型存在多个玩家，并且在第 0 天提交此后所有的行动方案，玩家之间相互无法通信且相互之间没有差别，在此基础上，将玩家的整个方案视为一个决策，我们就可以把这个状况建模为一个非合作，静态和对称的纳什均衡模型，接下来我们将探讨这个静态博弈的基本特点。

符号说明：

$N$ 表示玩家集合，$S_i$ 表示玩家策略集合，由于所有玩家都是对称的，所以 $\forall i, j \in N, S_i = S_j = S$，用 $M_i = M$ 表示每个玩家的收益矩阵。由于玩家是对称的，所以 $M = M^T$。

我们定义玩家间的共同知识：

- 玩家都了解整张游戏地图
- 玩家都知道其他玩家目标是获得尽可能多的收益
- 玩家都可以计算出大小为 $k$ 的最优无碰撞路径组合

$\forall i \in N$, 由于游戏给出的竞争惩罚非常高昂, 即对于 $a, b, c \in S, a \neq b, a \neq c$, 且$a \cup b > a \cup c$, 有$M_{aa} >> M_{ab} > M_{ac}$ ，如果为了获取最大收益而前往单人最优路线，很可能会与相同想法的其他玩家产生碰撞。出于对惩罚的恐惧，玩家会选择一条次优但是无碰撞或碰撞较少的路线。在最理想的情况下，玩家之间可以达成 $n!$ 种纯策略纳什均衡，即 $n$ 个玩家占据 $n$ 条最优无碰撞路径组合，使得每个玩家的平均收益最大，并且修改自己的路径会导致收益降低。但由于玩家是对称的且无法谈判和通讯，也不存在任何焦点，我们认为这种纯策略纳什均衡难以达成，即难以分配不同的路径给完全对称的玩家。

于是我们转而考虑混合策略纳什均衡，每个玩家以一定的概率将自己分配到一个路径池 $K$ 中的某条路径上，以让其他玩家无法预测自己的行动来避免与其他玩家相撞。在这种基于概率的分配中，玩家的对某条路径收益期望和总期望可以表示为:

$$E(S_i) = P(\text{选择}S_i\text{不发生碰撞}) * V(S_i) - P(\text{选择}S_i\text{发生碰撞}) * C, S_i \in S$$

$$E(Player) = \sum_{S_i \in S} P(S_i) * E(S_i), Player \in N$$

玩家对不同策略的概率分配有两种因素在起作用: 一种是为了避免碰撞选择更坏路径, 即 $E(S_i)$ 中第二项发挥的作用, 一种是为了提高收益选择消耗更少的路径, 即 $E(S_i)$ 中第一项发挥的作用。由于路径等价于决策，即 $K = S$。出于可计算性的需求，我们选择更小规模的路径池 $K' \in S$，通过平衡避免碰撞和降低消耗两种方法来决定出 $S$ 中大小为 $k$ 的子集作为 $K = S$ 的替代 $K$‘，尽量覆盖路径池为 $S$ 时的纳什均衡高概率路径，使其满足在路径池为 $K'$ 的情况下达成纳什均衡时每个玩家应用其平衡策略 $P^*$ 时，$E(Player)$ 最大。

### 4.3.2 第五关模型的求解

基于以上分析，我们认识到要解决第五关的问题，就要建立最优路径池 $K'$，并且计算出该最优路径组合对应的每个玩家的纳什均衡策略。最优路径池的优化问题是指，对于 $S$ 的子集 $K'$，当玩家间达成纳什均衡时，$E(Player)$ 的值最大。

对于任意一个选出的路径池 $K$，我们建立无差别方程，求纳什均衡下的各策略选择概率 $P(P_1), P(P_2), P(P_3)...$:

$$\begin{cases} E(P_1) = E(P_2) \\ E(P_2) = E(P_3) \\ E(P_3) = E(P_4) \\ ... \\ E(P_{k-1}) = E(P_k) \\ P(P_1) + P(P_2) + ... + P(P_k) = 1 \end{cases}$$

由于组合 $K$ 确定，每个路径的收益（计算碰撞部分的期望）也确定了。因此根据 $k$ 个方程构成的线性方程组就可以解出 $k$ 个未知量 $P(P_1), P(P_2), ..., P(P_k)$，由纳什存在性定理得出该解就是纳什均衡的稳定解。得到解后，再反向计算出单个玩家的总期望：

$$E(Player) = E(P_1) + E(P_2) + E(P_3) + ... + E(P_k)$$

我们可以把上述方法凝练为一个函数，设为 $V(K) = E_K(Player)$，即选择路径组合 $K$ 时的玩家均衡状态下的期望。在建立了评估函数 $V(k)$ 的基础上，我们选择迭代惩罚法生成规模从大到小的低碰撞路径组合，并使用 $V(K)$ 来评价它，进而找到最优的路径池规模和路径池内容。接下来，我将介绍迭代惩罚法的基本思路。迭代惩罚法的核心思想是通过给已经选择了的路径上的边施加边权惩罚，使得新选出的路径尽量避免已经选择了的路径上的边。接下来我们将给出该算法的伪代码描述：

**Algorithm 1** 迭代惩罚法

**Require:** $G = (V, E, C_0)$: 初始化边权为 $C_0$ 的地图.

$N_{players}$: 玩家数量 $n$.

$K$: 路径池的大小 ($K \geq n$).

$P_{value}$: 添加到重复边的惩罚性边权.

**Ensure:** $Portfolio$: 一组 $K$ 个相互不同的路径 $\{P_1^*, P_2^*, \ldots, P_K^*\}$.

$Values$: 单人路径损耗 $\{V_1^*, V_2^*, \ldots, V_K^*\}$.

Initialize $Portfolio\emptyset$

Initialize $Values\emptyset$

Let $C_{current}C_0$ {以最初边权初始化}

**for** $i = 1$ to $K$ **do**

  {Phase 1: 找到单人最优路径}

  $P_i^*$DynamicProgramming$(G, C_{current})$

  {Phase 2: 存储路径和它经过的边}

  Add $P_i^*$ to $Portfolio$

  $V_i^*$CalculateSinglePlayerPayoff$(P_i^*, C_0)$

  Add $V_i^*$ to $Values$

  {Phase 3: 惩罚经过存储路径中的边的路径}

  **for** each edge $e$ in path $P_i^*$ **do**

    $C_{current}(e)C_{current}(e) + P_{value}$

  **end for**

**end for**

**return** $Portfolio, Values$

应用迭代惩罚法得到一系列大小为 $n, n+1, n+2, ..., k$ 的路径池，计算其中最大的 $V(K)$:

$$\max_{K \in \{K_n, K_{n+1}, ..., K_k\}} \{V(K)\}$$

该 $K$ 记为 $K'$，在 $K'$ 上计算纳什均衡概率，就可以得到所有玩家的最优反应。我们编写了 Python 代码，得到第五关的结果为:

路径池 $K$: {1->5->5->6->13, 1->1->1->5->6->13}

收益矩阵 M:

| | **玩家 2** → 路径 1 | **玩家 2** → 路径 2 |
|---|---|---|
| **玩家 1** → 路径 1 | $(8555, 8555)$ | $(8885, 8885)$ |
| **玩家 1** → 路径 2 | $(8885, 8885)$ | $(8555, 8555)$ |

纳什均衡 P：(50%,50%)

该结果意味着，一般情况下玩家应当以 50% 的概率提交路径 1->5->5->6->13，50% 的概率提交路径 1->1->1->5->6->13。

### 4.3.3 第六关模型的建立

第六关不要求在一开始就提交方案，每名玩家的行动都可以参考其他玩家在当天的行动，因此这是一个动态决策问题。单独考察一名玩家在整个游戏过程中每天的决策行为，由于每个玩家在做出决策时的位置不同，我们会发现这个是一个有限玩家数、有限决策数的存在纳什均衡的动态、非对称博弈论问题。因此我们将每天的决策视为一个子博弈，并通过计算自己和其他人的收益矩阵来求解每个子博弈的每个玩家的最优反应，从而得到一般情况下的最优解。

首先，计算收益矩阵，我们使用第三关、第四关用到的 MDP 模型计算执行完某行动后的状态的 V 函数加上即时奖励作为收益，即

$$M_a(S_a, S_b) = R(S, S'_a, a) + V(S'_a) - C_{ab}$$

其中 $C_{ab}$ 代表 a，b 行动发生碰撞的损耗。

类似第五关，每名玩家在某个特定局面下存在决策空间 $S$，$S$ 取决于玩家们所处于的时空位置，存在三种情况：

- 三名玩家处于初始位置，此时玩家是对称的，并且购买行为不会触发惩罚机制，他们根据一定决策购买相同多的物资
- 某名玩家处于村庄，这时其他玩家若考虑他会购买多少物资，状态空间将变得非常庞大，所以我们假定每个玩家在村庄执行的购买行动只有一种可能，就是购买导向最大 V 函数数量的物资
- 所有玩家都处于非终点、非村庄，此时行动的数量有限，仅为原地停留、行走和挖矿（若为矿山）

收益矩阵可以表示为

| Action | **Stay** | **Move To** | **Mine** | **Buy** |
|---|---|---|---|---|
| **Stay** | $V(S'_{SS}) - C_{SS}$ | $V(S'_{SM}) - C_{SM}$ | $V(S'_{SN}) - C_{SN}$ | $V(S'_{SB}) - C_{SB}$ |
| **Move To** | $V(S'_{MS}) - C_{MS}$ | $V(S'_{MM}) - C_{MM}$ | $V(S'_{MN}) - C_{MN}$ | $V(S'_{MB}) - C_{MB}$ |
| **Mine** | $V(S'_{NS}) - C_{NS}$ | $V(S'_{NM}) - C_{NM}$ | $V(S'_{NN}) - C_{NN}$ | $V(S'_{NB}) - C_{NB}$ |
| **Buy** | $V(S'_{BS}) - C_{BS}$ | $V(S'_{BM}) - C_{BM}$ | $V(S'_{BN}) - C_{BN}$ | $V(S'_{BB}) - C_{BB}$ |

#### 4.3.4 第六关模型的求解

在算法的离线阶段中，为评估任意状态的未来潜在价值，首先忽略其他玩家的存在，求解一个单人模式下的 MDP，其目标是最大化玩家的最终期望资金。我们对参与每一次子博弈的每个玩家单独运行第二问所述价值迭代方法，求出其单人情况下的最优值函数，在在线阶段用作参考。

在算法的在线阶段中，我们利用前一阶段求出的 $V$ 函数，判断自己和其他人的行动收益，进而形成收益矩阵。对任意玩家，其做出行动 $a$ 的收益是该行动的即时收益和目标价值函数。考虑到其他玩家的行动可能“妨碍”本玩家的行动，我们计算本玩家的行动和其他玩家的行动同时发生时产生的不同收益，即收益矩阵，该矩阵涉及到三个玩家两两之间的收益关系，因此在数学上表述为一个三阶张量。

考虑到状态空间极为巨大，我们对该三阶张量进行了一定简化，即将购买物资视单个行动，产生的后继状态为某个玩家的单人最优决策（按照最大值函数进行决策）。在此基础上，我们先搜索纯策略纳什均衡，时间复杂度为 $O(|a|^3)$，这里的 a 代表每个人的行动集合。由于我们削减了状态空间，所以计算量可以接受。如果无法计算出纯策略纳什均衡，我们使用线性方程组的方式求解混合策略纳什均衡，进而得到该子博弈局面下每个玩家各自的最优反应。

我们编写了 python 代码求解该问题，结果为：

# 五、 模型的分析与检验

## 5.1 最优性分析

### 5.1.1 问题 1 最优性分析

分析第一关的最优路径，我们可以发现，最好的策略并未选择最短路径直奔终点，而是在初期选择绕道前往矿山。由于未来天气已知，模型精确计算出在矿山进行共 9 天的的挖矿活动，剩余的资金比直接穿越沙漠要高。同时，在前往矿山途中或挖矿期间，模型选择在村庄 15 进行 2 次补给，我们经过人工分析和计算，模拟最优策略，同模型的最优策略进行对比，验证了该策略是解决第一关和第二关的最优策略。

第二关的地图存在多个矿山和村庄。其最优策略与第一关略有不同。模型选择了首先前往村庄 A（区域 39）购买资源进行补给，随后折返矿山 A（区域 30）挖矿 6 天，然后前去村庄 A（区域 39）进行资源补给，之后到达矿山 B（区域 55）挖矿 7 天，最后到达终点，没有经过村庄 B（区域 62）。分析其原因在于村庄二的地理位置较远，距离通往终点的主干道更远，使得挖矿与补给之间的路径成本更高。经过人工验证，我们也无法找到更优的策略。

### 5.1.2 问题 2 最优性分析

问题 2 总体上我们采用了马尔科夫决策模型，并使用期望评估的方法反向推导出了最优值函数，通过最优值函数的定义我们可以知道在期望的视角下由最优质函数导出的策略为最佳策略，即 $\forall V, V_{\pi}^{*}(End) \geq V(End)$。

对第三关而言，我们使用了具体的数据测试最优值函数解的最优性：

| 统计指标 数值 |
| --- |
| 成功率 100% |
| 平均收益（元）8928.20 |
| 最高收益（元）9065.00 |
| 最低收益（元）8825.00 |
| 收益标准差 68.16 |

**表 1　统计指标汇总**

### 5.1.3 问题 3 最优性分析

问题三的最优性由纳什均衡的定义保证。我们已经假设玩家都是理性的，那么他们意识到自己如果可以改变策略从而获取更高收益的话，就一定会改变策略或策略的概率

分布，直到找到一个改变就会降低收益的策略共识，这就是问题三最优性的基础。

在实际模拟的结果（见支撑材料）中，我们发现一部分玩家会主动获取更多收益，另一些玩家则会在概率的作用下选择退让。

## 参考文献


[1] 万海川, 贺知明, 宋腾飞. 基于动态规划理论的改进型价值迭代算法[J]. 雷达科学与技术, 2015, 13(5):501-507.

[2] 廖路祥. 基于博弈论的相邻港口竞争策略研究[D/OL]. 浙江海洋大学, 2024. DOI: 10.27747/d.cnki.gzjhy.2024.000358.

[3] 赵小刚, 陈刚, 胡启平. 纳什均衡及其在计算机科学中的应用[J/OL]. 武汉大学学报(理学版), 2015, 61(5):409-418. DOI: 10.14188/j.1671-8836.2015.05.001.

# 附录 A　文件列表

| 文件名 | 功能描述 |
| --- | --- |
| pb1$_1$.*cpp* | 问题一 (1) 程序代码 |
| pb1$_2$.*cpp* | 问题一 (2) 程序代码 |
| pb2.cpp | 问题二程序代码 |
| pb3.py | 问题三程序代码 |

# 附录 B　代码

pb1_1.cpp

```cpp
#include <iostream>
#include <stack>
#include <vector>


using namespace std;
typedef pair<int, int> pr;

const pr consume[3] = {{5,7}, {8,6}, {10,10}};
const pr mass = {3,2};
const pr price = {5,10};
const int limit = 1200;
const int inital = 10000;
const int profit = 1000;
const int cost[31] = {0, 1, 1, 0, 2, 0, 1, 2, 0, 1, 1,
                      2, 1, 0, 1, 1, 1, 2, 2, 1, 1,
                      0, 0, 1, 0, 2, 1, 0, 0, 1, 1};

const int n = 12;
const int m = 30;

int dp[13][31][401][601];

struct Info {
```

```
25      int p,d,i,j;
26      void print() {printf("p:%d,d:%d,i:%d,j:%d\n",p,d,i,j);}
27      void print(int pp) {printf("p:%d,d:%d,i:%d,j:%d,w:%d\n",p,
        d,i,j,pp);}
28  }info[13][31][401][601];
29
30
31  std::vector<std::vector<int> > graph(20);
32  void add(int a, int b) {graph[a].push_back(b); graph[b].
        push_back(a);}
33
34  void init() {
35      add(1,2);add(2,3);add(3,4);add(4,5);add(5,6);add(6,7);
36      add(7,8);add(8,9);
37      add(7,10);add(10,11);add(11,12);
38  }
39
40  void solve() {
41
42      memset(dp, 0xff, sizeof(dp));
43
44      for(int i = 0; i*mass.first <= limit; i ++) {
45          for(int j = 0; i*mass.first + j*mass.second <= limit;
        j ++) {
46              dp[1][0][i][j] = inital - i*price.first - j*price.
        second;
47          }
48      }
49
50      for(int d = 1; d <= m; d ++) {
51
52          cout << "d = " << d << endl;
53
54          for(int p = 1; p <= n; p ++) {
55
```

```
 for(int i = 0; i*mass.first <= limit; i ++) {
     for(int j = 0; i*mass.first + j*mass.second <= limit; j ++) {
         Info& ifo = info[p][d][i][j];
         int maxx = -1;
         pr c = consume[cost[d]];
         if(cost[d] != 2 && (i+2*c.first)*mass.first+(j+2*c.second)*mass.second <= limit) {
             for(auto x: graph[p]) {
                 int t = dp[x][d-1][i+2*c.first][j+2*c.second];
                 if(t > maxx) {
                     maxx = t;
                     ifo.p = x;
                     ifo.d = d-1;
                     ifo.i = i + 2*c.first;
                     ifo.j = j + 2*c.second;
                 }
             }
         }

         if ((i+c.first)*mass.first+(j+c.second)*mass.second <= limit && dp[p][d-1][i+c.first][j+c.second] > maxx ) {
             maxx = dp[p][d-1][i+c.first][j+c.second];
             ifo.p = p;
             ifo.d = d-1;
             ifo.i = i + c.first;
             ifo.j = j + c.second;
         }

         if(p == 9) {
             if((i+3*c.first)*mass.first+(j+3*c.second)*mass.second <= limit)  {
                 int t = dp[p][d-1][i+3*c.first][j+3*c.
```

```
second];
                if(t >= 0) {
                    t += profit;
                    if(t > maxx ) {
                        maxx = t;
                        ifo.p = p;
                        ifo.d = d-1;
                        ifo.i = i+3*c.first;
                        ifo.j = j+3*c.second;
                    }
                }

            }
        }
        if(maxx < 0) break;

        dp[p][d][i][j] = maxx;
        // if(i%20 == 0 && j %20 == 0)
        // printf("dp[%d][%d][%d][%d] = %d\n", p,d,i,j,
maxx);

     }
 }

 if(p == 7) {
     for(int i = 0; i*mass.first <= limit; i ++) {
         for(int j = 0; i*mass.first + j*mass.second <=
limit; j ++) {
             Info& ifo = info[p][d][i][j];
             int maxx = dp[p][d][i][j];

                 for(int i1 = i; i1 >= 0; i1 --) {
                     for(int j1 = j; j1 >= 0; j1 --) {
                         if(dp[p][d][i-i1][j-j1] < 0) break
;
```

```
                            int t = dp[p][d][i-i1][j-j1] - 2*
price.first*i1 - 2*price.second*j1;
                            if(t < 0) continue;
                            if(t > maxx) {
                                maxx = t;
                                ifo.p = p;
                                ifo.d = d;
                                ifo.i = i-i1;
                                ifo.j = j-j1;
                            }
                        }
                    }
                dp[p][d][i][j] = maxx;
                // printf("dp[%d][%d][%d][%d] = %d\n", p,d,i,j
,maxx);
            }
        }
    }


        }
    }

    int maxi = 0, maxj = 0, maxans = dp[12][30][0][0];
    for(int i = 0; i*mass.first <= limit; i ++) {
        for(int j = 0; i*mass.first + j*mass.second <= limit;
j ++) {
            if(dp[12][30][i][j] > maxans) {
                maxans = dp[12][30][i][j];
                maxi = i;
                maxj = j;
            }
        }
    }

```

```
148     printf("i = %d, j = %d, ans = %d\n", maxi, maxj, dp
    [12][30][maxi][maxj]);
149     Info cur= {12,30,maxi,maxj};
150 
151     while(cur.d > 0) {
152         cur.print(dp[cur.p][cur.d][cur.i][cur.j]);
153         cur = info[cur.p][cur.d][cur.i][cur.j];
154     }
155     cur.print(dp[cur.p][cur.d][cur.i][cur.j]);
156 
157 }
158 
159 int main() {
160 
161     init();
162     solve();
163 
164 
165 
166     return 0;
167 }
```

pb1_2.cpp

```
1  #include <iostream>
2  #include <stack>
3  #include <vector>
4  #include <cstring>
5  
6  
7  using namespace std;
8  typedef pair<int, int> pr;
9  
10 const pr consume[3] = {{5,7}, {8,6}, {10,10}};
11 const pr mass = {3,2};
12 const pr price = {5,10};
```

```
const int limit = 1200;
const int inital = 10000;
const int profit = 1000;
const int cost[31] = {0, 1, 1, 0, 2, 0, 1, 2, 0, 1, 1,
                      2, 1, 0, 1, 1, 1, 2, 2, 1, 1,
                      0, 0, 1, 0, 2, 1, 0, 0, 1, 1};

const int n = 14;
const int m = 30;

// 使用动态分配代替大数组
int ****dp;
struct Info {
    int p,d,i,j;
    void print() {printf("p:%d,d:%d,i:%d,j:%d\n",p,d,i,j);}
    void print(int pp) {printf("p:%d,d:%d,i:%d,j:%d,w:%d\n",p,
   d,i,j,pp);}
};
Info ****info;


std::vector<std::vector<int> > graph(20);
void add(int a, int b) {graph[a].push_back(b); graph[b].
   push_back(a);}

// 内存分配函数
void allocateMemory() {
    // 分配 dp 数组
    dp = new int***[15];
    for(int i = 0; i < 15; i++) {
        dp[i] = new int**[31];
        for(int j = 0; j < 31; j++) {
            dp[i][j] = new int*[401];
            for(int k = 0; k < 401; k++) {
                dp[i][j][k] = new int[601];
```

```
                for(int l = 0; l < 601; l++) {
                    dp[i][j][k][l] = -1;
                }
            }
        }
    }

    // 分配 info 数组
    info = new Info***[15];
    for(int i = 0; i < 15; i++) {
        info[i] = new Info**[31];
        for(int j = 0; j < 31; j++) {
            info[i][j] = new Info*[401];
            for(int k = 0; k < 401; k++) {
                info[i][j][k] = new Info[601];
            }
        }
    }
}

void init() {
    allocateMemory();
    for(int i = 1; i <= 13; i ++) {
        add(i,i+1);
    }
    add(11,13);  // 节点11和13相连
    // 移除了原来的add(12,14)和add(11,14)
}

void solve() {

    // memset(dp, 0xff, sizeof(dp)); // 移除这行，因为我们在分配时已经初始化了

    for(int i = 0; i*mass.first <= limit; i ++) {
```

```
80          for(int j = 0; i*mass.first + j*mass.second <= limit;
    j ++) {
81              dp[1][0][i][j] = inital - i*price.first - j*price.
    second;
82          }
83      }
84
85      for(int d = 1; d <= m; d ++) {
86
87          cout << "d = " << d << endl;
88
89          for(int p = 1; p <= n; p ++) {
90
91      for(int i = 0; i*mass.first <= limit; i ++) {
92          for(int j = 0; i*mass.first + j*mass.second <= limit;
    j ++) {
93              Info ifo = info[p][d][i][j];
94              int maxx = -1;
95              pr c = consume[cost[d]];
96              if(cost[d] != 2&&(i+2*c.first)*mass.first+(j+2*c.
    second)*mass.second <= limit) {
97                  for(auto x: graph[p]) {
98                      int t = dp[x][d-1][i+2*c.first][j+2*c.
    second];
99                      if(t > maxx) {
100                         maxx = t;
101                         ifo.p = x;
102                         ifo.d = d-1;
103                         ifo.i = i + 2*c.first;
104                         ifo.j = j + 2*c.second;
105                     }
106                 }
107             }
108
109             if ((i+c.first)*mass.first+(j+c.second)*mass.
```

```
    second <= limit&&dp[p][d-1][i+c.first][j+c.second] > maxx )
    {
110                 maxx = dp[p][d-1][i+c.first][j+c.second];
111                 ifo.p = p;
112                 ifo.d = d-1;
113                 ifo.i = i + c.first;
114                 ifo.j = j + c.second;
115             }
116
117             if(p == 8 || p == 11) {
118                 if((i+3*c.first)*mass.first+(j+3*c.second)*
    mass.second <= limit) {
119                     int t = dp[p][d-1][i+3*c.first][j+3*c.
    second];
120                     if(t >= 0) {
121                         t += profit;
122                         if(t > maxx ) {
123                             maxx = t;
124                             ifo.p = p;
125                             ifo.d = d-1;
126                             ifo.i = i+3*c.first;
127                             ifo.j = j+3*c.second;
128                         }
129                     }
130
131                 }
132             }
133             if(maxx < 0) break;
134
135             dp[p][d][i][j] = maxx;
136             // if(i%20 == 0  j %20 == 0)
137             // printf("dp[%d][%d][%d][%d] = %d\n", p,d,i,j,
    maxx);
138
139         }
```

```
 }

 if(p == 9 || p == 12) {
     for(int i = 0; i*mass.first <= limit; i ++) {
         for(int j = 0; i*mass.first + j*mass.second <= limit; j ++) {
             Info ifo = info[p][d][i][j];
             int maxx = dp[p][d][i][j];

                 for(int i1 = i; i1 >= 0; i1 --) {
                     for(int j1 = j; j1 >= 0; j1 --) {
                         if(dp[p][d][i-i1][j-j1] < 0) break;
                         int t = dp[p][d][i-i1][j-j1] - 2*price.first*i1 - 2*price.second*j1;
                         if(t < 0) continue;
                         if(t > maxx) {
                             maxx = t;
                             ifo.p = p;
                             ifo.d = d;
                             ifo.i = i-i1;
                             ifo.j = j-j1;
                         }
                     }
                 }
             dp[p][d][i][j] = maxx;
             // printf("dp[%d][%d][%d][%d] = %d\n", p,d,i,j,maxx);
         }
     }
 }


     }
 }
```

```
171
172     int maxi = 0, maxj = 0, maxans = dp[14][30][0][0];
173     for(int i = 0; i*mass.first <= limit; i ++) {
174         for(int j = 0; i*mass.first + j*mass.second <= limit;
    j ++) {
175             if(dp[14][30][i][j] > maxans) {
176                 maxans = dp[14][30][i][j];
177                 maxi = i;
178                 maxj = j;
179             }
180         }
181     }
182
183     printf("i = %d, j = %d, ans = %d\n", maxi, maxj, dp
    [14][30][maxi][maxj]);
184     Info cur= {14,30,maxi,maxj};
185
186     while(cur.d > 0) {
187         cur.print(dp[cur.p][cur.d][cur.i][cur.j]);
188         cur = info[cur.p][cur.d][cur.i][cur.j];
189     }
190     cur.print(dp[cur.p][cur.d][cur.i][cur.j]);
191
192 }
193
194 int main() {
195
196     init();
197     solve();
198
199
200
201     return 0;
202 }
```

pb2.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <iomanip>
#include <algorithm>
#include <locale>
using namespace std;
typedef pair<int, int> pr;

const pr cost[3] = {{3,4}, {9,9}, {10,10}};  // 晴朗、高温、沙暴的消耗
const pr mass = {3,2};
const int price[] = {5,10};
const int limit = 1200;
const int inital = 10000;
const int profit = 200;
const float disturb[2] = {0.5,0.5};  // 天气概率：晴朗0.5，高温0.5

vector<int> graph[26];
void add(int a, int b) {graph[a].push_back(b); graph[b].push_back(a);}

float v[9][11][400][600];

// 存储最优策略：对于每个状态，记录最优动作
struct Policy {
    int next_pos;    // 下一个位置
    int action_type; // 0: 移动, 1: 停留
    bool is_valid;

    Policy() : next_pos(-1), action_type(-1), is_valid(false) {}
    Policy(int pos, int act) : next_pos(pos), action_type(act)
```

```
   , is_valid(true) {}
};

Policy policy[9][11][400][600];

struct Info {
int p,d;
void print() {
printf("p = %d, d = %d\n", p,d);
}
}info [9][11][400][600];

void init() {
add(1,2); add(2,3); add(3,4);
add(1,5);add(5,6);add(6,7);add(7,8);add(8,4);
}

void solve () {

cout << "开始求解MDP..." << endl;

// 初始化策略和值函数
for(int p = 1; p <= 8; p++) {
    for(int d = 0; d <= 10; d++) {
        for(int w = 0; w < 400; w++) {
            for(int f = 0; f < 600; f++) {
                policy[p][d][w][f] = Policy();
                v[p][d][w][f] = -10000; // 初始化为很小的值
            }
        }
    }
}

cout << "初始化完成，设置边界条件..." << endl;

```

```
// 边界条件：第10天在终点4的价值
int boundary_count = 0;
for(int w = 0; w*mass.first <= limit&&w < 400; w ++) {
    for(int f = 0; w * mass.first + f*mass.second <= limit&&f < 600; f ++) {
        v[4][10][w][f] = w*0.5*price[0] + f*0.5*price[1];
        boundary_count++;
    }
}

cout << "边界条件设置完成，共" << boundary_count << "个状态" << endl;

for(int d = 9; d >= 1; d --) {
    cout << "d = " << d << endl;

    for(int p = 1; p <= 8; p ++) {

        for(int w = 0; w*mass.first <= limit&&w < 400; w ++) {
            for(int f = 0; w * mass.first + f*mass.second <= limit&&f < 600; f ++) {

                Info ifo = info[p][d][f][w];
                Policy pol = policy[p][d][w][f];

                float maxx = -10000;
                int best_next_pos = -1;
                int best_action = -1;

                // 尝试移动到相邻位置
                for(auto x : graph[p]) {
                    float newv[2] = {0,0}, newv_s = 0;
                    for(int ll = 0; ll < 2; ll ++) {
                        int new_w = w - 2*cost[ll].first;
                        int new_f = f - 2*cost[ll].second;
```

```
                        if(new_w < 0 || new_f < 0 || new_w >= 400 || new_f >= 600) {
                            newv[ll] = -10000;
                        } else {
                            newv[ll] = v[x][d+1][new_w][new_f];
                        }
                        newv_s += newv[ll] * disturb[ll];
                    }
                    if(newv_s > maxx) {
                        maxx = newv_s;
                        ifo = {x, d+1};
                        best_next_pos = x;
                        best_action = 0; // 移动
                    }
                }

                // 尝试停留在原地
                {
                    float newv[2] = {0,0}, newv_s = 0;
                    for(int ll = 0; ll < 2; ll ++) {
                        int new_w = w - cost[ll].first;
                        int new_f = f - cost[ll].second;
                        if(new_w < 0 || new_f < 0 || new_w >= 400 || new_f >= 600) {
                            newv[ll] = -10000;
                        } else {
                            newv[ll] = v[p][d+1][new_w][new_f];
                        }
                        newv_s += newv[ll] * disturb[ll];
                    }

                    if(newv_s > maxx) {
                        maxx = newv_s;
```

```
                        ifo = {p, d+1};
                        best_next_pos = p;
                        best_action = 1; // 停留
                    }
                }

                v[p][d][w][f] = maxx;
                pol = Policy(best_next_pos, best_action);
            }
        }
    }
}


float maxx = -1, ww, ff;
for(int w = 0; w*mass.first <= limit&&w < 400; w ++) {
    for(int f = 0; w * mass.first + f*mass.second <= limit&&f < 600; f ++) {
        v[1][0][w][f] = 10000 + v[1][1][w][f] - (w*price[0] + f*price[1]);
        if(v[1][0][w][f] > maxx) {
            maxx = v[1][0][w][f];
            ww =w, ff = f;
        }
    }
}

    cout << "最优期望价值: " << maxx << ", 最优初始购买: 水=" << ww << "箱, 食物=" << ff << "箱" << endl;
}

// 测试给定天气序列下的策略表现
float testPolicy(vector<int> weather_sequence, int init_w, int init_f) {
    int pos = 1;
```

```
159     int day = 1;
160     int water = init_w;
161     int food = init_f;
162     float money = inital - init_w * price[0] - init_f * price
    [1];
163 
164     cout << "\n=== 策略测试 ===" << endl;
165     cout << "初始状态: 位置=" << pos << ", 水=" << water << ",
     食物=" << food << ", 资金=" << money << endl;
166 
167     while(day <= 10) {
168         if(pos == 4) {
169             // 到达终点，退回剩余资源
170             money += water * 0.5 * price[0] + food * 0.5 *
    price[1];
171             cout << "第" << day-1 << "天到达终点，最终资金: "
    << money << endl;
172             return money;
173         }
174 
175         // 获取当前状态的最优策略
176         if(water >= 400 || food >= 600 || !policy[pos][day][
    water][food].is_valid) {
177             cout << "第" << day << "天: 状态无效或资源耗尽" <<
     endl;
178             return -1; // 失败
179         }
180 
181         Policy pol = policy[pos][day][water][food];
182         int weather = weather_sequence[day-1]; // 天气：0=晴
    朗，1=高温
183 
184         cout << "第" << day << "天: 位置=" << pos << ", 天气="
     << (weather==0?"晴朗":"高温")
185              << ", 水=" << water << ", 食物=" << food;
```

```
186
187         // 执行动作
188         int next_pos = pol.next_pos;
189         int consume_multiplier = (pol.action_type == 0) ? 2 :
    1; // 移动消耗2倍，停留消耗1倍
190
191         int water_consume = cost[weather].first *
    consume_multiplier;
192         int food_consume = cost[weather].second *
    consume_multiplier;
193
194         if(water < water_consume || food < food_consume) {
195             cout << " -> 资源不足，游戏失败!" << endl;
196             return -1;
197         }
198
199         water -= water_consume;
200         food -= food_consume;
201         pos = next_pos;
202
203         cout << " -> " << (pol.action_type == 0 ? "移动到" : "
    停留在") << pos
204                   << ", 消耗水=" << water_consume << ", 食物=" <<
    food_consume << endl;
205
206         day++;
207     }
208
209     // 10天结束但未到达终点
210     cout << "10天结束未到达终点，游戏失败!" << endl;
211     return -1;
212 }
213
214 // 生成所有可能的10天天气序列并测试
215 void testAllWeatherSequences(int init_w, int init_f) {
```

```
    cout << "\n=== 测试所有可能的天气序列 ===" << endl;

    vector<float> results;
    int total_sequences = 1024; // 2^10
    int success_count = 0;
    float total_success_money = 0;

    for(int seq = 0; seq < total_sequences; seq++) {
        vector<int> weather(10);
        int temp = seq;
        for(int i = 0; i < 10; i++) {
            weather[i] = temp % 2;
            temp /= 2;
        }

        cout << "\n天气序列 " << seq+1 << ": ";
        for(int i = 0; i < 10; i++) {
            cout << (weather[i] == 0 ? "晴" : "热");
        }

        float result = testPolicy(weather, init_w, init_f);
        if(result > 0) {
            results.push_back(result);
            success_count++;
            total_success_money += result;
        }

        if(seq < 10 || seq % 100 == 0) { // 只显示前几个和每100个的结果
            cout << " -> 结果: " << (result > 0 ? to_string(result) : "失败") << endl;
        }
    }

    cout << "\n=== 统计结果 ===" << endl;
```

```
249     cout << "成功序列数: " << success_count << "/" <<
    total_sequences << endl;
250     cout << "成功率: " << (float)success_count /
    total_sequences * 100 << "%" << endl;
251 
252     if(success_count > 0) {
253         cout << "平均成功资金: " << total_success_money /
    success_count << endl;
254 
255         // 找最好和最坏结果
256         float best = *max_element(results.begin(), results.end
    ());
257         float worst = *min_element(results.begin(), results.
    end());
258         cout << "最好结果: " << best << endl;
259         cout << "最坏结果: " << worst << endl;
260     }
261 }
262 
263 
264 
265 
266 int main() {
267 
268     init();
269     solve();
270 
271     // 找到最优初始策略
272     float best_value = -1;
273     int best_w = 0, best_f = 0;
274 
275     for(int w = 0; w*mass.first <= limit&&w < 400; w ++) {
276         for(int f = 0; w * mass.first + f*mass.second <= limit
    &&f < 600; f ++) {
277             float value = 10000 + v[1][1][w][f] - (w*price[0]
```

```
    + f*price[1]);
278             if(value > best_value) {
279                 best_value = value;
280                 best_w = w;
281                 best_f = f;
282             }
283         }
284     }
285 
286     cout << "最优策略: 初始购买水=" << best_w << "箱, 食物="
    << best_f << "箱" << endl;
287     cout << "期望价值: " << best_value << endl;
288 
289     // 测试几个特定的天气序列
290     cout << "\n=== 测试特定天气序列 ===" << endl;
291 
292     // 全晴朗
293     vector<int> all_sunny = {0,0,0,0,0,0,0,0,0,0};
294     cout << "\n全晴朗天气:";
295     for(int w : all_sunny) cout << (w==0?"晴":"热");
296     testPolicy(all_sunny, best_w, best_f);
297 
298     // 全高温
299     vector<int> all_hot = {1,1,1,1,1,1,1,1,1,1};
300     cout << "\n全高温天气:";
301     for(int w : all_hot) cout << (w==0?"晴":"热");
302     testPolicy(all_hot, best_w, best_f);
303 
304     // 交替天气
305     vector<int> alternate = {0,1,0,1,0,1,0,1,0,1};
306     cout << "\n交替天气:";
307     for(int w : alternate) cout << (w==0?"晴":"热");
308     testPolicy(alternate, best_w, best_f);
309 
310     // 如果想测试所有序列（可能很慢），取消下面注释
```

```
    // testAllWeatherSequences(best_w, best_f);

    return 0;
}
```

pb3.py

```python
import heapq
import numpy as np
from collections import defaultdict

# -- 核心修正：严格使用第五关附件中的所有参数 --
CONFIG = {
    "players": 2, "days_limit": 10, "weight_limit": 1200, "initial_cash": 10000,
    "mine_income": 200, "water_kg": 3, "food_kg": 2, "water_price": 5, "food_price": 10,
    "base_consumption": { "Sunny": {"water": 3, "food": 4}, "Hot": {"water": 9, "food": 9} },
    "weather": [None, "Sunny", "Hot", "Sunny", "Sunny", "Sunny", "Sunny", "Hot", "Hot", "Hot", "Hot"],
    "adj": {
        1: [2, 4, 5], 2: [1, 3, 4], 3: [2, 4, 8], 4: [1, 2, 3, 5, 7],
        5: [1, 4, 6], 6: [5, 7, 13], 7: [4, 5, 6, 12], 8: [3, 9],
        9: [8, 10, 11], 10: [9, 11], 11: [9, 10, 12], 12: [7, 11, 13], 13: [6, 12]
    },
    "start_node": 1, "end_node": 13, "mine_node": 9
}

def get_consumption(day, action):
    weather = CONFIG["weather"][day]
    base = CONFIG["base_consumption"][weather]
    w_consum, f_consum = base["water"], base["food"]
```

```
    multiplier = {"stay": 1, "move": 2, "mine": 3}.get(action, 1)
    return w_consum * multiplier, f_consum * multiplier

def find_k_best_paths(k):
    paths_pool = []
    penalty_on_edges = defaultdict(int)
    for _ in range(k):
        # -- 核心修正：寻路算法的目标是最小化金钱花费（花费-收入）--
        # 状态: (总金钱花费, day, pos, path)
        pq = [(0, 0, CONFIG["start_node"], [(0, CONFIG["start_node"])])]
        visited = defaultdict(lambda: float('inf'))
        visited[(0, CONFIG["start_node"])] = 0
        best_path_found = None

        while pq:
            monetary_cost, day, pos, path = heapq.heappop(pq)
            if pos == CONFIG["end_node"] and day <= CONFIG["days_limit"]:
                final_path = path
                for stop_day in range(day + 1, CONFIG["days_limit"] + 1):
                    final_path.append((stop_day, pos))
                best_path_found = final_path
                break

            if monetary_cost > visited[(day, pos)] or day >= CONFIG["days_limit"]:
                continue

            next_day = day + 1
            possible_next_steps = [(pos, "stay")] + [(n, "move") for n in CONFIG["adj"].get(pos, [])]
```

```
51            for next_pos, action_type in possible_next_steps:
52                current_action = action_type
53                if action_type == "stay" and pos == CONFIG["
    mine_node"]:
54                    current_action = "mine"
55
56                w_c, f_c = get_consumption(next_day,
    current_action)
57                resource_cost = w_c * CONFIG['water_price'] +
    f_c * CONFIG['food_price']
58                income = CONFIG['mine_income'] if
    current_action == 'mine' else 0
59
60                # 新的成本函数：净金钱花费
61                net_monetary_cost = resource_cost - income
62
63                edge_penalty = penalty_on_edges.get((day, pos,
     next_pos), 0)
64                new_total_cost = monetary_cost +
    net_monetary_cost + edge_penalty
65
66                if new_total_cost < visited[(next_day,
    next_pos)]:
67                    visited[(next_day, next_pos)] =
    new_total_cost
68                    heapq.heappush(pq, (new_total_cost,
    next_day, next_pos, path + [(next_day, next_pos)]))
69
70        if best_path_found:
71            paths_pool.append(best_path_found)
72            for i in range(len(best_path_found) - 1):
73                d, p_curr = best_path_found[i]
74                _, p_next = best_path_found[i+1]
75                penalty_on_edges[(d, p_curr, p_next)] += 500 #
     施加惩罚以寻找不同路径
```

```
76
77      return paths_pool
78
79  def calculate_actual_consumption(path_self, path_opponent):
80      total_w, total_f, total_income = 0, 0, 0
81      for day in range(1, len(path_self)):
82          pos_self, prev_pos_self = path_self[day][1], path_self
    [day-1][1]
83          pos_opp, prev_pos_opp = path_opponent[day][1],
    path_opponent[day-1][1]
84
85          action_self = "mine" if pos_self == CONFIG["mine_node"
    ] and pos_self == prev_pos_self else ("move" if pos_self !=
    prev_pos_self else "stay")
86          action_opp = "mine" if pos_opp == CONFIG["mine_node"]
    and pos_opp == prev_pos_opp else ("move" if pos_opp !=
    prev_pos_opp else "stay")
87
88          move_collision = (action_self == "move" and action_opp
     == "move" and prev_pos_self == prev_pos_opp and pos_self ==
     pos_opp)
89          mine_collision = (action_self == "mine" and action_opp
     == "mine" and pos_self == pos_opp)
90
91          w_c, f_c = get_consumption(day, action_self)
92          income = CONFIG["mine_income"] if action_self == "mine
    " else 0
93
94          if move_collision: w_c *= 2; f_c *= 2
95          if mine_collision:
96              base_cons = CONFIG["base_consumption"][CONFIG["
    weather"][day]]
97              w_c, f_c = base_cons["water"] * 3, base_cons["food
    "] * 3
98              income /= 2
```

```
99
100         total_w += w_c; total_f += f_c; total_income += income
101
102     return total_w, total_f, total_income
103
104 def calculate_payoff(path1, path2):
105     w1, f1, income1 = calculate_actual_consumption(path1,
    path2)
106     w2, f2, income2 = calculate_actual_consumption(path2,
    path1)
107     payoffs = []
108     for w, f, income in [(w1, f1, income1), (w2, f2, income2)
    ]:
109         cost = w * CONFIG["water_price"] + f * CONFIG["
    food_price"]
110         weight = w * CONFIG["water_kg"] + f * CONFIG["food_kg"
    ]
111
112         if weight > CONFIG["weight_limit"] or cost > CONFIG["
    initial_cash"]:
113             payoffs.append(-float('inf'))
114         else:
115             final_cash = CONFIG["initial_cash"] - cost +
    income
116             payoffs.append(final_cash)
117     return tuple(payoffs)
118
119 def solve_msne(payoff_matrix):
120     k = len(payoff_matrix)
121     A = np.zeros((k, k))
122     for i in range(k - 1):
123         for j in range(k):
124             A[i, j] = payoff_matrix[i][j][0] - payoff_matrix[i
    + 1][j][0]
125     A[k - 1, :] = 1
```

```
    b = np.zeros(k); b[k - 1] = 1
    try:
        probabilities = np.linalg.solve(A, b)
        return probabilities
    except np.linalg.LinAlgError:
        return None

def format_path_brief(path, path_id):
    w, f, income = calculate_actual_consumption(path, path)
    cost = w * CONFIG['water_price'] + f * CONFIG['food_price']
    final_cash = CONFIG['initial_cash'] - cost + income
    route = "->".join([str(p[1]) for p in path])
    return f"路径 {path_id}: 单人最终资金 {final_cash:.0f}元。路线概览: {route[:50]}..."

if __name__ == "__main__":
    K_PATHS = 2
    print(f"--- 第一步: 寻找前 {K_PATHS} 条最优路径 (目标: 最大化资金) ---")
    path_pool = find_k_best_paths(K_PATHS)
    for i, p in enumerate(path_pool):
        print(format_path_brief(p, i + 1))

    print(f"\n--- 第二步: 构建 {K_PATHS}x{K_PATHS} 支付矩阵 ---")
    payoff_matrix = [[(0,0) for _ in range(K_PATHS)] for _ in range(K_PATHS)]
    for i in range(K_PATHS):
        for j in range(K_PATHS):
            payoff_matrix[i][j] = calculate_payoff(path_pool[i], path_pool[j])

    print("支付矩阵 (玩家1收益, 玩家2收益):")
    header = "          " + "".join([f"  P2->路径{j+1}      "
```

```
    for j in range(K_PATHS)])
155     print(header)
156     for i in range(K_PATHS):
157         row_str = f"P1->路径{i+1} "
158         for j in range(K_PATHS):
159             row_str += f"({payoff_matrix[i][j][0]:.0f}, {
    payoff_matrix[i][j][1]:.0f})   "
160         print(row_str)
161 
162     print("\n--- 第三步：求解混合策略纳什均衡 ---")
163     probabilities = solve_msne(payoff_matrix)
164 
165     if probabilities is not None and np.all(probabilities >=
    -1e-9):
166         print("计算出的均衡策略概率为:")
167         for i, prob in enumerate(probabilities):
168             print(f"  - 选择 路径 {i+1} 的概率: {prob:.2%}")
169         expected_payoff = np.dot(probabilities, [payoff_matrix
    [0][j][0] for j in range(K_PATHS)])
170         print(f"\n在此均衡下，每位玩家的期望收益为: {
    expected_payoff:.2f} 元。")
171         print("\n结论：基于正确的地图数据和以最大化资金为目标
    的寻路算法，我们得到了最终的混合策略解。")
172     else:
173         print("无法计算出有效的混合策略纳什均衡。")
```