赛区评阅编号（由赛区组委会填写）：

# 2021 年高教社杯全国大学生数学建模竞赛

# 承　诺　书

我们仔细阅读了《全国大学生数学建模竞赛章程》和《全国大学生数学建模竞赛参赛规则》（以下简称 “竞赛章程和参赛规则”，可从 http://www.mcm.edu.cn 下载）。

我们完全清楚，在竞赛开始后参赛队员不能以任何方式，包括电话、电子邮件、“贴吧"、QQ 群、微信群等，与队外的任何人（包括指导教师）交流、讨论与赛题有关的问题；无论主动参与讨论还是被动接收讨论信息都是严重违反竞赛纪律的行为。

我们完全清楚，在竞赛中必须合法合规地使用文献资料和软件工具，不能有任何侵犯知识产权的行为。否则我们将失去评奖资格，并可能受到严肃处理。

**我们以中国大学生名誉和诚信郑重承诺，严格遵守竞赛章程和参赛规则，以保证竞赛的公正、公平性。如有违反竞赛章程和参赛规则的行为，我们将受到严肃处理。**

我们授权全国大学生数学建模竞赛组委会，可将我们的论文以任何形式进行公开展示（包括进行网上公示，在书籍、期刊和其他媒体进行正式或非正式发表等）。

我们参赛选择的题号（从 A/B/C/D/E 中选择一项填写）：_____

我们的报名参赛队号（12 位数字全国统一编号）：_____

参赛学校（完整的学校全称，不含院系名）：_____

参赛队员 (打印并签名) ：1._____

指导教师或指导教师组负责人　(打印并签名)：_____

（指导教师签名意味着对参赛队的行为和论文的真实性负责）

日期：_____年____月____日

**（请勿改动此页内容和格式。此承诺书打印签名后作为纸质论文的封面，注意电子版论文中不得出现此页。以上内容请仔细核对，如填写错误，论文可能被取消评奖资格。）**

赛区评阅编号：
（由赛区填写）

全国评阅编号：
（全国组委会填写）

# 2021 年高教社杯全国大学生数学建模竞赛

## 编 号 专 用 页

赛区评阅记录（可供赛区评阅时使用）：

| | | | | | | |
|---|---|---|---|---|---|---|
| 评阅人 | | | | | | |
| 备注 | | | | | | |

送全国评阅统一编号:
（赛区组委会填写）

**（请勿改动此页内容和格式。此编号专用页仅供赛区和全国评阅使用，参赛队打印后装订到纸质论文的第二页上。注意电子版论文中不得出现此页。）**

# 穿越沙漠游戏的最优策略分析

## 摘要

本论文基于 2020 年数学建模比赛 B 题，对穿越沙漠游戏中的最优解进行分析与解答。其中，条件分别在天气是否已知、是否有多名玩家等情形，借助相关算法，制定并解得相应的最优策略。其中，相关答案见附件。

在问题一中，全程天气状况、地形已知，且仅考虑一名玩家。先对沙漠地图进行图论分析，并使用 Flord-Warshall 算法抽象，得到关键节点。并在此基础上，使用 DFS 算法，对地图进行深度优先搜索，以穷举法的方式得到初始状态下的最优解。此外，使用逆推的方式，先确立目标，再回溯到出发点，该方式具有更强的目的性与针对性。由于变量较多，本题目使用的编程思想为面向对象编程。

第二问中，由于并没有给出具体的天气状况，行动路线、时间分配等也充满了不确定性。因此需要引入动态规划的概念与仿真实验。为了更好的应对环境带来的不确定性，可以采用更激进的方式来规划路线。在规划实际的路线与时间中，我们依然使用以资金、资源等约束条件建立规划模型，继而得到参数之间的关系。基于图论抽象图，对于分析不同节点属性即可得到最终的一般策略，并最终仿真进行验证。

第三问中，由于引入了多玩家博弈，导致资源等约束条件在不同玩家之间相互制衡。在第一问和第二问的基础上，进入完全静态的博弈模型。通过纳什均衡建立在博弈的条件下的线性规划模型，求得混合策略。在针对天气的不确定性方面，我们也使用了马尔可夫决策法，引入风险接受程度，在实际情况下使用相应的策略。此外，通过蒙特卡洛模拟分析得出风险接受程度对决策的影响，使得问题结论更加普遍，得出一般的策略。

**关键字：**深度优先搜索算法 图论 静态博弈 马尔可夫决策 蒙特卡洛算法

# 1 概述

## 1.1 问题背景

### 1.1.1 数学抽象模型

考虑如下的小游戏：玩家凭借一张地图，利用初始资金购买一定数量的水和食物（包括食品和其他日常用品），从起点出发，在沙漠中行走。途中会遇到不同的天气，也可在矿山、村庄补充资金或资源，目标是在规定时间内到达终点，并保留尽可能多的资金。

从数学角度分析，可抽象为在约束条件的前提下，且在不确定因素的风险影响下，在可能的多人资源博弈的情况下，获得资金剩余量的最优解。

### 1.1.2 目标任务

问题一：在一名玩家的情况下，且未来天气状况已知，对于给定地图时，存在一游戏最优策略，通过走最优路线与时间安排，使得最终剩余资金最多。

问题二：在不确定天气状况的情况下，对于给定地图，通过考虑关键点和不同的决策阶段，建立一人的决策路径，使得利润最大化。

问题三：如何在多人游戏且玩家之间会相互影响下，在不确定的环境下，预先做出策略并得到动态规划下的最优策略。最终得到博弈中的普遍结论与最优解。

## 1.2 问题分析

在本题目的游戏中，最佳策略由路径、物资、天气三个方面构成。在第一问一个玩家的前提下，需要充分考虑三个自变量的情况，优化出花费最少的一条路径来。针对物资，需比较水和食物的携带率，在不超过负重的情况下，携带尽可能多的携带率高的物资，以合理安排水和食物的购买数量。得到最佳策略后，进一步优化算法，降低算法复杂度。本题目涉及到的算法有 Dijstra 算法、DFS 算法、图论、动态规划、静态博弈等内容，将在下面论文与程序中体现。

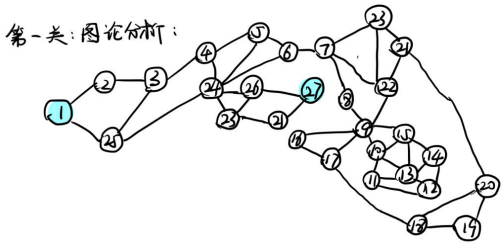
# 2 图论与贪心算法

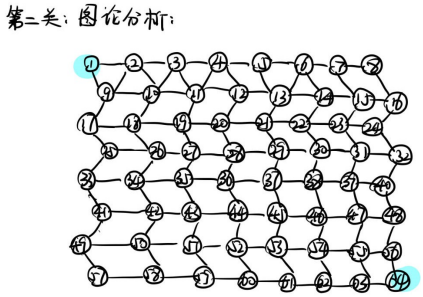
## 2.1 图论分析与有向图模型

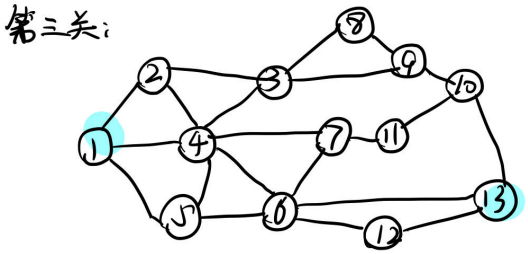
如图，通过 Dijstra 算法，建立点之间的关系，并用图论的数学方法，分析出关系矩阵与距离矩阵。该算法是典型最短路径算法，用于计算一个节点到其他节点的最短路径。它的主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止。具体步骤如

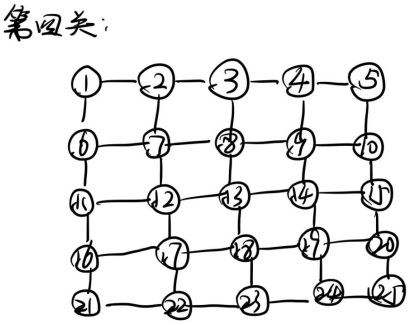
下：通过 Dijkstra 计算图中的最短路径时，我们指定原点为开始点。再引入两个集合 A 和 B，分别记录已求出的最短路径的顶点和相应的最短路径的长度；另外一个则是记录还没有找到最短路径的顶点和距离（图论中的抽象距离）。此外，通过将已经过的点加入对应的路径，重复造作，直到遍历完所有的点。

其中，对地图分析之后的拓扑图如下：



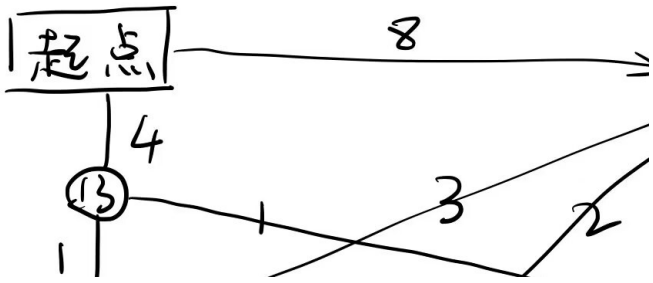




## 2.2 网格模型图论优化

在前两问中的沙漠地图里，先假定所有区域天气相同，除了起点等特殊点之外，其余各点是等效的。考虑到游戏目标为寻找最多资金剩余，因此路线并不需要记录或得出。也就是说：我们在分析时，仅需要考虑起点、终点、矿山和村庄的关系即可，因为只有这些节点才决定了玩家做最优解时的路线和规划。此外，在普通节点中，便于优化后续的 DFS 算法，可以拿出两个具有代表性的点来，分别取 13 和 19。（该模型的引入与分析均以第一关的情况为例，在后续的关卡中，原理与本分析思路大致相同）




前两关中，因为提前假设天气已知且所有区域天气均相同，因此可暂时不考虑天气对地图的影响。但在第三关、第四关中，由于天气状态不确定（仅有一个天气约束条件），所以存在

路径决策问题。也就是说，在第三四问的条件下，地图中的各个点不能做等效的简化处理。为了找到累计状态最多的位置，或者称作权重最高的位置，可以引入 Dijkstra 算法与匹配算法进行模型简化。方法如下：首先使用穷举法，对天气情况进行仿真[1]，在随机天气的情况下，找出不同的最优解情况。在众多最优解的路径与时间中，找到某点，使得路径重合次数最多，或者累计时间经过最长的点。在获得点雨点之间的全部路径之后，使用匹配算法，得到关键节点。

进而得出：第一关的距离矩阵与关系矩阵：

$$\begin{matrix} 0 & 6 & 8 & 3 \\ 6 & 0 & 2 & 3 \\ 8 & 2 & 0 & 5 \\ 3 & 3 & 3 & 0 \end{matrix}$$

关系矩阵为：

$$\begin{matrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$

第二关的距离矩阵与关系矩阵为：

$$\begin{matrix} 0 & 7 & 8 & 9 & 9 & 11 \\ 7 & 0 & 1 & 3 & 4 & 4 \\ 8 & 1 & 0 & 3 & 3 & 3 \\ 9 & 3 & 2 & 0 & 1 & 2 \\ 9 & 4 & 3 & 1 & 0 & 2 \\ 11 & 4 & 3 & 2 & 2 & 0 \end{matrix}$$

关系矩阵为：

$$\begin{matrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$

在第三关与第四关的地图中，与第一关类似，也可写出相应的矩阵：

$$\begin{matrix} 0 & 6 & 8 & 3 \\ 6 & 0 & 2 & 3 \\ 8 & 2 & 0 & 5 \\ 3 & 3 & 3 & 0 \end{matrix}$$

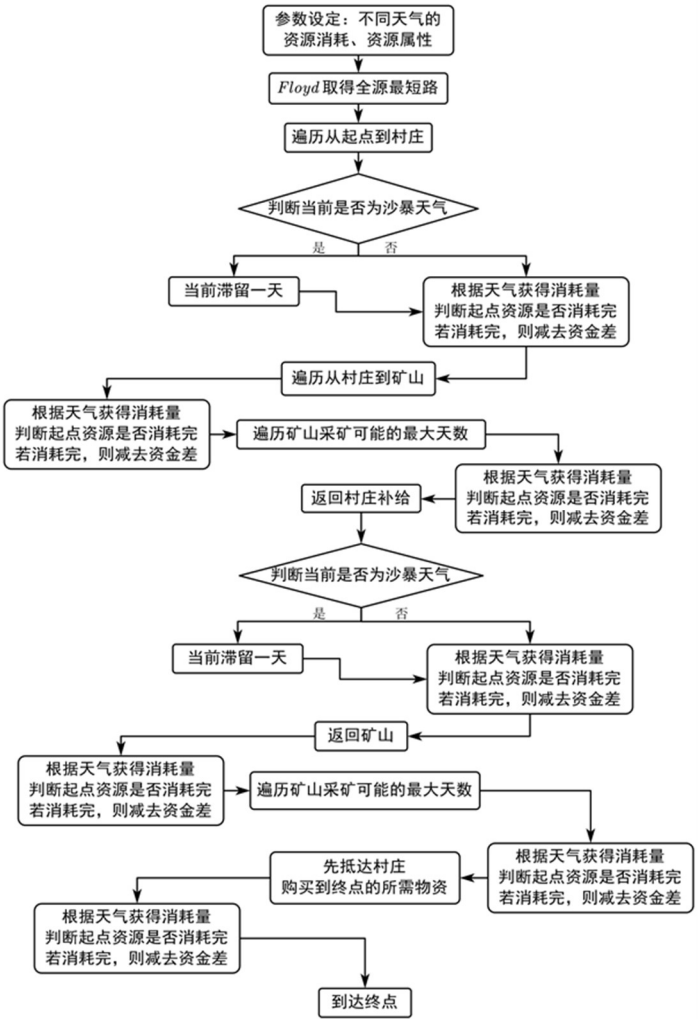
## 2.3 贪心算法的引入与基于随机化贪心算法的求解

考虑到随机化贪心算法，或者说不直接取贪心策略的最优值而是从此最优值附近的一定区间内

[1] 从概论的角度来说，仿真的结果具有偶发性。但在仿真次数足够多的情况下，我们可近似认为频率即为概论。我们在此处仿真了 1000 次。

找到一个目标函数最优值，在贪心原则证明正确的前提下，通过足够次数的随机试验，即可得到最优解。随机化贪心算法适合动态规划类问题的求解，且其复杂度远小于搜索算法，故本文采用随机化贪心算法求解带天气变化因素的优化模型。

其中，流程图如下：




# 3 约束条件的建立与状态转移方程

本题目使用面向对象分析与编程方法，对不同属性的地点进行逐个分析。每个点的约束条件类似，可先对一种进行详细分析。毕竟 DFS 算法是一种基于迭代的算法，不同节点类型对分析约束条件影响不大。

## 3.1 阶段变量与状态变量表示

本题解的变量命名尽量使用英文翻译，以便阅读，并不符合数学的命名规范。

初始变量与最终目标量如下表：

| | | | |
|---|---|---|---|
| weight_water | 单位水的重量 | fund_basic | 基础资金 |
| weight_food | 单位食物的重量 | income | 基础收入 |
| price_water | 单位水的价格 | date | 限额日期 |
| price_food | 单位食物的价格 | fund | 保有资金量 |
| n | 地点属性总数 | optimum_origin_water | 优化后水的数量 |
| capacity | 背包容量 | optimum_origin_food | 优化后食物的数量 |

中间变量表示如下表：

| | | | |
|---|---|---|---|
| days | 时间变动形参 | water | 水量形参 |
| dayss | 天数属性 | food | 食物量形参 |
| now | 现在为第几天 | type | 地点属性 |
| weight | 目前背包重量 | consumption_food | 食物消耗量形参 |
| fund_now | 资金变动形参 | consumption_water | 水消耗量形参 |
| cost_food | 单次食物花销 | cost_water | 单次水花销 |
| attday | 天数变动常量 | position | 目前所处地点 |

以村庄（type=1）为例：具有如下的一些约束条件：

### 3.2 购买物资约束

第 1 天购买的水和食物即为初始物资：

cost_food=consumption_food[days][now][i]

cost_water=consumption_water[days][now][i]

玩家第一天购买的物资总花费不能大于初始资金：

food+water<cost_food+cost_water

玩家在第 1 天的剩余资金为初始资金减去购置物资的花费：

food_now=food-cost_food

对食物的消耗与购买进行分析，与第 1 天相同。当食物量大于所要花费的食物价格时：

food_now=food-cost_food

否则：

cost=cost-2*(cost_food-food)*price_food

weight_now=weight_now-(cost_food-food)*weight_food

weight=weight-cost_food*weight_food-cost_water*weight_water

水资源的消耗与食物同理。

### 3.3 玩家购买约束

玩家经过或在村庄停留时可用剩余的初始资金或挖矿获得的资金随时购买水和食物，每箱价格为基准价格的 2 倍 （玩家购买物资的时刻是当天所有行动结束后，也就是说，需要耽误一天的时间）。

资金的保有量可如下表示：

fund_now=fund_now-2*(cost_food*price_food+cost_water*price_water)

fund_now=fund_now+income

资金的花销可表示为：

2*(purchase_water *amount_wateri+purchase_food *amount_foodi)<=fundi−1

在终点处，有对资金量的如下约束条件：

fund<=fund_now+water*price_water+food*price_food

加入天气后对规划的影响：

sumx=sumx+2*consumption_water_weather[weather[now+i]]

sumy=sumy+2*consumption_food_weather[weather[now+i]]

### 3.4 玩家行走约束

positioni,j<=∑k=(1~27)position_i−(1,k)×relation_postion-(k,j)

∑j=(1~27)distance_i−(1,j)×distance-(i,j)=1−boolean_walk_i

此外，如果天气是沙暴，则不能出行，此条件用程序里的 if 语句执行。

玩家挖矿约束：玩家在第 i 天挖矿的话，必须保证前一天和这一天都停留在同一个矿山，nj 为第 j 个矿山的地区编号：

∑j=1~aposition_i−1,n_j×position_i,nj>=Boolean_mine_i

### 3.5 负重约束

weight_water*amount_water+weight_food+amount_food<=capacity

矛盾约束：如果玩家当天行走，则不能挖矿和购买物资。该约束可用 if 语句判断。

此外，在容量有限的情况下，尽可能购买价格较贵但质量较小的东西，比如食物。因为村庄的资源价格比较高，所以尽可能的起点买食物，并且因为背包有上限，所以要着重看所拥有的资源的总质量。

### 3.6 目标函数

∑i(1~31)f(i,27)×

(fund_now_i+12×(purchase_water*amount_wateri+purchase_food*amount_food_i))

第一二问约束条件类似。

# 4 已知天气下的最优策略

## 4.1 模型求解概述

在前两关中，整个地形的所有天气状况已知，且行走次数有限（或者说玩家行动策略有限），因此可使用穷举法的方式遍历整个过程。但由于有时间和资源的约束，所以还是需要使用动态规划的方式，来更好地解决这个问题。本问题中游戏策略众多，单纯使用穷举法的计算量过大。我们最终决定使用动态规划的方法，将多阶段决策问题中的决策序列转化为若干子策略，来减少计算的复杂度。

遍历最后一阶段中所有处于终点位置的状态，总计花费最少（剩余资金最多）的策 略序列即为该初始状态下的最优策略。模型求解中，将考虑具体关卡的所有初始状态， 以得到该关卡的全局最优策略。

由于在起点处（人为定义为第 0 天）在不同的物资补给策略，这些策略会影响初始状态，进而影响游戏过程中的决策序列和最终结果。因此，使用多重搜索算法分别考虑第 0 天水和食物的补给，对第 0 天的所有补给策略进行搜索，每种补给策略对应一个初始状态。确定初始状态后，使用动态规划模型计算出每种初始状态所对应的最优策略。最后，从所有初始状态所对应的最优策略中选出最优策略，即为关卡的全局最优策略。但在有村庄与矿山的地图中，由于补给策略与时间、路线策略过多，增加了算法的复杂度，因此我们在制定最优策略时，先让程序运行，运行之后再二次记录，将路线与时间记录下来。每次经过村庄或到达终点时，若资源存在缺口，则检查缺口是否可在上一次经过村庄时得到满足。如果不满足，则进行剪枝处理，这也能够减小计算的复杂度。此外，由于求解均为单向进行，求解过程则可引入记忆化搜索，来改进这个计算程序。但由于程序中可较方便的使用 break，因此我们并未使用该策略。

此外未采用该剪枝策略的原因还要一个是容易发生组合爆炸。诸如此类约束条件复杂的优化模型最优解求解问题，一般采用剪枝优化的搜索算法对可能的最优解空间进行遍历得出最优解，但由于每天天气变化的影响，搜索过程中的剪枝优化效果不佳，时间复杂度极高，且容易发生组合爆炸。最终我们依然使用随机化贪心算法。

# 5 未知天气的最优策略

第三关和第四关的天气由于存在随机性，不能够给出一个确定性的最优策略，所以在设计方案之后需要对方案的结果进行评价。我们在本关使用了数学模型建立、实际情况抽象和仿真三个部分。其中，三者能得到一个比较普遍的结论。

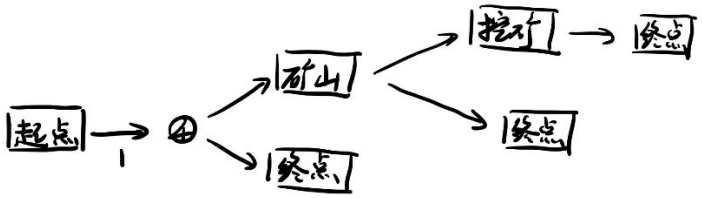
所以我们认为动态规划的思路也能用于未知天气的情况。

处理天气的思路一种情况是上面提到到按照一定分布随机生成，然后统计每一种情况分析出最有的策略。

也可以利用马尔可夫链，建立了天气预测而模型。根据第一关天气转换情况预测其余关卡的天气情况，在根据问题一的模型得出在不同天气情况下玩家的最佳策略，利用对应的天气情况概率和最佳策略下的最大资金，可以得出最终收益的数学期望，选择数学期望最大的一种策略。

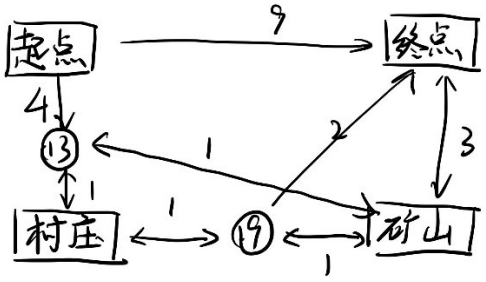
## 5.1 图论模型建立

与前两关类似，我们可以将网络进行预先简化为只有关键节点的模型。第三关较为简单，可直接画出图来：



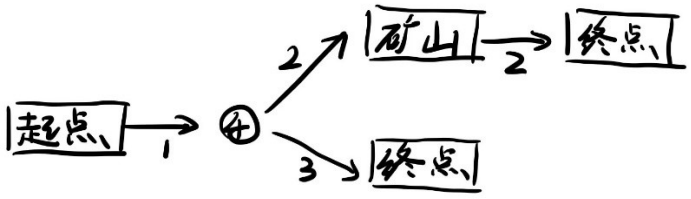

其中，以第四关为例（第四关的地图与第一关较为相似），为了实现最优策略，避免走路造成更多的消耗，途中必然会选择最短路径，途中的大部分位置是等价的。但根据上述的 Dijstra 算法可知，点 13、点 19 为地图关键点。在地图上看，玩家可以在此决定去向，无论是矿山还是村庄。从仿真结果看，这两个点是路径交会最多的一个点，说明了这点四通八达，给予了玩家更多的选择。因此，将关键节点点 13 与点 9 拿出来，有助于分析模型。

在地点去向上，部分点只能存在单向的关系。比如村庄不能去起点，。这些根据常识显而易见。



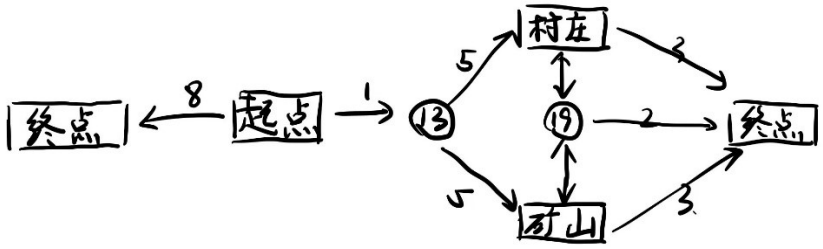

## 5.2 决策路径建立

第三关较为简单，通过对终点进行展开和分析，得到以下结构：




以第四关为例，玩家可以选择直接前往终点，终止游戏，也可以选择先前往矿山，然后工作或休息，当满足一定的条件，比如携带的水或食物不够时，触发终止条件，前往终点。基于该模型，我们逐步分析玩家的最优决策过程，是否应该“直接终止”还是前往矿山或村庄；在初始阶段、挖矿阶段、终止阶段，玩家应该依照天气采取怎样不同的策略；何种条件会触发结束挖矿阶段，开启终止阶段。

建立好简化版的网络模型后，我们可以建立玩家的一般决策路径，包括决策的各个阶段和各自的触发条件，然后再量化比较不同路径的效果，从而得出最优策略。在路线选择上，决策路径分为两个方向：第一是直接前往终点（可根据直觉判断，毕竟在地图上离得很近），二是绕行前往矿山补充资金，再经村庄，再到达终点。




### 5.2.1 直奔终点

第一种情况指挖矿产生亏损的情况；第二种情况的是指挖矿所带来的收益不如绕路所产生的资源消耗，因此该式指的是参与者挖矿所赚净收益不足以弥补其“绕路”产生的资源消耗所对应的金额；这两种情况下，参与者的决策应为：选择最短路径到达终点。、

因此，直奔终点适合那种天气特别恶劣的情况，因路中消耗急剧增加，因此尽快到达是最优解。

### 5.2.2 绕路至矿山与村庄

当绕路为更优解时，分为以下几种情况：第一是参与者在一定时间内挖矿所得净收益超过了他“绕路”前往村庄产生的资源消耗所对应的金额与他购买的在时间内挖矿所需的资源所对应的金额的总和；第二是参与者在村庄补给的资源重量没有超过他的负载上限。当这两个情况都满足时，绕路为更好的解决方案。

## 5.3 影响因素分析

### 5.3.1 位置类型影响

在没有村庄时，玩家的决策路径是由起点直接前往终点，或由起点直接前往矿山后再前往终点；在有村庄时，玩家决策路径会增加村庄与矿山之间的往返。如果没有村庄，那么玩家需要合理安排到矿山的时间以及初始购买的粮食，同时在保证在预期时间能够到达终点；如果有村庄，玩家需要合理安排在矿山和村庄之间的时间，保证能够在预定时间内到达村庄的情况下最大化收益。

### 5.3.2 天气影响

在非挖矿阶段，玩家玩家应该不管高温与否直接前往目的地；在挖矿阶段，应该对收益与成本进行量化分析。如果有沙尘暴天气，若玩家选择挖矿，应该提前前往终点；是否在沙尘暴天气挖矿取决于实际收益是否高于支出。

高温情况下，玩家的资源消耗量为原来的 2~3 倍。由于未来的天气情况是未知的，如果高温天气当天不走，未来可能仍然是高温，所以在非矿山的位置，玩家都应该不管与否高温直接前往目的地。在矿山位置所做的决策则应该取决于资源成本、天气、村庄及其距离等因素。此外，沙尘暴天气不能行走，但沙尘暴天气下能够挖矿。沙尘暴天气只能够留在原地或挖矿，所以沙尘暴主要会产生两个影响：需要提前前往终点以保证按时到达终点、需要根据成本、利润决策是否应该挖矿。

## 5.4 决策路径与阶段分析

在上文中，我们探讨玩家应该如何决策路径，以及不同阶段应该作出的决策，包括应该挖矿还是终止、什么天气应该挖矿、是否前往村庄、是否前往终点等，最终可以使得剩余的金钱最大化。

该问题的目标为，在初始资金给定的情况下，最大化最终的剩余的资金。我们定义可能出现的沙尘暴天数为、绕道挖矿与村庄的总距离、挖矿天数为、后续的总花费、挖矿收益、起点到终点距离等参数，可得到以下的约束条件，进而进行决策：

$$cost=cost-2*(cost\_food-food)*price\_food$$

$$weight\_now=weight\_now-(cost\_food-food)*weight\_food$$

$$weight=weight-cost\_food*weight\_food-cost\_water*weight\_water$$

使得 fund 得到最大化。

因此可得到如下几个结论：只有在绕道挖矿能产生正收益且收益能够弥补绕道的费用，挖矿才是划算的；以及只有在绕道前往村中购买资源，使得能够多产生的收益，能够弥补绕道前往村庄的费用和购买资源的费用，前往村庄才是划算的。在具体讨论中应该沿用该策略，通过比较潜在收益、剩余资源等因素，考虑最终应该采取的路径与策略。具体描述如下：

第一：如果前往矿山可能可以挖矿获得的收益，不足以弥补绕道的成本和无法挖矿时多消耗的水和粮食的费用，则不应该前往矿山。第二：有村庄的情况下是否应该挖矿取决于水和粮食的购买、

前往村庄和矿山的 综合成本，如果成本合算，参考剩余天数购买水和粮食。第三：如果携带的资源刚好在极端天气能够到达终点，且无法额外购买，则需要 即刻前往终点。第四：如果出现沙尘天气，需要根据距离与剩余时间，提前前往终点。

### 5.5 与已知天气情况下的区别

在已知天气的情况下，由于我们知道未来所要消耗的资源，因此我们可以控制所保有的资源到终点时正好消耗完。为了使到达终点的时候资金尽可能的多，我们不能买多余生存需要的水和食物，玩家在起点和村庄购买资源的价格高于终点返还的资金，因此在天气情况已知的情况下，玩家知道维持自己生存所需要的最少资源，所以有能力在到达终点的时候控制资源恰好耗尽。此策略仅仅适用于全局天气已知的情况。但三四关由于对未来天气与资源消耗的不确定性，所以我们不能保证所购买的资源恰好消耗干净。所以我们只能多购买一些资源，以应对未来的不确定性与风险。

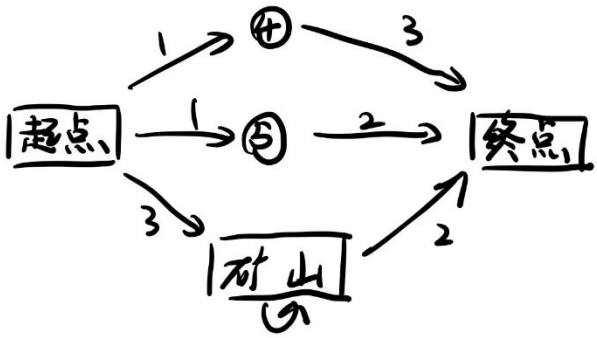
此外，因为村庄的资源价格比较高，所以尽可能的起点买食物，并且因为背包有上限，所以玩家倾向于在起点购买价格较贵且质量较小的资源。会尽可能的多买食物。这个是最重要的！

第三关天气随机，可通过统计的方法找规律。

第四关与前三关不同。第四关有了新的决策点。这个决策点 13 距离起点只有 4 天的路程，而且是玩家在沙漠中的必经点，这和第三关的情况非常相似。同时决策点距离村庄和矿山都只有一天的行程，在决策点的装屯点直接影响了玩家的决策，因此是玩家做决策的关键点。

## 6 动态博弈分析与风险评估

第五关的地图的决策路径选择如下图所示：（得到方法与上述方法一致）




我们先考虑只有一位玩家的情况。在本文预设的游戏中，一位完全控制资源与路径的玩家的目标都是在规定时间内到达终点，并挣取尽可能多的资金。一般的最优策略为前往矿山挖矿，该“无敌模式”玩家在前往矿山的路径与挖矿日期的选择上进行博弈。但第五关的前提条件特殊，由于挖矿收益较低，且天气情况已经给定，经过前文所建立的动态规划模型并求解，发现采取挖矿策略最终剩余金额与起点直接前往终点最终剩余金额相同。在如此的前提条件下，对于该玩家而言最优的策略不再是挖矿，而是直接以最近的路径返回终点。

随后我们引入博弈的情况。经计算，本问中玩家无论是直接前往终点还是挖矿，最优解背包容量都不会超过最大限制，所以玩家的目标仅仅关注如何让结束时的剩余金额最大化，即都希望采取直接从起点尽快前往终点的策略。但当游戏中 2 个玩家都采取这一策略，那么由于游戏规则的限制，他们各自的结果甚至不如单人挖矿的结果收益更高。

# 7 算法简述

## 7.1Depth-First-Search 算法

本程序使用 DFS（depth_first_search）算法，该算法可遍历整个地图里的各个元素。通过递归算法，使得计算出所有分支情况下的结果，最后寻找到最大的资金剩余量。

在天气情况与补给情况过多的前提下，DFS 算法的时间与空间的复杂度极大，因此，可通过先假定某一变量不变的前提下，对另外一组条件进行分析。因此将动态规划引入程序，大大减少了无用条件或不符合实际情况的遍历。必要时对程序的运行进行剪枝处理，以便及时止损。

此外，DFS 的遍历并没有对起点处进行处理。由于在起点处仅和补给策略有关，且大大影响决策结果，因此在起点出使用逆推的方式进行查找物资策略，即：先假定终点处所剩余物资数为 0，通过 DFS 逆推法，计算出起点处的物资携带量。

其中，第一关与第二关的代码结构与思路几乎相同。

其中排序的方式为冒泡排序法。（注：至于为什么不用循环标定法来记录最大值，是因为冒泡排序能够得到规划的优劣顺序，便于后期总结与抽象提取规划模型。）

## 7.2Dijkstra 算法

Dijkstra 算法解决了图 G=<V,E>上带权的单源最短路径问题。戴克斯特拉算法使用类似广度优先搜索的方法解决赋权图的单源最短路径问题。该算法常用于路由算法或者作为其他图算法的一个子模块。一般在应用中，被认为是最良优先搜索的一个特例。

在本次研究中，该算法与图论相结合使用，在计算机编程方面使用三层嵌套循环来实现判断，且使用递归的方法来使用。因最短路路径的本质就是比较在两个顶点之间中转点，比较经过与不经过中转点的距离哪个更短，所以本算法正好契合本题目。

但由于该算法的时间复杂度(V^3)与空间复杂度(V^2)极高，因此在做题目的时候，采用了去支处理，即：通过逐个判断 position 与 dayss 参数的量值，及时停止掉不必要的路径与运算。

## 7.3 蒙特卡洛方法

蒙特卡洛方法是一种基于随机数和统计抽样，以便近似求解数学物理问题的方法， 这种方法又称统计实验法，或者计算机随机模拟方法，由冯诺依曼命名；在这里我们也可用蒙特卡洛方法，对该问题中的模型进行仿真，以便检验该决策模型的合理性。

该方法属于离散数学范畴。在本题中，与马尔可夫决策法相关，来进行对风险与获益之间的权衡。在风险与获益中得到平衡，使得在风险可控的情况下，得到最终的决策方案。

## 7.4 马尔可夫决策

马可夫决策过程是离散时间随机控制过程，是马尔可夫链的推广。它提供了一个数学框架，用于在结果部分随机且部分受决策者控制的情况下对决策建模。 MDP 对于研究通过动态规划解决的最佳化问题很有用。

在本题中，该决策方法与动态规划相结合。显式模型通过从分布中采样简单地生成生成模型，并且生成模型的重复应用生成轨迹模拟器。在相反的方向上，只能通过回归分析研究近似模型。所以只能与动态规划的回归分析来做拟合。评价相关的风险与决策方式的必要性与可行性，可使题目中的玩家对选择路线的风险与获益进行评估与判断。

## 7.5Floyd-Warshall 算法

Floyd-Warshall 算法，中文亦称弗洛伊德算法，是解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权（但不可存在负权回路）的最短路径问题，同时也被用于计算有向图的传递闭包。

这个算法与图论紧密结合。

与 DFS、BFS 算法不同，我们的 DFS 用来求一个点到一个点之间的最短路径是可行的，但多个点的话，DFS 算法就有些捉襟见肘了。如果我们使用 n 次 DFS 算法的话，那时间、空间复杂度会非常的高，且在数学表达与梳理上非常复杂不清晰。此外，我们不可能记录每个点到任意一个点的最短路径，所以我们引入这个算法了。

在本题中，我们求得路径与时间规划的最优解，因此我们就可以把起点和终点看作最最关键的点，把村庄、矿山等作为最关键点，把 Dijkstra 算法得出的点设为关键点，其余为中转 k 个点，来求得最优策略。

但是如果我们采用每次询问都进行一次 Floyd 算法的话查找两个点的最短路径，显然复杂度太高。

我们注意到图论中点的对称性，点的集合不为空（事实也如此），但边的集合可以为空，并且边的多样性是会影响到我们的选择的，至关重要。假设一共有 a 个点，经对称优化后有 k 个“优化点”，因此这个时间我们便可以设置为 k 的值，即在 k 时间内中转，不能超过 k 时间，因此我们就可以每次询问使用一次 loyd 算法了，但是我们的 k 是固定的，我们只需要两重循环就好了。

# 8 决策结果

通过附录程序的计算之后，最终结果详见附件一与附件二。

第一关最优策略为：出发（第 0 天）带 178 箱水与 333 箱食物；第 8 天经过村庄，补给 163 箱水；第 10 天到达矿山，第 11 至第 19 天停留，第 11 天与第 17 天不挖矿，其余时间挖矿；第 21 天经过村庄补给 36 箱水与 16 箱食物；第 24 天到达终点。最终剩余资金为 10470 元。

第二关最优策略为：出发（第 0 天）带 130 箱水与 405 箱食物；第 10 天经过村庄，补给 189 箱水；第 11 天停留于该村庄；第 12 天到达矿山，第 13 天至第 18 天挖矿；第 19 天经过村庄补给 196 箱水与 186 箱食物；第 30 天到达终点。最终剩余资金为 12730 元。

第三关与第四关由于天气未知，因此我们使用随机仿真的方式进行模拟决策。在最激进的策略中，我们得到的结论如下：在三关中，187 箱水与 210 箱食物；第四关中，带 193 箱水和 70 箱食物。与第一关、第二关的情况类似。因此我们认为模型较为合理。

第五关由于多人进行游戏，通过面向对象与继承类的方式对单人游戏场景进行封装，进而仿真出相关的情形。由于约束条件太少，目前我们仅验证了多人游戏能够跑通的可行性。但整体来说，确实存在一最优策略，使得多人游戏博弈目标最大化。

# 9 模型推广与评价

我认为最重要的推广与评价是在第三问中。

在问题三中，由于游戏玩法由单人变为多人，人的社会性也逐渐显现。在本论文的分析中，均按照人是“理性的”为假设与基础进行的分析与讨论。但在现实中，由于社会心理学的因素，人往往是不理性的，甚至是莽撞的。在现实中，玩家之间可能会产生合作，共同追求利益的最大化；也可能产生报复心理，在玩家之间产生资源抢占与内耗的现象。因此我认为，在真实世界中，第五关与第六关的情况更加复杂、多样与随机。那么如何能在能成功的情况下，风险尽可能小、资源分配尽可能合理、利益尽可能最大化，是目前社会学上需要考虑的问题，而不仅仅是数学模型与算法能够解决的问题了。

此外，在第三问中，基于静态博弈与动态博弈的思路，在多玩家参与且决策结果会互相影响的情况下，基于前两问的部分结论，建立了玩家的行为策略的基本模型。我们的模型是静态完全信息博弈，即以预先假定玩家之间不存在任何信息交换。那么如果可以信息交换，那么会是什么样的结论呢？我认为可以类比计划经济与市场经济。如果让“可以信息交换”类比“计划经济”，让“完全独立运行”类比“市场经济”，那么我们可以得到：在生产力低下、资源有限匮乏不足的情况下，还是比较适合“独立运行，信息不交换”的策略；而当资源比较充足的情况下，应当使用宏观调控，进行资源上的协调。（在经济运行模式上，采用西方经济学的理论。）

在第一问与第二问的推广中，我们可以分别在数学上得到下面几点：

基于上述的模型和算法，可以直接求解出第一、二问的全局最优策略。模型的适应性强、

稳定性高，模型可以推广到任意一种全程天气状况已知的地图情形。将原始地图简化为有向图模型后，通过分析游戏规则中的决策方案和约束条件，建立了完整的动态规划模型，包括状态变量、决策变量、状态转移函数等。因为任何情况已知，所以可以得到确切的最优解。非常方便简单。

在环境不确定的情况下，该模型通过分析不确定因素的累计情况，并以此为基础建立了玩家的决策路径。同时，也建立了玩家的收益与各个行为之间的关联，从而建立了数个基础的决策依据，具体环境下，只需进行成本分析和环境分析，即可获得玩家的一般最佳策略。这时，由于天气的不确定性，存在了风险评估与判断，这更加符合实际情况。在天气的假设中，也有两种技术路线，比如假定整个地图天气相同或者每个区域都有不同的天气，在这里根据不同的假设也会得到不相同的结论（但总体上大趋势一样）。由于关键结点之前玩家碰到的环境不一致，在关键结点玩家需要根据具体的情况做出决策，同时计算出玩家在不同情况下的收益，考虑环境的不确定性，即可分析出最佳策略。

## 论文致谢

本论文是在该课程讲师的指导下完成的。本文中所用到的线性规划、运筹学等知识均为数学模型课程上得到学习。老师严谨的治学态度和渊博的学识，朴实无华及平易近人的人格魅力对我影响深远！此外，老师对于算法的理解与讲授，使我受益匪浅。特别是本文用到的 DFS 算法，使小组成员对于算法与本项目的理解更加深入与透彻。感谢学校与老师给予我们小组数学建模的机会，使得我们能够提升自己数学建模与抽象的能力。在此，谨向我的大学与老师致以诚挚的感谢与敬意！

## 参考文献

[ 1 ] Robert Johansson *Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib* Apress ISBN: 978-1484242452

[ 2 ] 姜启源 谢金星 叶俊 《数学模型（第五版）》 高等教育出版社 ISBN: 9787040492224

[ 3 ] 黄万艮 《多目标路的扩展 Dijstra 算法》 2004 Doi57.1624/374

[ 4 ] 叶强 《强化学习：马尔科夫决策过程》 https://zhuanlan.zhihu.com/p/28084942 2023.5.6

[ 5 ] Wikipedia-zh-community 《图论》https://zh.wikipedia.org/wiki/%E5%9B%BE%E8%AE%BA 2023.5.6

[ 6 ] Tang, Z.G., Wu, X. and Zhang, Y. (2020) Towards a Better Understanding of Randomized Greedy Matching. Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, 1097-1110.

附件一：

第一关

| 日期 | 所在区域 | 剩余资金数 | 剩余水量 | 剩余食物量 |
|---|---|---|---|---|
| 0 | 1 | 5780 | 178 | 333 |
| 1 | 25 | 5780 | 162 | 321 |
| 2 | 24 | 5780 | 146 | 309 |
| 3 | 23 | 5780 | 136 | 295 |
| 4 | 23 | 5780 | 126 | 285 |
| 5 | 22 | 5780 | 116 | 271 |
| 6 | 9 | 5780 | 100 | 259 |
| 7 | 9 | 5780 | 90 | 249 |
| 8 | 15 | 4150 | 243 | 235 |
| 9 | 13 | 4150 | 227 | 223 |
| 10 | 12 | 4150 | 211 | 211 |
| 11 | 12 | 4150 | 201 | 201 |
| 12 | 12 | 5150 | 177 | 183 |
| 13 | 12 | 6150 | 162 | 162 |
| 14 | 12 | 7150 | 138 | 144 |
| 15 | 12 | 8150 | 114 | 126 |
| 16 | 12 | 9150 | 90 | 108 |
| 17 | 12 | 9150 | 80 | 98 |
| 18 | 12 | 10150 | 50 | 68 |
| 19 | 12 | 11150 | 26 | 50 |
| 20 | 13 | 11150 | 10 | 38 |
| 21 | 15 | 10470 | 36 | 40 |
| 22 | 9 | 10470 | 26 | 26 |
| 23 | 21 | 10470 | 10 | 14 |
| 24 | 27 | 10470 | 0 | 0 |
| 25 | | | | |
| 26 | | | | |
| 27 | | | | |
| 28 | | | | |
| 29 | | | | |
| 30 | | | | |

| 第二关 | | | | |
|---|---|---|---|---|
| 日期 | 所在区域 | 剩余资金数 | 剩余水量 | 剩余食物量 |
| 0 | 1 | 5300 | 130 | 405 |
| 1 | 2 | 5300 | 114 | 393 |
| 2 | 3 | 5300 | 98 | 381 |
| 3 | 4 | 5300 | 88 | 367 |
| 4 | 4 | 5300 | 78 | 357 |
| 5 | 5 | 5300 | 68 | 343 |
| 6 | 13 | 5300 | 52 | 331 |
| 7 | 13 | 5300 | 42 | 321 |
| 8 | 22 | 5300 | 32 | 307 |
| 9 | 30 | 5300 | 16 | 295 |
| 10 | 39 | 5200 | 10 | 283 |
| 11 | 39 | 3410 | 179 | 273 |
| 12 | 30 | 4410 | 163 | 261 |
| 13 | 30 | 5410 | 148 | 240 |
| 14 | 30 | 6410 | 124 | 222 |
| 15 | 30 | 7410 | 110 | 204 |
| 16 | 30 | 8410 | 86 | 186 |
| 17 | 30 | 9410 | 56 | 156 |
| 18 | 30 | 9410 | 26 | 126 |
| 19 | 39 | 5730 | 196 | 200 |
| 20 | 46 | 5730 | 180 | 188 |
| 21 | 55 | 5730 | 170 | 174 |
| 22 | 55 | 6730 | 155 | 153 |
| 23 | 55 | 7730 | 131 | 135 |
| 24 | 55 | 8730 | 116 | 114 |
| 25 | 55 | 9730 | 86 | 84 |
| 26 | 55 | 10730 | 62 | 66 |
| 27 | 55 | 11730 | 47 | 45 |
| 28 | 55 | 12730 | 32 | 24 |
| 29 | 56 | 12730 | 16 | 12 |
| 30 | 64 | 12730 | 0 | 0 |

附件二：

```
import numpy as np

#marked_point 0-> start；  1-> village；2 -> mine; 3-> end
position = np.array((0,1,2,3))
# The distance between the marked position
distance=np.array(((0, 6, 8, 3),
                   (6, 0, 2, 3),
                   (8, 2, 0, 5),
                   (3, 3, 5, 0)))
# The relation between the marked position is found by graph theory by hand
relation_position = np.array(((0, 1, 1, 1),
                              (0, 0, 1, 1),
                              (0, 1, 0, 1),
                              (0, 0, 0, 0)))
#The weather in module 1 in 30 days
weather = np.array((2, 2, 1, 3, 1,
                    2, 3, 1, 2, 2,
                    3, 2, 1, 2, 2,
                    2, 3, 3, 2, 2,
                    1, 1, 2, 1, 3,
                    2, 1, 1, 2, 2))
#Weight&Price
weight_water=3
weight_food=2
price_water=5
price_food=10
#Consumption in different weather
consumption_water_weather=np.array((0, 5, 8, 10))
consumption_food_weather=np.array((0, 7, 6, 10))
#The number of terrain's type
n=4
#Original argument
capacity=1200
fund_basic=10000
income=1000
date=30
consumption_water=np.zeros((32,4,4))
consumption_food=np.zeros((32,4,4))
# How long does it take from i to j on day k
dayss = np.zeros((32, 4, 4))
```

```
fund=0
#Record the everyday information
activity_everyday=np.zeros(32)
position_everyday=np.zeros(32)
optimum_position = np.zeros(32)
optimum_activity = np.zeros(32)
optimum_origin_water=0
optimum_origin_food=0

#Use the algorithm of DFS. I learned it on GitHub!
def depth_first_search(days:int, now:int,weight:int, fund_now:int,water:int, food:int, type:int)->None:
    activity_everyday[days]=type
    position_everyday[days]=now
    global fund, optimum_origin_food, original_water

    #village
    if position[now]==1:
        weight=capacity-water*weight_water-food*weight_food

    for i in range(n):
        if relation_position[position[now]][position[i]]:
            cost_food=consumption_food[days][now][i]
            cost_water=consumption_water[days][now][i]
            cost=fund_now
            weight_now=weight
            if food>=cost_food:
                food_now=food-cost_food
            else:
                food_now=0
                cost=cost-2*(cost_food-food)*price_food
                weight_now=weight_now-(cost_food-food)*weight_food
            if water>=cost_water:
                water_now=water-cost_water
            else:
                water_now=0
                cost=cost-2*(cost_water-water)*price_water
                weight_now=weight_now-(cost_water-water)*weight_water

            if weight_now<0 or cost<0:
                continue
            depth_first_search(days+dayss[days][now][i],i,weight_now,cost,water,food,0)

        #mine
    if position[now]==2:
```

```
            attday=days
            cost_food=consumption_water_weather[weather[attday]]
            cost_water=consumption_food_weather[weather[attday]]
            attday+=1
            if water>=cost_water:
                x=x-cost_water
                cost_water=0
            else:
                cost_water=cost_water-water
                water=0
            if food>=cost_food:
                food=food-cost_food
                cost_food=0
            else:
                cost_food=cost_food-food
                food=0
            weight=weight-cost_food*weight_food-cost_water*weight_water
            fund_now=fund_now-2*(cost_food*price_food+cost_water*price_water)
            if weight>=0 and fund>=0:
                depth_first_search(attday,now,weight,fund_now,water,food,1)

            attday=days
            cost_water=consumption_water_weather[weather[attday]]*2
            cost_food=consumption_food_weather[weather[attday]]*2
            attday+=1
            if water>=cost_water:
                x=x-cost_water
                cost_water=0
            else:
                cost_water=cost_water-water
                water=0
            if food>=cost_food:
                food=food-cost_food
                cost_food=0
            else:
                cost_food=cost_food-food
                food=0
            weight=weight-cost_food*weight_food-cost_water*weight_water
            fund_now=fund_now-2*(cost_food*price_food+cost_water*price_water)
            fund_now=fund_now+income
            if weight>=0 and fund>=0:
                depth_first_search(attday,now,weight,fund_now,water,food,2)

    if position[now]==3:
```

```
            if fund<=fund_now+water*price_water+food*price_food:
                optimum_origin_water=water
                optimum_origin_food=food
                fund=fund_now+water*price_water+water*price_food
                for i in range(date+1):
                    optimum_position[i]=position_everyday[i]
                    optimum_activity[i]=activity_everyday[i]
                activity_everyday[days]=-1
                position_everyday[days]=-1
                return

        if days>=date:
            activity_everyday[days]=-1
            position_everyday[days]=-1
            return

        activity_everyday[days]=-1
        position_everyday[days]=-1

#Magic function!
if __name__ =='__main__':
        for i in range(date+1):
            activity_everyday[i]=-1
            position_everyday[i]=-1

        for i in range(date+1):
            for j in range(n):
                for k in range(n):
                    if relation_position[position[j]][position[k]]:
                        now, count, sumx, sumy = 0, 0, 0, 0
                        while count<distance[j][k]:
                            if weather[now+i]!=3:
                                count+=1
                                sumx=sumx+2*consumption_water_weather[weather[now+i]]
                                sumy=sumy+2*consumption_food_weather[weather[now+i]]
                            else:
                                sumx=sumx+consumption_water_weather[weather[now+i]]
                                sumy=sumy+consumption_food_weather[weather[now+i]]

                            now+=1
                            if now+i >=date:
                                break
                        if count <distance[j][k]:
                            sumx=xumy=20000
```

```
                        now=30
                    consumption_water[i][j][k]=sumx
                    consumption_food[i][j][k]=sumy
                    dayss[i][j][k]=now
        print(type(dayss[0,0,0]))

        dictionary={}
        for i in range(capacity+1):
            c2=i//weight_food
            c1=(capacity-i)//weight_water
            dictionary.setdefault((c1,c2),0)
            if not dictionary[(c1,c2)]:
                print((c1,c2))
                depth_first_search(0,0,0,fund_basic-c2*consumption_food-
c1*consumption_water,c1,c2,-1)
            dictionary[(c1,c2)]=1
    print(fund)
```

```
import numpy as np

#marked_point 0-> start；  1-> village；2 -> mine; 3-> end
position = np.array((0,2,1,2,1,3))
# The distance between the marked position
distance=np.array(((0, 7, 8, 9, 9, 11),
                   (7, 0, 1, 3, 4, 4),
                   (8, 1, 0, 2, 3, 3),
                   (9, 3, 2, 0, 1, 2),
                   (9, 4, 3, 1, 0, 2),
                   (11, 4, 3, 2, 2, 0)))
# The relation between the marked position is found by graph theory by hand
relation_position = np.array(((0, 1, 1, 1),
                              (0, 0, 1, 1),
                              (0, 1, 0, 1),
                              (0, 0, 0, 0)))
#The weather in module 1 in 30 days
weather = np.array((2, 2, 1, 3, 1,
                    2, 3, 1, 2, 2,
                    3, 2, 1, 2, 2,
                    2, 3, 3, 2, 2,
                    1, 1, 2, 1, 3,
                    2, 1, 1, 2, 2))
#Weight&Price
weight_water=3
```

```
weight_food=2
price_water=5
price_food=10
#Consumption in different weather
consumption_water_weather=np.array((0, 5, 8, 10))
consumption_food_weather=np.array((0, 7, 6, 10))
#The number of terrain's type
n=6
#Original argument
capacity=1200
fund_basic=10000
income=1000
date=30
consumption_water=np.zeros((32,6,6))
consumption_food=np.zeros((32,6,6))
# How long does it take from i to j on day k
dayss = np.zeros((32, 6, 6))
fund=0
#Record the everyday information
activity_everyday=np.zeros(32)
position_everyday=np.zeros(32)
optimum_position = np.zeros(32)
optimum_activity = np.zeros(32)
optimum_origin_water=0
optimum_origin_food=0

#Use the algorithm of DFS. I learned it on GitHub!
def depth_first_search(days:int, now:int,weight:int, fund_now:int,water:int, food:int, type:int)->None:
    activity_everyday[days]=type
    position_everyday[days]=now
    global fund, optimum_origin_food, original_water

    #village
    if position[now]==1:
        weight=capacity-water*weight_water-food*weight_food

    for i in range(n):
        if relation_position[position[now]][position[i]]:
            cost_food=consumption_food[days][now][i]
            cost_water=consumption_water[days][now][i]
            cost=fund_now
            weight_now=weight
            if food>=cost_food:
                food_now=food-cost_food
```

```
                else:
                    food_now=0
                    cost=cost-2*(cost_food-food)*price_food
                    weight_now=weight_now-(cost_food-food)*weight_food
                if water>=cost_water:
                    water_now=water-cost_water
                else:
                    water_now=0
                    cost=cost-2*(cost_water-water)*price_water
                    weight_now=weight_now-(cost_water-water)*weight_water

                if weight_now<0 or cost<0:
                    continue
                depth_first_search(days+dayss[days][now][i],i,weight_now,cost,water,food,0)

        #mine
    if position[now]==2:
        attday=days
        cost_food=consumption_water_weather[weather[attday]]
        cost_water=consumption_food_weather[weather[attday]]
        attday+=1
        if water>=cost_water:
            x=x-cost_water
            cost_water=0
        else:
            cost_water=cost_water-water
            water=0
        if food>=cost_food:
            food=food-cost_food
            cost_food=0
        else:
            cost_food=cost_food-food
            food=0
        weight=weight-cost_food*weight_food-cost_water*weight_water
        fund_now=fund_now-2*(cost_food*price_food+cost_water*price_water)
        if weight>=0 and fund>=0:
            depth_first_search(attday,now,weight,fund_now,water,food,1)

        attday=days
        cost_water=consumption_water_weather[weather[attday]]*2
        cost_food=consumption_food_weather[weather[attday]]*2
        attday+=1
        if water>=cost_water:
            x=x-cost_water
```

```
                cost_water=0
            else:
                cost_water=cost_water-water
                water=0
            if food>=cost_food:
                food=food-cost_food
                cost_food=0
            else:
                cost_food=cost_food-food
                food=0
            weight=weight-cost_food*weight_food-cost_water*weight_water
            fund_now=fund_now-2*(cost_food*price_food+cost_water*price_water)
            fund_now=fund_now+income
            if weight>=0 and fund>=0:
                depth_first_search(attday,now,weight,fund_now,water,food,2)

    if position[now]==3:
        if fund<=fund_now+water*price_water+food*price_food:
            optimum_origin_water=water
            optimum_origin_food=food
            fund=fund_now+water*price_water+water*price_food
            for i in range(date+1):
                optimum_position[i]=position_everyday[i]
                optimum_activity[i]=activity_everyday[i]
            activity_everyday[days]=-1
            position_everyday[days]=-1
            return

    if days>=date:
        activity_everyday[days]=-1
        position_everyday[days]=-1
        return


    activity_everyday[days]=-1
    position_everyday[days]=-1

#Magic function!
if __name__ =='__main__':
    for i in range(date+1):
        activity_everyday[i]=-1
        position_everyday[i]=-1

    for i in range(date+1):
```

```
        for j in range(n):
            for k in range(n):
                if relation_position[position[j]][position[k]]:
                    now, count, sumx, sumy = 0, 0, 0, 0
                    while count<distance[j][k]:
                        if weather[now+i]!=3:
                            count+=1
                            sumx=sumx+2*consumption_water_weather[weather[now+i]]
                            sumy=sumy+2*consumption_food_weather[weather[now+i]]
                        else:
                            sumx=sumx+consumption_water_weather[weather[now+i]]
                            sumy=sumy+consumption_food_weather[weather[now+i]]

                        now+=1
                        if now+i >=date:
                            break
                    if count <distance[j][k]:
                        sumx=xumy=20000
                        now=30
                    consumption_water[i][j][k]=sumx
                    consumption_food[i][j][k]=sumy
                    dayss[i][j][k]=now
        print(type(dayss[0,0,0]))

        dictionary={}
        for i in range(capacity+1):
            c2=i//weight_food
            c1=(capacity-i)//weight_water
            dictionary.setdefault((c1,c2),0)
            if not dictionary[(c1,c2)]:
                print((c1,c2))
                depth_first_search(0,0,0,fund_basic-c2*consumption_food-
c1*consumption_water,c1,c2,-1)
            dictionary[(c1,c2)]=1
    print(fund)
```