期末大作业

小组成员：

队长：尤冠杰-20053068-智能科学学院

队员1：孙浏鑫-20053070-智能科学学院

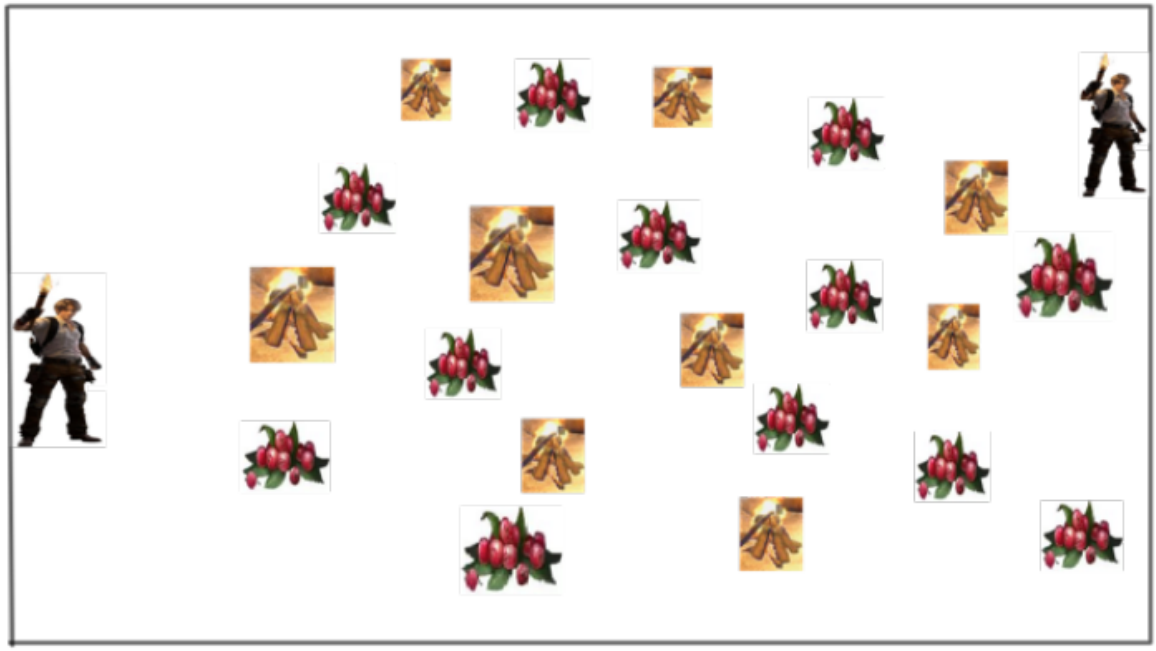
队员2：邓慧明-20020022-计算机学院

1 问题描述

《明日之后》是网易开发的一款生存类游戏，讲述这样一个游戏情节：

病毒肆虐各国，人类文明险些毁灭，为了能够在末世中“活下去”，志同道合的伙伴集结起来，一起在病毒蔓延、感染者遍地、资源有限、天气严酷的世界中求生。在游戏中，人物有两类重要的行为，一类是要通过寻找各种食物来维持自身生存，另一类是通过各种御寒措施来降低自身的寒冷程度从而提高生存能力。我们把这两类行为对应到人物的两个特征，前者称为饱食度，刻画人物饥饿的状态，饱食度为负表示处于饥饿状态，饱食度越小，人物生存越难；后者称为舒适度，刻画人物寒冷的状态，舒适度为负表示处于寒冷状态，舒适度越小，游戏人物生存越难。

以该游戏为背景，我们设计这样一个简化的场景。假设游戏人物活动区域为一个空间区域，空间区域中不同位置分布有一些食物和篝火，人物从该区域某一个位置进入，从区域另一个位置出去，如下图所示：



游戏人物在该区域行走的规则如下：

（1）人物到达食物点吃到食物，其饱食度将提高，提高程度依赖于食物数量。

（2）人物到达篝火位置，其舒适度将提高，提高程度依赖于篝火的大小。

（3）人物在平路（即z坐标相同）行走100米，饱食度和舒适度均降低5个单位，若走上坡路（z坐标增加），饱食度和舒适度每走100米均降低6个单位，若走下坡路（z坐标减少），饱食度和舒适度每走100米均降低4个单位。假设上、下坡已经等效为两点之间直线行走。

（4）当人物到达食物点或篝火点，若饱食度和舒适度中任意一个小于-5，人物将死亡，无法通过食物或篝火提高饱食度或舒适度。

（5）假设人物在开始位置时的饱食度和舒适度均为10。

（6）要求人物到达终点时，饱食度和舒适度均不小于-3。

附件中给出了食物点和篝火点的信息，第一列为点的编号，第2-4列为食物点或篝火点的位置坐标，第一行为起点信息，最后一行为终点信息；第5列为点的类型，1表示该点为食物点，0表示该点为篝火点，第6列为人物位于该点时可通过补充食物或利用篝火提高其饱食度或舒适度的大小，间接代表了该处食物的数量或篝火的大小。

请建立数学模型和算法解决以下问题：问题1：规划该人物从起点到终点的路线（用序号表示），使其经过的食物点和篝火点的次数最少。问题2：在第一问的基础上，进一步考虑人物行走的路径尽可能短，规划其从起点到达终点的路线（用序号表示）。问题3：在篝火点，游戏人物可以制作火把携带，制作火把将使得饱食度降低0.5个单位，但携带的火把能支持人物行走20米而不降低舒适度。请在第一问和第二问的规划路线基础上，进一步考虑人物可制作火把携带的方案，使得人物到达终点后的饱食度和舒适度尽可能高。

问题1：规划该人物从起点到终点的路线（用序号表示），使其经过的食物点和篝火点的次数最少。

问题2：在第一问的基础上，进一步考虑人物行走的路径尽可能短，规划其从起点到达终点的路线（用序号表示）。

问题3：在篝火点，游戏人物可以制作火把携带，制作火把将使得饱食度降低0.5个单位，但携带的火把能支持人物行走20米而不降低舒适度。请在第一问和第二问的规划路线基础上，进一步考虑人物可制作火把携带的方案，使得人物到达终点后的饱食度和舒适度尽可能高。

2 建模过程

2.1 准备工作

2.1.1 数据可视化

第一个工作是要读取数据并在三维空间中绘制出来。

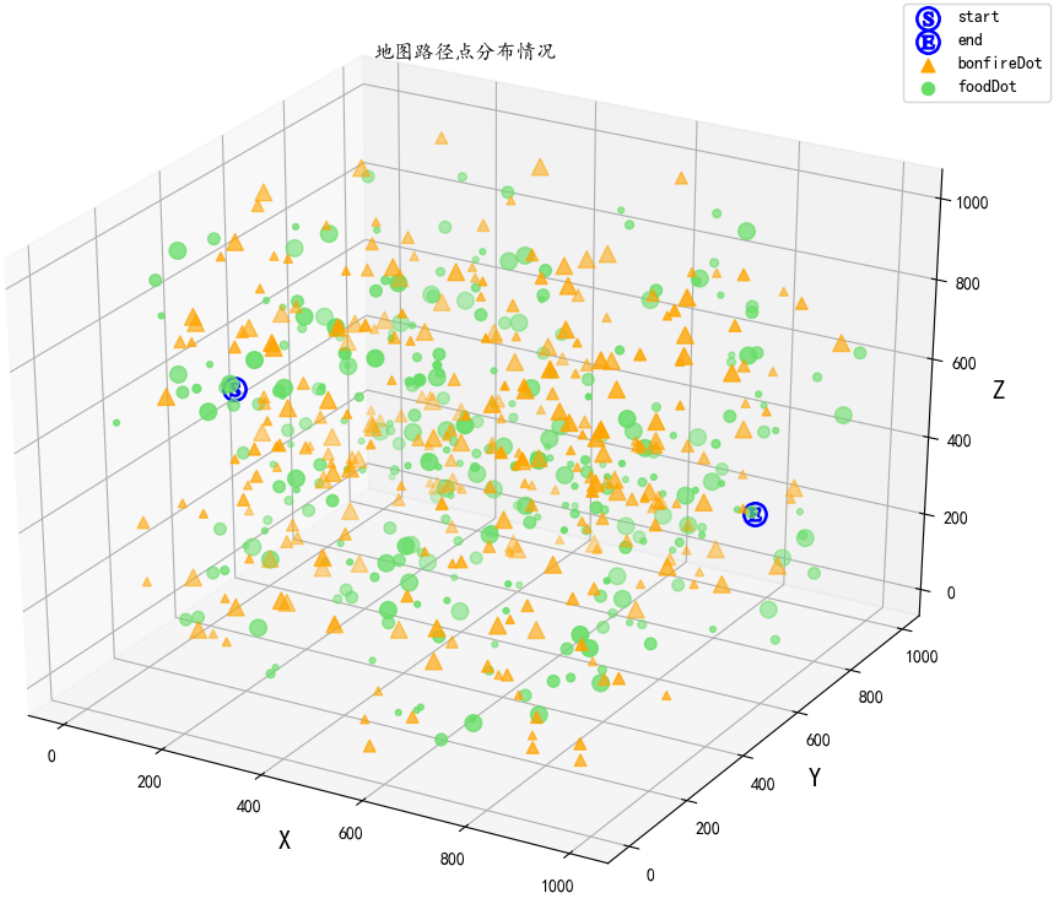
需要用到python的绘图库(matplotlib和Axes3D)，示例代码如下：

(参考python绘制3D散点图)

```
#!usr/bin/env python
# -*- coding: utf-8 -*-
# @author:ygj
# @file: 01-dataVisuliation.py
# @time: 2021/04/23


import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

if __name__ == "__main__":
    # Load data
    dots = np.loadtxt('./data.csv', delimiter=',')
    routeDots = dots[1:-1, :]
    bonfireIndex = [i for i, x in enumerate(routeDots[:, 4].tolist()) if x == 0]
    foodIndex = [i for i, x in enumerate(routeDots[:, 4].tolist()) if x == 1]

    startDot = dots[0, :]  # 起点
    endDot = dots[-1, :]  # 终点
    bonfireDots = routeDots[bonfireIndex, :]  # 篝火点
    foodDots = routeDots[foodIndex, :]  # 食物点

    '''
    可视化
    '''
    #解决中文显示问题
    plt.rcParams['font.sans-serif'] = ['KaiTi']
    plt.rcParams['axes.unicode_minus'] = False

    x1, y1, z1 = bonfireDots[:, 1], bonfireDots[:, 2], bonfireDots[:, 3]
    x2, y2, z2 = foodDots[:, 1], foodDots[:, 2], foodDots[:, 3]

    fig = plt.figure()
    ax = Axes3D(fig)
    # 图例设置
    startMarker = '$\circledS$'
    endMarker = '$\circledE$'

    # 散点绘制
```

```
    ax.scatter(startDot[1], startDot[2], startDot[3], c='b', s=200,
marker=startMarker, label='start')
    ax.scatter(endDot[1], endDot[2], endDot[3], c='b', s=200, marker=endMarker,
label='end')
    ax.scatter(x1, y1, z1, c='#FFA500', s=np.exp2(bonfireDots[:, 5]) * 6,
marker='^', label='bonfireDot')
    ax.scatter(x2, y2, z2, c='#68DE69', s=np.exp2(foodDots[:, 5]) * 3,
label='foodDot')

    # 添加坐标轴(顺序是Z, Y, X)
    ax.set_zlabel('Z', fontdict={'size': 15, 'color': 'black'})
    ax.set_ylabel('Y', fontdict={'size': 15, 'color': 'black'})
    ax.set_xlabel('X', fontdict={'size': 15, 'color': 'black'})
    # 添加图例
    ax.legend(loc='best')

    plt.title('地图路径点分布情况')
    plt.show()
    # fig.savefig('scatter.svg', dpi=600, format='eps')
```

可以查看到路径点数据分布如下：



蓝色点代表起点和终点，**黄色点**是篝火点，**绿色点**是食物点，**点大小**代表补给量的多少。

可以观察到：

1. 路径点分布较为均匀
2. 边缘的路径点较为稀疏
3. 没有个别点补给过大的情况

2.1.2 游戏框架搭建

为了更好的实现搜索策略，以及在改进算法过程中便于迭代代码，我们选择将路径点数据抽象为**路径点类**，并构建了一个**玩家类**，用来模拟玩家人物在路径点之间移动的行为。

路径点类：

主要包含路径点的序列、位置、补给量和补给类型。

实现代码如下：

```
class Dot:
    def __init__(self, index, position, food, supply):
        """
        路径点类
        :param index: 序号
        :param position: 位置
        :param isFood: 是否为食物
        :param isBonfire: 是否为篝火
        :param isArrived: 是否到达过
        :param supply: 补给数
        :param x, y, z: 坐标
        """
        self.index = int(index)
        self.position = position
        self.x, self.y, self.z = self.position
        self.isFood = bool(food)
        self.isBonfire = not bool(food)
        self.supply = supply

    def __eq__(self, other):
        """
        =运算符重载
        判断两点是否相同
        """
        return (self.index == other.index) and (self.x == other.x) and (self.y
== other.y) and (self.z == other.z)

    def info(self):
        """
        查看当前路径点信息
        """
        if self.isFood:
            print('Dot', self.index, ' Position: ', self.position, 'Food Supply:
', self.supply)
        elif self.isBonfire:
            print('Dot', self.index, ' Position: ', self.position, 'BonFire
Supply: ', self.supply)
```

玩家类：

主要包含玩家起点、终点、初始饱食度、初始舒适度、当前位置和路径。

同时包含了根据路径更新当前状态、移动到目标点、判断目标点是否能够到达、计算到目标点的消耗值、获得当前状态等函数。

实现代码如下：

```python
class Player:
    def __init__(self, startDot, endDot, satiety, comfort):
        """
        玩家类
        :param startDot:起点
        :param endDot:终点
        :param satiety:饱食度
        :param comfort:舒适度
        """
        self.route = [startDot]
        self.position = startDot.position
        self.x, self.y, self.z = self.position
        self.startDot = startDot
        self.endDot = endDot
        self.satiety, self.comfort = satiety, comfort
        self.initSC = [satiety, comfort]

    def update(self, route_set):
        '''
        更新路径信息
        :param route_set:路径
        :return:
        '''
        self.moveTo(self.startDot)
        self.route.clear()
        self.route.append(self.startDot)
        self.satiety, self.comfort = self.initSC
        for dot in route_set:
            self.moveTo(dot)

    def moveTo(self, targetDot):
        """
        移动到目标点
        :param targetDot:目标点
        :return:
        """
        currentDot = self.route[-1]
        supply = targetDot.supply
        # 到目标点要消耗的SC(饱食度和舒适度)值
        loss = self.scCostCal(targetDot)
        # 更新饱食度和舒适度
        self.satiety = self.satiety + targetDot.isFood * targetDot.supply -
self.scCostCal(targetDot)
        self.comfort = self.comfort + targetDot.isBonfire * targetDot.supply -
self.scCostCal(targetDot)
        # 将目标点加入路径
        self.route.append(targetDot)
        # 更新玩家位置
        self.position = targetDot.position
        self.x, self.y, self.z = self.position
        # 打印移动过程
        # print('From', currentDot.index,
        # 'to', targetDot.index,
        # currentDot.position, '--->', targetDot.position,
        # 'S:', self.satiety,
        # 'C:', self.comfort,
        # 'Supply:', supply,
        # 'Loss:', loss
```

```
        # )

    def approachable(self, targetDot):
        """
        判断目标点是否能到达
        :param targetDot:目标点
        :return:
        """
        scCost = self.scCostCal(targetDot)
        if targetDot == self.endDot:
            return not ((self.satiety - scCost <= -3) or (self.comfort - scCost
<= -3))
        else:
            return not ((self.satiety - scCost <= -5) or (self.comfort - scCost
<= -5))

    def getApproachableSet(self, dots):
        """
        返回当前状态下可到达的点的list
        :param dots: 所有路径点
        :return: 可到达路径点
        """
        approachable_set = []
        for dot in dots:
            if dot in self.route:
                continue
            if player1.approachable(dot):
                approachable_set.append(dot)
        return approachable_set

    def scCostCal(self, targetDot):
        """
        计算到目标点消耗的舒适度和饱食度
        :param targetDot:目标点
        :return:消耗值
        """
        currentDot = self.route[-1]
        if self.z > targetDot.z:
            scCost = euclidDistance(currentDot.position, targetDot.position) * 4
/ 100
        elif self.z < targetDot.z:
            scCost = euclidDistance(currentDot.position, targetDot.position) * 6
/ 100
        else:
            scCost = euclidDistance(currentDot.position, targetDot.position) * 5
/ 100
        return scCost

    def printInfo(self):
        """
        打印当前状态
        :return:
        """
        print('Position: ', self.position, 'satiety: ', self.satiety, 'comfort:
', self.comfort, 'step: ',
              len(self.route) - 1)

    def getInfo(self):
```

```
        """
        获得当前状态
        :return: 当前状态
        """
        info = 'Position: ' + str(self.position) + ' Satiety: ' +
str(self.satiety) + ' Comfort: ' + str(
            self.comfort) + ' Step: ' + str(len(self.route) - 2)
        return info

    def printRoute(self):
        """
        打印当前整条路径
        :return:
        """
        for i, dot in enumerate(self.route):
            try:
                print('Step', i, ': ', self.route[i].index, '--->', self.route[i
+ 1].index, self.route[i].position,
                      '--->',
                      self.route[i + 1].position)
            except:
                return

    def getRoute(self):
        """
        返回玩家路径
        """
        x = []
        y = []
        z = []
        for i, dot in enumerate(self.route):
            x.append(dot.x)
            y.append(dot.y)
            z.append(dot.z)
        return x, y, z

    def getRouteLength(self):
        """
        返回路径长度
        :return:
        """
        length = 0
        for i, dot in enumerate(self.route):
            if i < len(self.route) - 1:
                length += euclidDistance(self.route[i].position, self.route[i +
1].position)
        return length
```

在路径点类和玩家类之外，还增添了计算两点位置之间欧氏距离的函数：

```python
def euclidDistance(currentPos, targetPos):
    """
    求两个位置坐标的欧式距离
    :param currentPos: 当前坐标
    :param targetPos: 目标坐标
    :return: 欧氏距离(float)
    """
    return np.sqrt(np.sum(np.square(currentPos - targetPos)))
```

2.1.3 问题分析

经过可视化分析和小组讨论，我们将本次任务题理解为路径规划问题。这类问题通常可以通过状态空间搜索的方法解决。

目前主流的路径规划方法按照搜索方式可以分为传统搜索算法、启发式搜索算法和智能算法。

传统搜索算法主要包括深度优先搜索和广度优先搜索。深度优先搜索算法（简称DFS）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点的所在边都已被探寻过，搜索将回溯到发现节点的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。广度优先搜索算法（简称BFS）又称为宽度优先搜索从起点开始，首先遍历起点周围邻近的点，然后再遍历已经遍历过的点邻近的点，逐步的向外扩散，直到找到终点。在执行算法的过程中，每个点需要记录达到该点的前一个点的位置 —父节点。这样做之后，一旦到达终点，便可以从终点开始，反过来顺着父节点的顺序找到起点，由此就构成了一条路径。由于DFS和BFS没有利用环境信息，不适合本题最短路径算法，因此这里不做过多拓展。

启发式算法最经典的当属A^*算法。A^*算法吸取了Dijkstra 算法中的当前代价，为每个边长设置权值，不停的计算每个顶点到起始顶点的距离，以获得最短路线，同时也汲取贪婪最佳优先搜索算法中不断向目标前进优势，并持续计算每个顶点到目标顶点的距离，以引导搜索队列不断想目标逼近，从而搜索更少的顶点，保持寻路的最优解。A^*算法在运算过程中，每次从优先队列中选取$f(n)$值最小（优先级最高）的节点作为下一个待遍历的节点。A^*算法使用两个集合来表示待遍历的节点，与已经遍历过的节点，这通常称之为open_set和close_set。

A^*算法性能很大程度上依赖于启发函数$f(n)$的设计。$f(n)=g(n)+h(n)$，$f(n)$是节点n的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。$g(n)$ 是节点n距离起点的代价。$h(n)$是节点n距离终点的预计代价，这也就是A^*算法的启发函数。估价值由顶点到起始顶点的距离（代价）加上顶点到目标顶点的距离（启发函数）之和构成。过调节启发函数我们可以控制算法的速度和精确度。因为在一些情况，我们可能未必需要最短路径，而是希望能够尽快找到一个路径即可。

智能算法有很多，例如粒子群算法和蚁群算法。粒子群算法（Particle swarm optimization，PSO）是模拟群体智能所建立起来的一种优化算法，主要用于解决最优化问题（optimization problems）。1995年由 Eberhart和Kennedy 提出，是基于对鸟群觅食行为的研究和模拟而来的。蚁群算法是一种用来寻找优化路径的概率型算法。它由Marco Dorigo于1992年在他的博士论文中提出，其灵感来源于蚂蚁在寻找食物过程中发现路径的行为。这种算法具有分布计算、信息正反馈和启发式搜索的特征，本质上是进化算法中的一种启发式全局优化算法。

在本次小组作业前期的探索中，我和孙浏鑫同学兵分两路分别尝试了A^*算法和蚁群算法。

针对基于蚁群算法的思路，孙浏鑫同学已完成了三维TSP动态规划最小路径的程序实现工作。但是在添加饱食度和舒适度的限定条件时，遇到了较大困难，同时由于路径规划过程中涉及点数较多，更新速度也过慢，因此最终这条路径被我们放弃了。

最终我们选择了使用A^*算法作为本次建模的核心算法。

2.2 问题1

> 规划该人物从起点到终点的路线，使其经过的食物点和篝火点的次数最少。

2.2.1 A^*算法：

传统的A^*算法

以下是传统的A^*算法的流程：

```
*初始化open_set和close_set；
*将起点加入open_set中，并设置优先级为0（优先级最高）；
*如果open_set不为空，则从open_set中选取优先级最高的节点n：
    *如果节点n为终点，则：
        *从终点开始逐步追踪parent节点，一直达到起点；
        *返回找到的结果路径，算法结束；
    *如果节点n不是终点，则：
        *将节点n从open_set中删除，并加入close_set中；
        *遍历节点n所有的邻近节点：
            *如果邻近节点m在close_set中，则：
                *跳过，选取下一个邻近节点
            *如果邻近节点m不在open_set中，则：
                *设置节点m的parent为节点n
                *计算节点m的优先级
                *将节点m加入open_set中
```

传统的A*算法无法直接套用到本题当中，原因如下：

1. 传统的状态空间搜索算法是先搜索到终点再反推路径的，对路径没有记忆性。但本题中需要先记住当前路径的历史才能够根据玩家当前舒适度和饱食度推算下一步能够到达哪些路径点。
2. 传统A*算法中使用单一的open_set优先级队列保存待搜索节点，每个节点保存parent节点。但本题中前置路径变化会导致结点的后继空间发生变化，无法简单的用parent保存。

针对以上情况，我对A*算法进行了一些改进，以适应本题的要求：

改进的A^*算法

改进后的A^*算法流程如下：

```
*初始化route_set，open_set和approachable_set
*将起点加入route_set中
*更新当前player状态
*更新当前approachable_set，若节点在route_set中，则不加入
*将approachable_set加入到open_set中
*如果open_set不为空且route_set长度小于25(剪枝常数)，则：
    *如果open_set[-1]不为空，则从open_set[-1]中根据启发式选取优先级最高的点n:
        *如果节点为终点，则：
            *将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入route_set中
            *更新当前player状态
            *保存完整路径
            *continue
        *如果节点n不是终点，则：
            *将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入route_set中
            *更新当前player状态
            *更新当前approachable_set，若节点在route_set中，则不加入
            *将approachable_set加入到open_set中
    *如果open_set[-1]为空，则：
        *open_set.pop()
```

```
        *route_set.pop()
        *更新当前player状态
```

1. 首先针对路径无记忆性的问题：

   通过增加一个route_set用以保存当前玩家的历史路径，并在每次搜索过程中使用route_set更新玩家舒适度饱食度状态，并根据当前状态计算出一个可到达的后继节点列表approachable_set，并将整个approachable_set加入到open_set中。route_set、open_set和approachable_set均为栈。

2. 针对节点后继问题：

   每次搜索过程中从open_set的最后一个元素中根据启发式函数弹出优先级最高的后继节点，并将其加入到route_set中去，重复更新玩家状态和approachable_set，并将approachable_set加入到open_set中。若open_set的最后一个元素为空，则说明当前路径下无法找到后继节点。这时我们将open_set和route_set[]中最后一个元素弹出，并让玩家退后一个节点，继续搜索。

改进后的A^*算法最核心之处在于：

1. 每次搜索的后继节点作为一个整体（approachable_set）被加入到open_set中。每次执行下一次搜索时，先判断open_set的最后一个元素列表是否为空。为空就表明这条路径已经不存在可行的后继结点，并将open_set的最后一个元素弹出。
2. 使用route_set替代传统A^*算法的close_set，相当于每次搜索都记录了历史路径，而不是将所有搜索过的节点无序保存。完美避免了使用parent节点进行回溯导致无法更新玩家历史状态的问题。

   改进的A^*算法代码实现如下：

```
# searchResult用于保存搜索的可行路径结果
    searchResult = []
    # 初始化open_set, route_set和approachable_set
    route_set, open_set, approachable_set = [[], [], []]
    # 将起点加入route_set中
    route_set.append(dots[0])
    # 更新当前player状态
    player.update(route_set)
    # 更新当前approachable_set，若节点在route_set中，则不加入
    approachable_set = player.getApproachableSet(dots)
    # 将approachable_set加入到open_set中
    open_set.append(approachable_set[:])
    # 如果open_set不为空
    while open_set:
        # 如果open_set[-1]不为空
        if open_set[-1]:
            # 从open_set[-1]中根据*启发式*选取优先级最高的点n:
            i, bestDot = strategy(player, open_set[-1][:])
            # 如果节点为终点
            if bestDot == player.endDot:
                res += 1
                print(res)
                # 将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入route_set中
                route_set.append(open_set[-1].pop(i))
                # 更新当前player状态
                player.update(route_set)
                if res <= epoch:  # epoch次搜索
                    # 进行下一次搜索
                    route_set.pop()
                    player.update(route_set)
                    continue
```

```
                else:
                    return searchResult
            # 如果节点n不是终点
            else:
                # 将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入route_set中
                route_set.append(open_set[-1].pop(i))
                # 更新当前player状态
                player.update(route_set)
                # 更新当前approachable_set，若节点在route_set中，则不加入
                approachable_set.clear()
                approachable_set = player.getApproachableSet(dots)
                # 将approachable_set加入到open_set中
                open_set.append(approachable_set[:])
        # 如果open_set[-1]为空
        else:
            open_set.pop()
            route_set.pop()
            player.update(route_set)
```

还尝试了当搜索路径长度大于50时剪枝，但在实际搜索过程中发现没有必要，因此不在此赘述。

2.2.2 启发式设计

要使经过的点最少，就是要尽可能**每一步都走消耗最小且饱食度和舒适度最高的点**。其实就是让[目标点的能量值-到目标点消耗的能量]尽可能大。以此作为搜索算法的损失函数。

经过思考实际上应该是使[当前点到终点消耗值-目标点到终点消耗值+目标点的能量值-到目标点消耗的能量]尽可能大且大于0。

启发式函数如下：

$$max(当前点到终点消耗能量 - 目标点到终点消耗能量 + 目标点的能量 - 到目标点消耗的能量)$$

代码实现如下：

```
def leastDotsStrategy(player, approachable_set):
    """
    第一问策略：
    MAX[当前点到终点消耗值 - 目标点到终点消耗值 + 目标点的能量值 - 到目标点消耗的能量]
    """
    currentDot = player.route[-1]
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None
    cost = -float('inf')
    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = scCostCal(currentDot, endDot) - scCostCal(dot, endDot) +
dot.supply - scCostCal(currentDot, dot)
            if tmp > cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot
```

2.3 问题2

> 在第一问的基础上，进一步考虑人物行走的路径尽可能短。

启发式设计

与问题一相似，其实只要保证让[目标点的能量值-到目标点消耗的能量]尽可能大的同时**尽可能接近终点。**

启发式函数如下：

$$min(目标点到终点的距离)$$

代码实现如下：

```
def leastDistanceStrategy(player, approachable_set):
    """
    第二问策略
    使[目标点到终点距离]尽可能小
    """
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None

    cost = float('inf')

    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = euclidDistance(dot.position, endDot.position)
            if tmp < cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot
```

2.4 问题3

> 进一步考虑人物可制作火把携带的方案，使得人物到达终点后的饱食度和舒适度尽可能高。

在篝火点，游戏人物可以制作火把携带，制作火把将使得饱食度降低0.5个单位，但携带的火把能支持人物行走20米而不降低舒适度。本质上就是**在每一个篝火点增加一个可选的项：用0.5个饱食度换取1个舒适度。**

在搜索过程中增加制作火把策略：当路径点为篝火点时，若当前饱食度高于舒适度0.5以上，则进行篝火制作。

启发式设计

与前两问相似，其实只要保证让[目标点的能量值-到目标点消耗的能量]尽可能大。

$$max(目标点的能量-到目标点消耗的能量)$$

代码实现如下：

```
def maxSCStrategy(player, approachable_set):
```

```
    """
    第三问策略
    max[目标点的能量-到目标点消耗的能量]
    """
    currentDot = player.route[-1]
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None
    cost = -float('inf')
    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = dot.supply - scCostCal(currentDot, dot)
            if tmp > cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot
```

3 结果分析

为了更好对比不同启发式对算法的影响，我们以搜索到的前500条可行路径为结果进行分析。

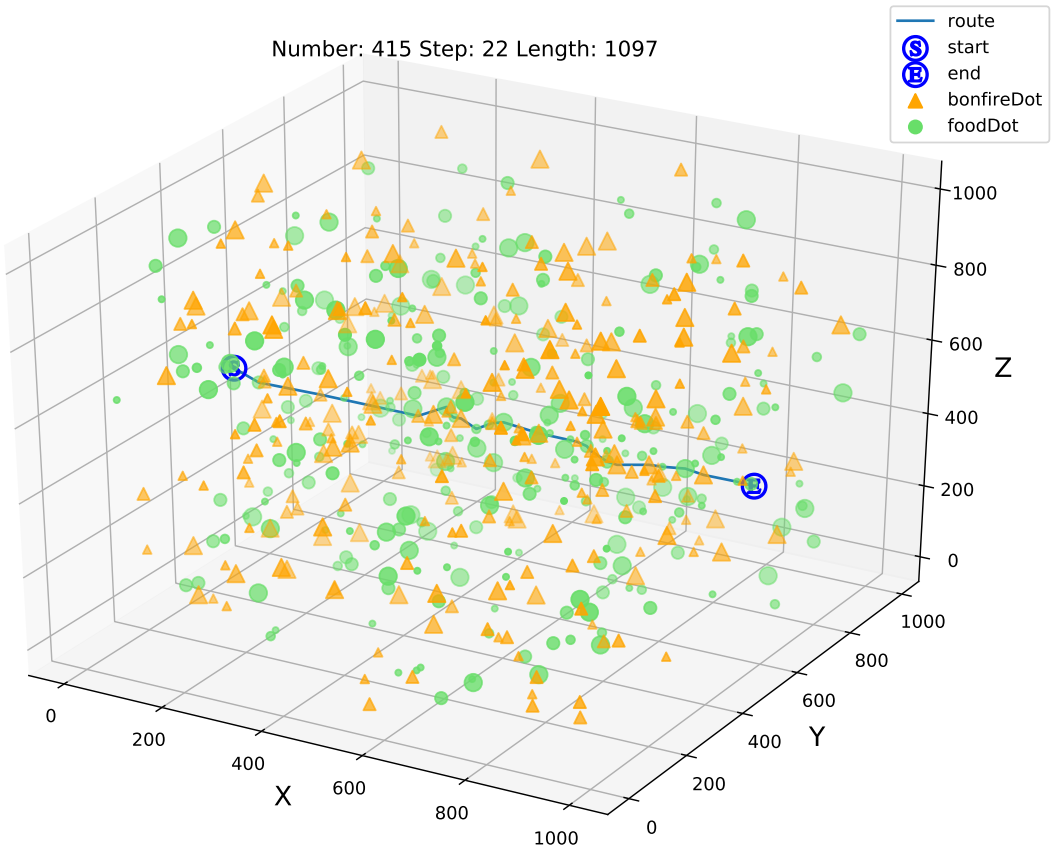
问题1

在第415次搜索中得到经过最少补给次数的路径，为22次，耗时69秒。

路径点序列为：

[0, 25, 91, 200, 232, 230, 243, 254, 270, 296, 307, 338, 387, 402, 411, 434, 452, 479, 501, 518, 537, 569, 587]

到终点时剩余的补给：饱食度-1.86 舒适度-2.86。



详细信息如下：

```
Position: [1000.     555.67   442.022]
Satiety: -1.8662648167212543 Comfort: -2.8662648167212534 Step: 22
time_cost:69.64824748039246
```

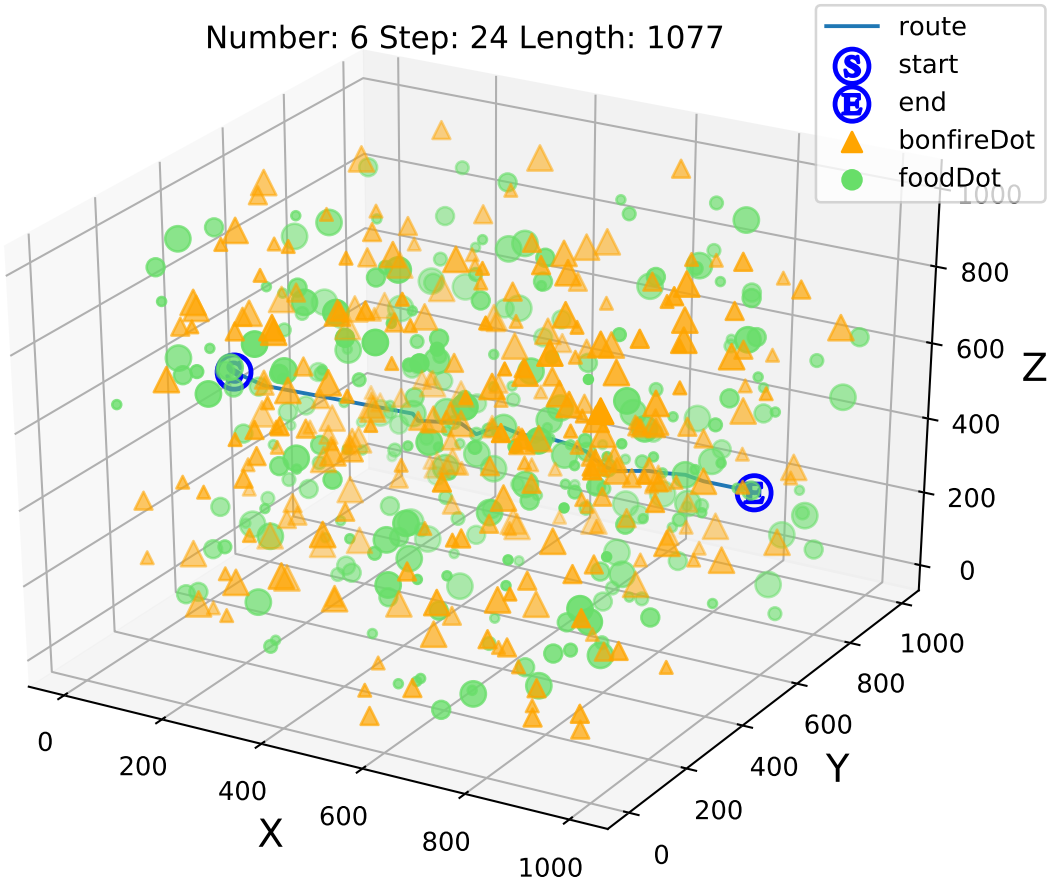
问题2

在第6次搜索中得到经过距离最短的路径，为1077米，耗时5秒。

路径点序列为：

[0, 33, 191, 200, 243, 254, 270, 296, 307, 326, 338, 345, 374, 387, 402, 411, 434, 452, 479, 501, 518, 537, 569, 576, 587]

到终点时剩余的补给：饱食度-2.58 舒适度-1.58。

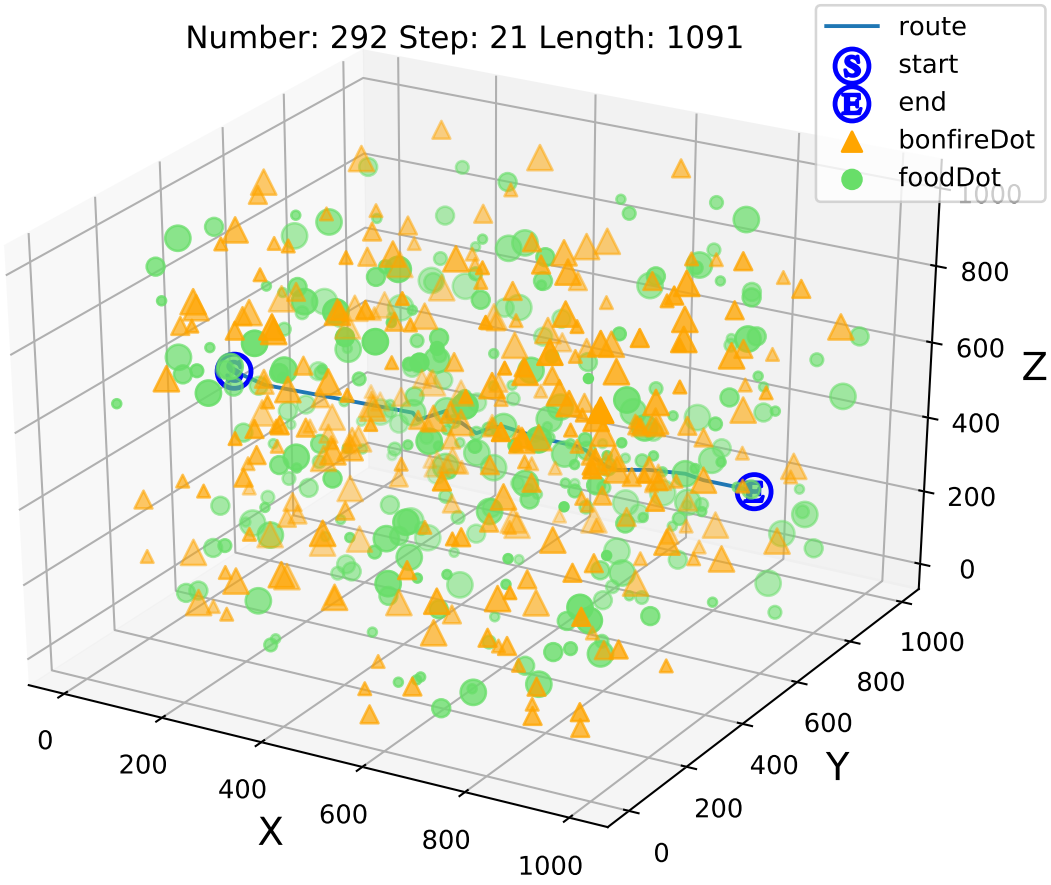


值得注意的是，在第292次搜索中得到了比第一问更好的结果，一条21个点的路径：

路径点序列为：

[0, 33, 191, 200, 232, 230, 243, 254, 270, 296, 307, 338, 387, 402, 411, 434, 452, 479, 518, 537, 569, 587]

到终点时剩余的补给：饱食度-2.63 舒适度-2.63。



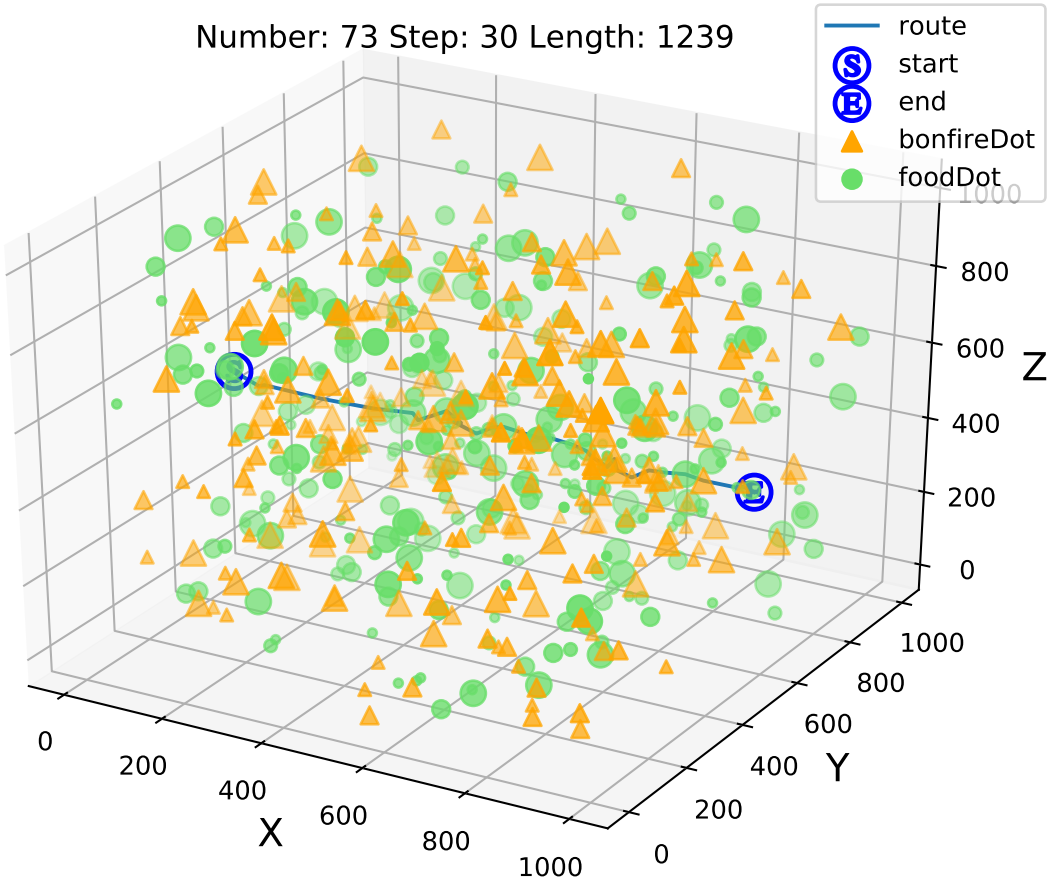
问题3

在第73次搜索中得到终点处饱食度和舒适度最高的路径，饱食度为0.95，舒适度为0.946 ，耗时8秒。

路径点序列为：

[0, 25, 33, 91, 133, 191, 200, 232, 254, 243, 270, 296, 307, 338, 345, 374, 387, 402, 411, 434, 446, 452, 463, 479, 501, 518, 537, 569, 576, 586, 587]

相应的移动距离和经过的补给点数量显著增加：经过30个路径点，路径长度为1239。



总结

综合来看，经过恰当的设定A^*算法的启发式，搜索算法的倾向性策略有明显的区分，并很好的完成了三个问题的路径搜索。实际上，若不限制搜索结果数量为500，而是无限制的搜索，不难发现可行路径数量是无限的（至少在有限时间内无法结束算法）。因此使用普通的深度优先搜索是无法在有限时间内逼近最优结果的，这也是A^*算法的价值所在。

4 心得体会

首先简单介绍本次作业分工情况：

作为组长，尤冠杰同学在前期进行了问题分析和技术路线的调研，并协调组员进行任务分工。

孙浏鑫同学之前有使用智能搜索算法的经验，因此主要负责在前期进行智能搜索算法的尝试，以及论文资料搜集。尤冠杰同学在上个学期选修了《人工智能》课程并接触了A^*算法，因此主要负责A^*算法及其改进模型的搭建、数据可视化模块以及最后论文的撰写。邓慧明同学主要负责数据的整理和论文的撰写。

本次建模过程中，我们小组的三人分工协作，发挥各自特长，以较高的效率完成了建模任务。

本次作业与期中的图像分析的很大不同之处在于：图像处理和分析任务通常有大量已有的库供我们使用，因此只要选择好技术路径，直接当掉包侠就好；但这次的任务虽然看起来简单，但需要我们自己从头搭建整个游戏模型和模拟玩家游戏过程，因此对我们的编码能力提出了一定挑战。经过这次任务，我学到了很多python编码的实用技巧，同时首次实用面向对象的方法实现了游戏环境的模拟，同时还学会了实用axes3D进行3d图像的绘制；在算法方面，由于传统的A^*算法无法简单的套用到我们的任务当中，因此在如何改进优化A^*算法上我走了许多弯路，包括尝试将问题转化为图论问题、尝试使用强化学习让玩家自己行走、使用蚁群算法等等，但都由于时间关系未能达到很好的效果。在经过大量的资料搜索后，我最终摸索到了通过增加route_set解决A^*算法路径记忆问题的方法，并最终实现了三个问题的搜索算法，较好的完成了任务。

最后感谢王丹老师一直以来以来的精彩授课和课下答疑，让我们能顺利的完成本次作业！

5 附录

0. 完整代码运行需要将数据文件处理为csv格式并命名为'data.csv'，格式如下：

0	0	500	500	0	0
1	3.02638	656.271	618.879	0	4
2	4.12313	605.713	947.624	0	4
3	7.92612	841.923	473.195	1	3
4	10.5955	571.063	355.66	0	4
5	15.9735	179.247	170.539	0	2

1. 改进的A^*算法

```
def ygjAStar(player, dots, strategy, epoch=50):
    """
    改进的A*算法
    :param player: 玩家
    :param dots: 路径点
    :param strategy: 启发式策略
    :return:
    """
    def saveRoute(filename, player, time_cost, loop):
        """
        保存路径到txt文件
        :param filename:
        :param player:
        :param time_cost:
        :param loop:
        :return:
        """
        with open(filename, 'a') as f:
            f.write(player.getInfo())
            f.write(' time_cost:' + str(time_cost) + '\n')
            line = '['
            for i, dot in enumerate(player.route):
                line = line + str(dot.index) + ', '
            line = line + ']\n'
            f.write(line)
            for i, dot in enumerate(player.route):
                try:
```

```
                    line = 'Step' + str(i) + ': ' +
str(player.route[i].index) + '--->' + str(
                        player.route[i + 1].index) +
str(player.route[i].position) + '--->' + str(
                        player.route[i + 1].position)
                    f.write(line)
                    f.write('\n')
                except:
                    break

    time_start = time.time()
    res = 0
    minRouteLength = float('inf')

    # searchResult用于保存搜索的可行路径结果
    searchResult = []
    # 初始化open_set, route_set和approachable_set
    route_set, open_set, approachable_set = [[], [], []]
    # 将起点加入route_set中
    route_set.append(dots[0])
    # 更新当前player状态
    player.update(route_set)
    # 更新当前approachable_set，若节点在route_set中，则不加入
    approachable_set = player.getApproachableSet(dots)
    # 将approachable_set加入到open_set中
    open_set.append(approachable_set[:])
    # 如果open_set不为空
    while open_set:
        # 如果open_set[-1]不为空
        if open_set[-1]:
            # 从open_set[-1]中根据*启发式*选取优先级最高的点n:
            i, bestDot = strategy(player, open_set[-1][:])
            # 如果节点为终点
            if bestDot == player.endDot:
                res += 1
                print(res)
                # 将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入
route_set中
                route_set.append(open_set[-1].pop(i))
                # 更新当前player状态
                player.update(route_set)
                # 记录route_set(不小于历史最优的路径不予以保存)
                if True:
                    minRouteLength = len(route_set) - 1
                    time_end = time.time()
                    time_cost = time_end - time_start
                    filename = './route/' + str(res) + '-' +
str(len(route_set) - 1) + '.txt'
                    saveRoute(filename, player1, time_cost, res)
                    searchResult.append([res, player.getRouteLength(),
player.getRoute()])
                if res <= epoch:  # 50次搜索
                    # 进行下一次搜索
                    route_set.pop()
                    player.update(route_set)
                    continue
                else:
                    return searchResult
```

```
            # 如果节点n不是终点
            else:
                # 将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入
route_set中
                route_set.append(open_set[-1].pop(i))
                # 更新当前player状态
                player.update(route_set)
                # 更新当前approachable_set，若节点在route_set中，则不加入
                approachable_set.clear()
                approachable_set = player.getApproachableSet(dots)
                # 将approachable_set加入到open_set中
                open_set.append(approachable_set[:])
        # 如果open_set[-1]为空
        else:
            # print('back!')
            open_set.pop()
            route_set.pop()
            player.update(route_set)
        # 显示部分
        # i = os.system("cls")
        # print('----------------------------')
        # print(len(open_set))
        # player.printInfo()
        # print(len(player.getApproachableSet(dots)))
        # player.printRoute()
        # print('----------------------------')
```

2. 问题一启发式策略

```
def leastDotsStrategy(player, approachable_set):
    """
    第一问策略：
    使[当前点到终点消耗值 - 目标点到终点消耗值 + 目标点的能量值 - 到目标点消耗的能量]尽可
能大且大于0
    """
    currentDot = player.route[-1]
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None
    cost = -float('inf')
    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = scCostCal(currentDot, endDot) - scCostCal(dot, endDot) +
dot.supply - scCostCal(currentDot, dot)
            if tmp > cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot
```

3. 问题二启发式策略

```
def leastDistanceStrategy(player, approachable_set):
```

```
    """
    第二问策略
    使[目标点到终点距离]尽可能小
    """
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None

    cost = float('inf')

    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = euclidDistance(dot.position, endDot.position)
            if tmp < cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot
```

4. 问题三启发式策略

```
def maxSCStrategy(player, approachable_set):
    """
    第三问策略
    """
    currentDot = player.route[-1]
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None
    cost = -float('inf')
    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = dot.supply - scCostCal(currentDot, dot)
            if tmp > cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot
```

5. 完整代码（需要在main进程中的调整策略以选择不同启发式）

```
#!usr/bin/env python
# -*- coding: utf-8 -*-
# @author:ygj
# @file: 02-ClassDot.py


import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
import os
import time

'''
Search:ygjAStar
Q1:leastDotsStrategy
Q2:leastDistanceStrategy
Q3:maxSCStrategy
'''


class Dot:
    def __init__(self, index, position, food, supply):
        """
        路径点类
        :param index: 序号
        :param position: 位置
        :param isFood: 是否为食物
        :param isBonfire: 是否为篝火
        :param isArrived: 是否到达过
        :param supply: 补给数
        :param x, y, z: 坐标
        """
        self.index = int(index)
        self.position = position
        self.x, self.y, self.z = self.position
        self.isFood = bool(food)
        self.isBonfire = not bool(food)
        self.supply = supply

    def __eq__(self, other):
        """
        =运算符重载
        判断两点是否相同
        """
        return (self.index == other.index) and (self.x == other.x) and
(self.y == other.y) and (self.z == other.z)

    def info(self):
        """
        查看当前路径点信息
        """
        if self.isFood:
            print('Dot', self.index, ' Position: ', self.position, 'Food
Supply: ', self.supply)
        elif self.isBonfire:
            print('Dot', self.index, ' Position: ', self.position, 'BonFire
Supply: ', self.supply)


class Player:
    def __init__(self, startDot, endDot, satiety, comfort):
        """
        玩家类
        :param startDot:起点
        :param endDot:终点
        :param satiety:饱食度
        :param comfort:舒适度
```

```
        """
        self.route = [startDot]
        self.position = startDot.position
        self.x, self.y, self.z = self.position
        self.startDot = startDot
        self.endDot = endDot
        self.satiety, self.comfort = satiety, comfort
        self.initSC = [satiety, comfort]

    def update(self, route_set):
        '''
        更新路径信息
        :param route_set:路径
        :return:
        '''
        self.moveTo(self.startDot)
        self.route.clear()
        self.route.append(self.startDot)
        self.satiety, self.comfort = self.initSC
        for dot in route_set:
            self.moveTo(dot)

    def moveTo(self, targetDot):
        """
        移动到目标点
        :param targetDot:目标点
        :return:
        """
        currentDot = self.route[-1]
        supply = targetDot.supply
        # 到目标点要消耗的SC(饱食度和舒适度)值
        loss = self.scCostCal(targetDot)
        # 更新饱食度和舒适度
        self.satiety = self.satiety + targetDot.isFood * targetDot.supply -
self.scCostCal(targetDot)
        self.comfort = self.comfort + targetDot.isBonfire * targetDot.supply
- self.scCostCal(targetDot)
        # 将目标点加入路径
        self.route.append(targetDot)
        # 更新玩家位置
        self.position = targetDot.position
        self.x, self.y, self.z = self.position
        # 打印移动过程
        # print('From', currentDot.index,
        # 'to', targetDot.index,
        # currentDot.position, '--->', targetDot.position,
        # 'S:', self.satiety,
        # 'C:', self.comfort,
        # 'Supply:', supply,
        # 'Loss:', loss
        # )

    def approachable(self, targetDot):
        """
        判断目标点是否能到达
        :param targetDot:目标点
        :return:
        """
```

```
        scCost = self.scCostCal(targetDot)
        if targetDot == self.endDot:
            return not ((self.satiety - scCost <= -3) or (self.comfort -
scCost <= -3))
        else:
            return not ((self.satiety - scCost <= -5) or (self.comfort -
scCost <= -5))

    def getApproachableSet(self, dots):
        """
        返回当前状态下可到达的点的list
        :param dots: 所有路径点
        :return: 可到达路径点
        """
        approachable_set = []
        for dot in dots:
            if dot in self.route:
                continue
            if player1.approachable(dot):
                approachable_set.append(dot)
        return approachable_set

    def scCostCal(self, targetDot):
        """
        计算到目标点消耗的舒适度和饱食度
        :param targetDot:目标点
        :return:消耗值
        """
        currentDot = self.route[-1]
        if self.z > targetDot.z:
            scCost = euclidDistance(currentDot.position, targetDot.position)
* 4 / 100
        elif self.z < targetDot.z:
            scCost = euclidDistance(currentDot.position, targetDot.position)
* 6 / 100
        else:
            scCost = euclidDistance(currentDot.position, targetDot.position)
* 5 / 100
        return scCost

    def printInfo(self):
        """
        打印当前状态
        :return:
        """
        print('Position: ', self.position, 'satiety: ', self.satiety,
'comfort: ', self.comfort, 'step: ',
              len(self.route) - 1)

    def getInfo(self):
        """
        获得当前状态
        :return: 当前状态
        """
        info = 'Position: ' + str(self.position) + ' Satiety: ' +
str(self.satiety) + ' Comfort: ' + str(
            self.comfort) + ' Step: ' + str(len(self.route) - 2)
        return info
```

```
    def printRoute(self):
        """
        打印当前整条路径
        :return:
        """
        for i, dot in enumerate(self.route):
            try:
                print('Step', i, ': ', self.route[i].index, '--->',
self.route[i + 1].index, self.route[i].position,
                      '--->',
                      self.route[i + 1].position)
            except:
                return

    def getRoute(self):
        """
        返回玩家路径
        """
        x = []
        y = []
        z = []
        for i, dot in enumerate(self.route):
            x.append(dot.x)
            y.append(dot.y)
            z.append(dot.z)
        return x, y, z

    def getRouteLength(self):
        """
        返回路径长度
        :return:
        """
        length = 0
        for i, dot in enumerate(self.route):
            if i < len(self.route) - 1:
                length += euclidDistance(self.route[i].position,
self.route[i + 1].position)
        return length


def euclidDistance(currentPos, targetPos):
    """
    求两个位置坐标的欧式距离
    :param currentPos: 当前坐标
    :param targetPos: 目标坐标
    :return: 欧氏距离(float)
    """
    return np.sqrt(np.sum(np.square(currentPos - targetPos)))


def scCostCal(startDot, targetDot):
    '''
    求两个点间的饱食度舒适度消耗
    :param startDot:起点
    :param targetDot:终点
    :return:消耗值
    '''
```

```
    if startDot.z > targetDot.z:
        scCost = euclidDistance(startDot.position, targetDot.position) * 4 /
100
    elif startDot.z < targetDot.z:
        scCost = euclidDistance(startDot.position, targetDot.position) * 6 /
100
    else:
        scCost = euclidDistance(startDot.position, targetDot.position) * 5 /
100
    return scCost


def routeVisual(searchResult):
    # 路径信息：第n条，长度l，路径点
    n, l, route = searchResult
    x, y, z = route[0], route[1], route[2]
    # Load data
    dots = np.loadtxt('./data.csv', delimiter=',')
    routeDots = dots[1:-1, :]
    bonfireIndex = [i for i, x in enumerate(routeDots[:, 4].tolist()) if x
== 0]
    foodIndex = [i for i, x in enumerate(routeDots[:, 4].tolist()) if x ==
1]
    startDot = dots[0, :]  # 起点
    endDot = dots[-1, :]  # 终点
    bonfireDots = routeDots[bonfireIndex, :]  # 篝火点
    foodDots = routeDots[foodIndex, :]  # 食物点
    x1, y1, z1 = bonfireDots[:, 1], bonfireDots[:, 2], bonfireDots[:, 3]
    x2, y2, z2 = foodDots[:, 1], foodDots[:, 2], foodDots[:, 3]

    # 绘图
    fig = plt.figure()
    ax = Axes3D(fig)
    # 图例设置
    startMarker = '$\circledS$'
    endMarker = '$\circledE$'

    # 散点绘制
    ax.scatter(startDot[1], startDot[2], startDot[3], c='b', s=200,
marker=startMarker, label='start')
    ax.scatter(endDot[1], endDot[2], endDot[3], c='b', s=200,
marker=endMarker, label='end')
    ax.scatter(x1, y1, z1, c='#FFA500', s=np.exp2(bonfireDots[:, 5]) * 6,
marker='^', label='bonfireDot')
    ax.scatter(x2, y2, z2, c='#68DE69', s=np.exp2(foodDots[:, 5]) * 3,
label='foodDot')

    # 路径绘制
    ax.plot(x, y, z, label='route')

    # 添加坐标轴(顺序是Z, Y, X)
    ax.set_zlabel('Z', fontdict={'size': 15, 'color': 'black'})
    ax.set_ylabel('Y', fontdict={'size': 15, 'color': 'black'})
    ax.set_xlabel('X', fontdict={'size': 15, 'color': 'black'})
    # 添加图例
    ax.legend(loc='best')
```

```
    line = 'Number: ' + str(n) + ' Step: ' + str(len(route[0]) - 2) + '
Length: ' + str(int(round(l)))
    plt.title(line)
    # plt.show()
    savepath = str(n) + '-' + str(len(route[0]) - 2) + '-' + str(round(l)) +
'.svg'
    fig.savefig(savepath, dpi=600)
    plt.close()


def leastDotsStrategy(player, approachable_set):
    """
    第一问策略：
    使[当前点到终点消耗值 - 目标点到终点消耗值 + 目标点的能量值 - 到目标点消耗的能量]尽可
能大且大于0
    """
    currentDot = player.route[-1]
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None
    cost = -float('inf')
    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = scCostCal(currentDot, endDot) - scCostCal(dot, endDot) +
dot.supply - scCostCal(currentDot, dot)
            if tmp > cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot


def leastDistanceStrategy(player, approachable_set):
    """
    第二问策略
    使[目标点到终点距离]尽可能小
    """
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None

    cost = float('inf')

    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = euclidDistance(dot.position, endDot.position)
            if tmp < cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot
```

```
def maxSCStrategy(player, approachable_set):
    """
    第三问策略
    """
    currentDot = player.route[-1]
    endDot = player.endDot

    bestDotIndex = None
    bestDot = None
    cost = -float('inf')
    for i, dot in enumerate(approachable_set):
        if dot == player.endDot:
            bestDotIndex, bestDot = i, dot
            break
        else:
            tmp = dot.supply - scCostCal(currentDot, dot)
            if tmp > cost:
                bestDotIndex, bestDot = i, dot
                cost = tmp
    return bestDotIndex, bestDot


def ygjAStar(player, dots, strategy, epoch=50):
    """
    改进的A*算法
    :param player: 玩家
    :param dots: 路径点
    :param strategy: 启发式策略
    :return:
    """
    def saveRoute(filename, player, time_cost, loop):
        """
        保存路径到txt文件
        :param filename:
        :param player:
        :param time_cost:
        :param loop:
        :return:
        """
        with open(filename, 'a') as f:
            f.write(player.getInfo())
            f.write(' time_cost:' + str(time_cost) + '\n')
            line = '['
            for i, dot in enumerate(player.route):
                line = line + str(dot.index) + ', '
            line = line + ']\n'
            f.write(line)
            for i, dot in enumerate(player.route):
                try:
                    line = 'Step' + str(i) + ': ' +
str(player.route[i].index) + '--->' + str(
                        player.route[i + 1].index) +
str(player.route[i].position) + '--->' + str(
                        player.route[i + 1].position)
                    f.write(line)
                    f.write('\n')
                except:
```

```
                    break

    time_start = time.time()
    res = 0
    minRouteLength = float('inf')

    # searchResult用于保存搜索的可行路径结果
    searchResult = []
    # 初始化open_set, route_set和approachable_set
    route_set, open_set, approachable_set = [[], [], []]
    # 将起点加入route_set中
    route_set.append(dots[0])
    # 更新当前player状态
    player.update(route_set)
    # 更新当前approachable_set，若节点在route_set中，则不加入
    approachable_set = player.getApproachableSet(dots)
    # 将approachable_set加入到open_set中
    open_set.append(approachable_set[:])
    # 如果open_set不为空
    while open_set:
        # 如果open_set[-1]不为空
        if open_set[-1]:
            # 从open_set[-1]中根据*启发式*选取优先级最高的点n:
            i, bestDot = strategy(player, open_set[-1][:])
            # 如果节点为终点
            if bestDot == player.endDot:
                res += 1
                print(res)
                # 将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入
route_set中
                route_set.append(open_set[-1].pop(i))
                # 更新当前player状态
                player.update(route_set)
                # 记录route_set(不小于历史最优的路径不予以保存)
                if True:
                    minRouteLength = len(route_set) - 1
                    time_end = time.time()
                    time_cost = time_end - time_start
                    filename = './route/' + str(res) + '-' +
str(len(route_set) - 1) + '.txt'
                    saveRoute(filename, player1, time_cost, res)
                    searchResult.append([res, player.getRouteLength(),
player.getRoute()])
                if res <= epoch:  # 50次搜索
                    # 进行下一次搜索
                    route_set.pop()
                    player.update(route_set)
                    continue
                else:
                    return searchResult
            # 如果节点n不是终点
            else:
                # 将节点n从open_set[-1]中删除(open_set[-1].pop())，并加入
route_set中
                route_set.append(open_set[-1].pop(i))
                # 更新当前player状态
                player.update(route_set)
                # 更新当前approachable_set，若节点在route_set中，则不加入
```

```
                approachable_set.clear()
                approachable_set = player.getApproachableSet(dots)
                # 将approachable_set加入到open_set中
                open_set.append(approachable_set[:])
        # 如果open_set[-1]为空
        else:
            # print('back!')
            open_set.pop()
            route_set.pop()
            player.update(route_set)
        # 显示部分
        # i = os.system("cls")
        # print('------------------------------')
        # print(len(open_set))
        # player.printInfo()
        # print(len(player.getApproachableSet(dots)))
        # player.printRoute()
        # print('------------------------------')


if __name__ == "__main__":
    # Load data
    data = np.loadtxt('./data.csv', delimiter=',')
    dots = []
    for dotData in data:
        dots.append(Dot(dotData[0], dotData[1:4], dotData[4], dotData[5]))
    # Init player
    player1 = Player(dots[0], dots[-1], 10, 10)
    # A* Search
    searchResults = ygjAStar(player1, dots, maxSCStrategy, epoch=500)
    # Visualiation
    for searchResult in searchResults:
        routeVisual(searchResult)
```