数据结构课程设计报告

———贪吃蛇

刘佳翔 2021K8009908040

版本：final

日期：2023 年 7 月 12 日

1 问题分析

对于本次课程设计的这一项关于贪吃蛇的任务，我们需要至少完成一下三个任务：

1 控制障碍物的生成，且防止障碍物将地图分割成不联通的部分；

2 控制食物的生成，且防止食物生成在障碍物和蛇的格子上；

3 控制蛇吃食物，且应该做到防止蛇把自己绕进死胡同撞死。

这个问题本质上是一个数据结构中的图相关的问题，所以需要我们用到图相关的知识，比如蛇在搜索的时候需要用到大量的图搜索相关内容，并且我们还需要使用一定的数据结构来储存图的内容，对于这个问题，邻阶矩阵似乎更为合适，因为对于贪吃蛇的问题而言，图中节点的结构都十分相近，在图搜索的时候方式也较为单一，所以邻接矩阵更为合适，并且这里的邻接矩阵更可以简化成隐式的邻接矩阵，因为我们可以直接用一个二维向量存储地图中的所有节点，进而通过对索引坐标的加或减来更快速的找到相应的节点进而实现搜索，因此我们可以用最简单的结构来保存整张地图并实现访问。

2 数据结构设计与实现

在本次任务中，我们主要构建了一个基本的类Node，其余几个主要的类我们都继承了这个类，从而构建了食物类Food，陷阱类Pit以及我们的蛇类Snake。对于整张地图，我们则是如上所诉利用了一个名为 Board 的二维向量来储存地图上每个节点的相关信息。

2.1 节点 Node

对与 Node 类我们将其定义放在了little_things.h文件中，主要封装了在图形化界面中对节点的绘制，以及节点的类别，和对节点坐标的更新相关函数，都属于基本函数，是为了便于后续对每个重要类别实现构造的，我们则不详细介绍。

2.2 陷阱类 Pit

对于陷阱类，我们将其定义放在了snake.h文件中，其继承了节点类 Node。对于节点类，我们主要介绍节点的生成。对于节点的生成，其首先应该放在蛇类的生成之后，防止陷阱生成在蛇的身上。而在算法的构造方面，我们最终选择以追求速度为前提，我们选择将陷阱随机生成在奇数行和奇数列相交的格子上，这样的话，以陷阱为中心的周围四个邻接格子上必然不会有新的陷阱生成从而我们无论陷阱的个数选择多少，我们终将不会将地图划分为不可分割的几部分。我们这样选择主要有一下几个考虑：首先是算法的速度，因为我们的任务重点应该是蛇的路径规划，所以我们应该将其他地方的算法速度加快，诚然，我们可以选择构造哈密顿回路的算法，但是找到图中的哈密顿回路是个 NP-hard 问题，假如我们想根据陷阱的生成构造哈密顿回路，那么算法的效率必然指数下降。其次是防止极端情况的产生，对于贪吃蛇问题，我们主要试想让地图中的更多的节点能够形成哈密顿回路，从而能够让蛇一直吃到食物，且能够将地图尽可能的填满，对于我们的算法，我们经过多次实验发现，对于一个节点的生成，最多导致一个左右的空白节点不能在哈密顿回路上，而对于其他算法，我们无法保证能够不产生一些极端情况。综上考量，我们选择了这个简单但稳定的算法。算法的伪代码如下:

```
>> gengerate_pit(Board):
>>    for (int i = 1; i < real_size - 1; i+=2)
>>      for (int j = 1; j < real_size - 1; j+=2)
>>        if (Board[i][j] == NONE)
>>        随机的令: Board[i][j] = PITS
```

2.3 食物类 Food

对于食物类是一个更为简单的类，同样，其定义在snake.h文件中, 我们首先只需保证食物不会生成在蛇和陷阱的格子上，导致游戏必然无解。其次我们通过多次进行贪吃蛇的寻路模拟发现了一个现象，当我们的食物生成在两个陷阱之间的时候（最多和两个陷阱邻接，因为陷阱的生成规律导致的），当游戏进行到后期的时候，及蛇的身体占据

了屏幕的大半的时候，这时蛇无论如何移动都无法吃到这个食物，这是因为此时这个节点无法成为蛇最终所处的哈密顿回路的一部分，所以我们在生成食物的时候额外增加了一个检测：食物邻接的格子中，陷阱的数目是否大于一个，如果大于的话，我们则选取其他地方生成食物。对于算法的伪代码和陷阱的相同，我们则不多赘述。

2.4 蛇类 Snake

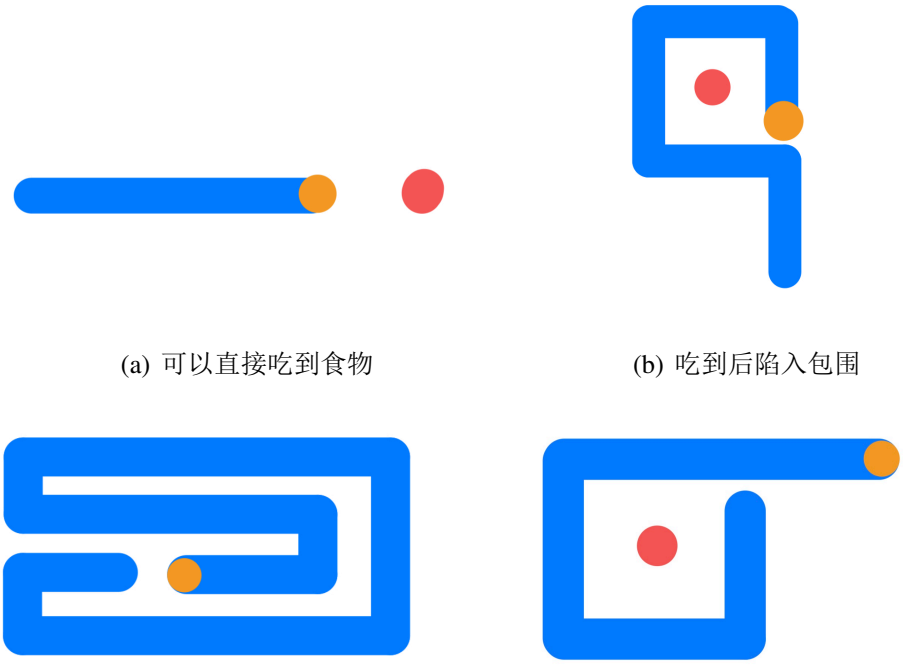
我们将重点放在蛇类的介绍上。对于蛇类的定义在snake.h文件中，对于相关函数的实现在snake.cpp文件中。对于蛇类 Snake，我们使用了一个列表 list 来储存蛇的身体中的 Node 小类，这样做的原因主要是，在设计蛇的移动方面，我们其实可以只去关注蛇的头部和尾部：在向前移动的时候，假如没有吃到食物，我们则可以不断将蛇尾弹出，而将一个新的 Node 作为蛇头加入，加入吃到食物，我们则不用将蛇的尾部弹出，因而我们在蛇移动的过程中，完全不用关注内部的任何节点。因此如果我们用 list 来储存蛇的身体的节点，我们的蛇每次移动的复杂读竟然能成为 $O(1)$！这是我们所追求的最好的情况，而假如用向量来进行存储的话，我们的算法复杂度则是蛇身长度的线性复杂度，因为对每次插入头的操作，会导致算法成为严格线性的复杂度，当游戏进行到后期的时候，这是很难以接收的。而对于遍历操作，列表和向量的复杂度应该是相当的，所以我们选择使用列表来储存蛇的身体部分。

对于其他的一些小的函数的设计，比较简单，我们则不详细介绍，详见snake.cpp。下面我们把重点放在蛇的寻路算法的的设计上，这是我们这次任务的最重要的部分。

首先我们将算法的大致思路列下：

```
>> 利用bfs所搜找到蛇头到食物的路径head_food
    >>如果head_food存在:
        先将一条虚拟的蛇移动到食物，利用bfs找到蛇头到尾巴的路径head_tail
>> 如果head_tail存在:
    采取head_food中移动的第一个方向
>> 如果head_tail不存在:
    >>找到一个远离蛇尾的方向long_head_tail，且保证按照这个方向走一步后蛇
      头还能找到蛇尾巴
    >>如果long_head_tail存在:
      采取这个方向
    >>如果long_head_tail不存在，采取bfs找到蛇头到蛇尾的路径short_head_
      tail
      >>如果short_head_tail存在，采取这个路径的第一个方向
      >>如果short_head_tail不存在，随机选取一个可行的方向
```

对于上述算法，我们首先需要明确的一点是，对于路径的耗散，都是一致的，即每一步的路径长度都是 1，因而我们可知，此时通过宽度优先搜索 bfs 找到的路径一定是最短的路径，所以我们可以不使用 Dijkstra 算法，可以进一步提高速度。其次，我们的算法实际上是相当于一个贪心算法，首先我们要找到到食物的最短路，但是一味的追寻最短路会导致蛇很大概率陷入死胡同，比如当蛇包围住食物的时候，蛇会毫不犹豫的进入自身的包围圈。因而我们要对食物的路径加以限制：当蛇吃下食物后，必然能够找到到蛇尾的路径，这样必然不可能陷入自身的包围圈。但是假如蛇吃下食物后找不到尾巴，或者无法找到去食物的路径，也就是 head_tail 不存在的时候。首先，我们是我们找不到去食物的路径，这种情况下，必然是蛇将食物包围住了，并且蛇头在包围圈外，这时我们将蛇头远离蛇尾，这样相当于向周围空旷的地区开拓，使得我们的蛇可以把包围圈打开。而如果吃下食物后找不到尾巴，这就是我们上述提到的：蛇会因为盲目吃食物而将自己引入包围圈，同样的，我们选取原理尾巴的路径。但是假如无法原理尾巴：对应的情况是，我们的蛇将自身构成了一个隧道，蛇头和蛇尾刚好位于隧道中。这时我们可以选择追上蛇尾进而希望逃出隧道。为了应对我们没有预料到的情况，我们最后采取了随机选取一个可行的行动，这里可行是指，走了一步后，能够找到尾巴，在不行，我们只能接收蛇的死亡了，但是我们的蛇基本上没有遇到这种情况，具体的各种情况如下图所示：



(a) 可以直接吃到食物　(b) 吃到后陷入包围

(c) 进入隧道　(d) 找不到去食物的路径

图 1: 贪吃蛇的几种可能情况

3 复杂度分析

首先是对食物和陷阱生成算法的复杂度分析，由于二者的算法逻辑都是遍历整个地图，多以他们的复杂度为 $O(n)$，其中 n 为地图中的节点的个数。

而对于贪吃蛇的寻路算法，我们分开计算，首先是找到食物的路径为地图节点的线性个数为 $O(n)$，对于初始化操作，也是节点的线性 $O(n)$，对于虚拟蛇的移动，每次移动为常数复杂度（我们在上面进行过分析），而蛇最多移动节点的个数步因而虚拟蛇移动复杂度也为 $O(n)$，对于找尾巴的复杂度因为也为 bfs，因而复杂度也是线性的，对于随机选取可行的方向，移动时复杂度为常数，检测能否到尾巴复杂度为 $O(n)$。因而，将所有小算法复杂度叠加，可以得到的复杂度为线性复杂度 $O(n)$，算法的效率得到了很好的保证，则复杂度会变成 $O(n^2)$。因而我们可以说，我们的算法复杂度为 $O(n)$ 算法的效率比较令人满意。

4 结果分析

在本次项目中，我们为小蛇设定了三种尺寸的地图分别是 16×16，20×20，30×30。在细节方面，首先是，为了判断我们的基础逻辑是否正确，我们在游戏中增加了一个“单人模式”，经过验证发现，蛇无论是撞到障碍物还是越界，以及是自身，蛇均会死亡。而对于蛇自身的性能，表现得较为优异，我们选取多次实验中的最低分可以看到下图：

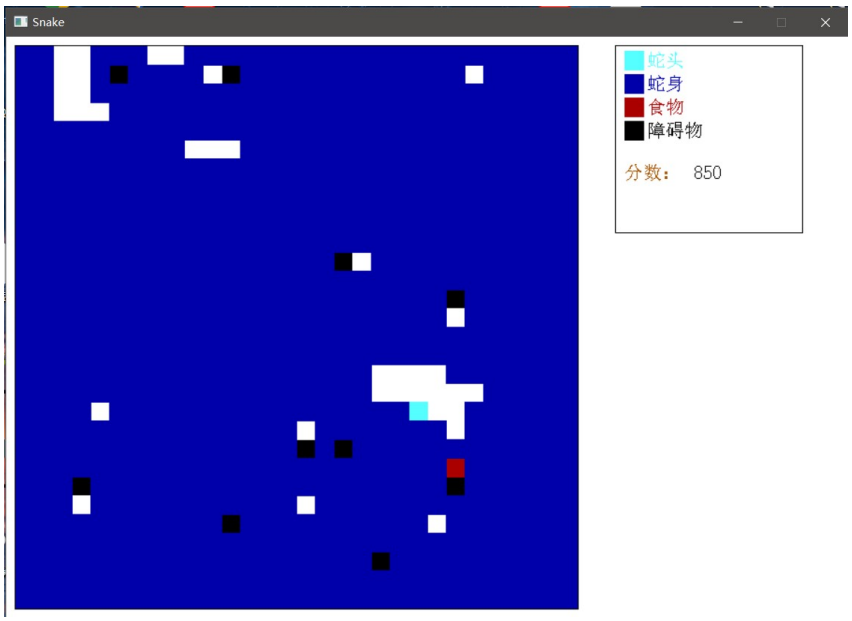


图 2: 实验中的最低分

在此时，虽说蛇仍有活动空间，但是蛇已经陷入了循环之中，所以我们及时停止了游

戏，我们下面来计算蛇此时的能效，首先是蛇此时的得分是 850 分，对于蛇的初始长度为 3，我们在计算分数的时候将初始长度减去了，所以蛇的实际长度应该为 $853+3=853$，而我们这张图中应该为大地图，所以总格数为 900，并且其中有 10 个障碍物，所以蛇占比为 $853/(900-10)=0.96$ 也即是说，此时蛇已经能够长到最大长度的 96% 了，表明最差效果也大体令人满意。蛇最终没有将格子填满的原因我们之前也进行过一定的分析，主要原因是，由于陷阱的生成，使得图中剩下的节点不一定能够构成哈密顿回路，从而使得一些节点无法被贪吃蛇访问，因而贪吃蛇的算法可以认为比较成功。对于我们贪吃蛇的平局水准见下图：

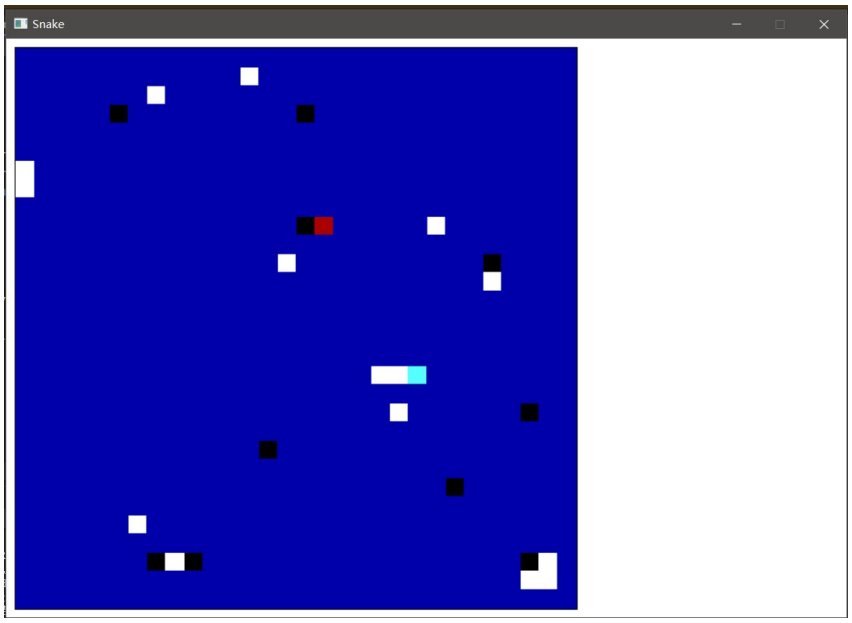


图 3: 实验中的一次随机采样

最终，为了给游戏增加难度，我们还给项目设计了人机对战模式，也就是人和 AI 进行战斗，在我们的测试中，发现上述算法仍然能够很好的发挥作用，蛇遇到人会主动得避开，蛇也会灵活的去吃到食物，证明我们的算法可以胜任障碍移动的场景，表明算法较为成功。

另外，我们还增加了机器与机器间的对战模式，其中由于蓝色的蛇会在绿色蛇移动之后再发生移动，所以为了公平起见，我们特意设计绿蛇撞到蓝蛇不会死亡。

5 成员分工

本次任务均为刘佳翔一人完成，实现了障碍物生成，食物生成，贪吃蛇寻路算法设计，以及实现了图形化界面显示，以及多条蛇的人机对战模式。