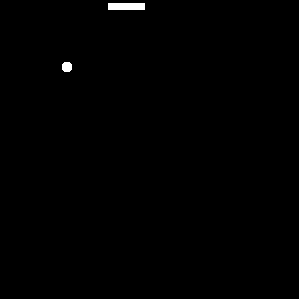
效果图：[附件\46 贪吃蛇算法.gif](附件/46   贪吃蛇算法.gif)



设计：

3个关键元素：方阵，食物，蛇

状态构成： 蛇的状态+食物位置

状态转移：蛇移动到食物的位置，转移到下一个状态

状态判定：

相似问题：吃豆人程序

本质上是搜索问题。

利用广度优先搜索，在10\*10的地图上都需要很久的时间。不过，问题已经解决了部分。

在设计这个广度优先搜索算法的时候我犯了一个严重的失误，但也能称做是典型的错误，就是没有仔细考虑条件判断，将静态蛇的位置错误地用于了所有的衍生蛇。

## 问题1：使用广度优先搜索能发现路径，但是在n次重复后，蛇很有可能只能对某一区域而非全部进行搜索

一般而言，蛇身将图分为两个区域：能到达的区域和不能到达的区域。

解决方法1（蛇身内判断）：在搜索到食物之后，先对蛇身是否将图分为内外两个区域进行判定。强条件是：如果分成两个以及以上的区域，就重新选择。

判定算法： 对蛇身进行遍历

输入： snake

输出：Divided or Not

for i in snake.all:

for j in reversed(snake.all):

如果i,j相差超过5个,且i,j曼哈顿距离为0，检查i,j之间是否存在内部点。

内部点检测算法：

输入：snake,端点i,j

输出：是否存在

已知条件：端点一定在矩形区域外侧的同一条边上

首先，蛇必须存在两个曼哈顿距离为0,1的不相邻点，这样才有可能造成封闭。其次，以这两个端点确定一个矩形区域

根据max,min的点来确定。

构造一条以端点i,j为首位的蛇k

然后对举行了区域的内部进行遍历，检查其是否属于k，如果存在不属于k的点，蛇身就构成了二分区域。

对点p具体判定方法为过点p作两条直线，检查直线与蛇身的交点个数，如果小于4个，就不是，否则就属于内部。

如果在不是曼哈顿距离足够小的地方使用判别，就会产生如下错误：

，其中的点（2，3）显然不是内部点，但被判别为内部点。

**评价方法1：实际上这种方法并不是有效的，在蛇的身体和地图尺寸相近的情况下，光依靠内部点判断是不行的**

方法2：边界确定法

确定与蛇身相邻的地图边界，然后检查这些地图边界是否可达。如果不可达，就存在隔离的点。比如：



将图分为两个部分，其边界不可达。

算法1：

确定蛇的最小矩形区域

则蛇可将矩形分为5个区域：左上，右上，左下，右下，中间

方法3：传播标记算法

如果一个点属于区域A，则另一个相邻的店也属于区域A。具体过程就是初始化矩阵A[n\*m],初始值为-1，蛇身标为0。

从第一行起每行，先找到第一个不属于蛇身的点，标记为1（区域1），然后对其周围的点进行标记。

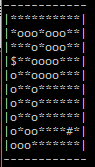
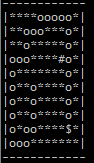
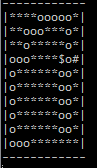
以后每个点，先检查其是否已经标记。如果没有，就寻找周围的第一个标记区域的点，将其标记为该区域。如果没有一个可用于标记的区域，则该点属于最新的未知区域（即最后一个已知区域+1）

## 问题2：蛇的搜索速度很慢

如果将这个算法和人思维相比，就会发现慢的原因之一在于它不能根据食物的位置来选择其他搜索策略。人能够看到全图，发现蛇头与食物之间的最近距离，然后使用近似算法，一步一步达到。实际上，广度优先搜索算法的复杂度总是与食物到蛇头的曼哈顿距离的高次方相关，显然，我并不希望看到这种制约。

改进建议：计算食物与蛇之间的相对位置关系

一下是一些参考：（它们计算得很慢）

算法1：启发式搜索算法 每进行一步，就对当前步骤进行评价，确定将要选择的下一个步骤。 启发式算法也包括对吃到食物之后是否会产生无解的情况进行判断，并排除这种情况