小组研讨——扫雷的简单实现

1. 数据结构的选择
2. 基本算法的实现
3. 进阶算法——旗帜的行为与交互

（我会玩一盘初级扫雷，加超链接<http://www.minesweeper.cn/>）

通过一次游戏，我们看到，扫雷游戏的基本元素就是一个个的“格子”，这个格子可能没被打开，可能什么都没有，也可能是数字、旗帜等，或者是雷，通过这样的事实，我们可以选择出适合的数据结构

1. 数据结构的选择

看到这么一个版面表格

描述已自动生成我们会很自然的想到要使用二维数组来存放这个地图，而且，无论地图上的格子是否被打开，是否被标以旗帜，这个格子的内容在游戏开始后就已经确定下来了

因此，我们可以定义两个int数组，一个map[10][10]用来存放内容，一个state[10][10]来记录格子的状态，即“未打开，被打开，旗帜，问号”等，可以使用不同的数值来区分这些状态

看到map与state这两个大小完全一致的数组，你会想到什么呢？

没错，我们可以定义这么一个结构体

typedef struct BLOCK{

int num;//记录内容物

int state;//记录状态

}BLOCK;

然后只需定义地图数组为BLOCK map[10][10]，这样这两个数组就合二为一了

你可能会问：“把这两个数组揉在一起有什么用吗？调用变得更不方便了！”

这样，我们再考虑一个问题，当我们选择数据结构时，一个原则是选择合适的数据类型。例如：如果你明知道一个整数能达到1e18的大小，就应该选择long long，使用int必然会导致错误；同样的，如果我们知道一个必然处于小范围的数据，如高考数学成绩，一定是介于0与150之间的，这时如果选择long long来存放就会显得有点多余。

所以，对于我们要存放的这个扫雷地图，我们想要减少冗余的空间，从而减少存放地图消耗的内存。毕竟在实现相同功能的前提下，占据空间越小越好嘛。

既然这样，我们考虑一下我们需要存储的内容的范围。对于格子的内容，一个格子周边的雷数只可能是0-8，或者格子本身就是雷，不可能有其他的状态了。

那么，我们可以人为规定，当一个格子是雷时，block.num=15

对于不是雷的格子，记录其周边的雷数即可

这样，我们只需要0-8以及15这几个数字就能表示出每个格子的内容了，用int来存是不是太浪费了呢？

于是，我们想到了结构体的位域这个语法

对于num，只需要存储0-8与15这几个数，分配4个比特位就够了

对于state，只有四种状态（未打开，被打开，旗帜，问号），可以用0-3来对应，于是分配2个比特位就够了，但是由于自动对齐的存在，这里分配了4个比特位，即：

typedef struct BLOCK{

int num:4;//记录内容物

int state:4;//记录状态

}BLOCK;

当然，这样写是有问题的，我们用sizeof看一下

文本, 白板

描述已自动生成 图形用户界面, 文本, 应用程序

描述已自动生成

理想很丰满，现实很骨感

我们应改成这样

typedef struct BLOCK{//地图格子信息结构体

unsigned char num:4;//记录内容物

unsigned char state:4;//记录状态

}BLOCK;

图形用户界面, 文本, 应用程序

描述已自动生成

成功了！这是因为结构体的长度一定是最长的数据元素的整数倍，使用int来定义，就算只分配1个比特位，所占内存空间也是4字节

通过使用位域，我们将原来占8字节的结构体瘦身成了1字节，缩小了存储地图占据的空间。当然，我们在对使用位域的成员赋值时，一定要注意不能超过其定义的位数，否则就会对其他成员的数据进行覆盖，进而导致地图损坏

除了地图，我们还有一个东西需要处理

在玩扫雷时，最最重要的就是我们点击的格子是什么，也就是这些格子的“坐标”，对于这么重要的，我们在将来的算法中会经常操作的东西，用一个结构体来封装起来就再好不过了

typedef struct POS{//地图坐标结构体

int row;//对应地图行数

int col;//对应地图列数

}POS;

可以类比数学中的坐标，我们想让我们自己定义的POS结构体也能进行一定的数学运算，从而便于我们将来的算法实现

这里我们选择偷偷用一下C++

加法：

POS operator+ (const POS& pos){//定义地图坐标之间的加法

return POS(row + pos.row, col + pos.col);

}

是否相等：

bool operator== (const POS& pos){//定义地图坐标之间的等于比较

return row == pos.row && col == pos.col;

}

以及最重要的，是否合法：

bool is\_legal(void) {//判断地图坐标是否合法

return row >= 1 && row <= map\_row && col >= 1 && col <= map\_col;

}

合起来是这样的

typedef struct POS{//地图坐标结构体

int row;//对应地图行数

int col;//对应地图列数

POS(int row=0, int col=0){//在不定义新的坐标结构体前提下返回一个坐标结构体常量

this->row=row;

this->col=col;

}

POS operator+ (const POS& pos){//定义地图坐标之间的加法

return POS(row + pos.row, col + pos.col);

}

bool operator== (const POS& pos){//定义地图坐标之间的等于比较

return row == pos.row && col == pos.col;

}

bool is\_legal(void) {//判断地图坐标是否合法

return row >= 1 && row <= map\_row && col >= 1 && col <= map\_col;

}//map\_row，map\_col为地图的行数与列数

}POS;

有了这么一个“结构体”，我们之后的算法实现就会便利不少

我们规定，行数与列数均从1开始，方便使用

1. 基本算法的实现

方便起见，我们将地图数组设为全局变量

埋雷算法

一个朴素的想法：随机生成行数与列数，如果对应格子不是雷，就埋雷，直到埋下的雷数达到了要求的雷数

提示：生成[a,b]的随机数为rand()%(b-a+1) + a

代码如下：

void SetMines(void)

{

srand(time(0));//设置随机数种子

int mine\_set=0;

do{

int rand\_row = rand() % map\_row + 1;

int rand\_col = rand() % map\_col + 1;

if (map[rand\_row][rand\_col].num != 15){//该格不为雷

map[rand\_row][rand\_col].num = 15;//埋雷

mine\_set++;

}while (mine\_set < map\_mines);//直到埋雷数等于地图雷数

}

对于这个朴素的想法，我们想一想边界情况

对于这样一个地图：长100，宽100，有9999个雷

使用这种上述算法，循环大概要执行100000次（当然对于计算机来说不是问题）

这是因为我们的坐标是完全随机的，在埋雷的后期有大概率会选中已经埋雷的格子，导致许多次无效的埋雷

解决方法有很多，这里我们选择反选法，即：

当雷数大于非雷数（即：地图总格数-雷数）时，先把地图全部埋雷，再随机从地图上挖雷，直到剩余的雷数等于要求的雷数

这样，就能保证不会有过多次无效的操作

代码这里就不提供了，跟上一种差不多

当然，如果你有耳闻过洗牌算法，还可以通过其来实现随机埋雷，效率比上面两个都更高

提示random\_shuffle(begin , end)，C++STL函数，以O(n)的复杂度根据随机数种子打乱数组

看代码

void SetMines(void)

{

srand(time(0));//设置随机数种子

int temp[map\_row\*map\_col];

for (int i=0; i<map\_row\*map\*col; i++)

if (i<map\_mine)

temp[i] = i<map\_mine ? 1 : 0;

std::random\_shuffle(&temp[0], &temp[map\_row\*map\_col-1]);

int cnt=0;

for (int row=1; row<=map\_row; row++){

for (int col=1; col<=map\_col; col++){

map[row][col].num = temp[cnt++]==1 ? 15 : 0;//埋雷

}

}

}

此外，我们还可以传入玩家操作的第一个格子的坐标，从而避免某些倒霉蛋第一步踩雷的情况

遍历周边格子

在之后的算法中，我们会多次用到遍历周边格子的操作，这里单独拎出来讲讲

由于我们定义了坐标结构体的加法，实现就变得简单了起来

定义全局数组POS surround[8]={{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}};

(C++11标准)

可以把这个数组的内容看成一个个向量，当这些“向量”与给定坐标相加时，就能实现遍历周边格子的效果，即：

for (int i=0; i<8; i++){

act\_pos = pos + surround[i];

/\*对act\_pos做些什么\*/

}

也可以这样写

for (auto move : surround){

POS act\_pos = pos + move;

/\*对act\_pos做些什么\*/

}//C++

填数算法

一个朴素的想法是，等到雷全部埋好后，遍历地图的每个格子，对不是雷的格子，数出周围的雷数并写入。当然，要在对数组操作前，判断坐标是否合法

void SetNum(void)

{

for (int row=1; row<=map\_row; row++){

for (int col=1; col<=map\_col; col++){//遍历每一个格子

int cnt=0;//计数器归零

if (map[act\_pos.row][act\_pos.col].num!=15){//对不是雷的格子才能填写数字

for (int i=1; i<8; i++){//遍历周边格子

POS act\_pos = POS(row, col) + surround[i];

if (act\_pos.is\_legal() && map[act\_pos.row][act\_pos.col].num==15)//坐标合法，且坐标对应格子为雷时

cnt++;//计数器加一

map[row][col].num=cnt;//写入

}

}

}

}

}

可以看到，这种算法的效率并不高

一种更高明的想法是，在埋雷时就随之更新周边的数字（采用朴素想法埋雷），实现也很简单，只要对先前的代码进行小小的修改

void SetMines(POS user\_pos)

{

srand(time(0));//设置随机数种子

int mine\_set=0;

do{

int rand\_row = rand() % map\_row + 1;

int rand\_col = rand() % map\_col + 1;

if (map[rand\_row][rand\_col].num != 15 && !(POS(rand\_row, rand\_col) == user\_pos)){//该格不为雷且不是用户点击的格子时

map[rand\_row][rand\_col].num = 15;//埋雷

mine\_set++;

for (int i=0; i<8; i++){//对周围八个格子

POS act\_pos = POS(row, col) + surround[i];

if (act\_pos.is\_legal() &&

map[act\_pos.row][act\_pos.col].num!=15){//当格子坐标合法，且不是雷时

map[act\_pos.row][act\_pos.col].num ++;//该格数字加1

}

}

}

}while (mine\_set <map\_mines);//直到埋雷数等于地图雷数

}

打开格子

容易知道，只有未打开的格子才能被打开，被标旗帜或者问号的格子都不能打开

而且，如果打开的是一个雷，游戏就结束了，所以需要传递游戏运行状态参数

并且，我们未雨绸缪一下，在将来判断游戏胜利条件时，我们需要知道打开的格子数，所以我们使用一个全局变量map\_open记录打开的格子数

代码如下

void OpenBlock(POS act\_pos, int\* game\_state)

{

if (act\_pos.is\_legal()){//当用户点击坐标合法时

BLOCK\* pblock = &map[act\_pos.row][act\_pos.col];//取个别名，方便使用

if (pblock-> state == 0){//格子没被打开时格子才能被打开

pblock-> state = 1;

map\_open++;//被打开的格子数加1

if (pblock->num == 15)//打开了雷

\*game\_state = 1;//游戏失败

}

}

}

而且我们在游戏过程中会发现，如果我们打开了一个空格（即该格周边都没有雷），游戏会自动把那一片数字都打开。这种效果的实现也很简单：当打开了一个空格时，遍历打开这个空格周边的格子即可，实现过程表现出来就是一个递归函数

void OpenBlock(POS act\_pos, int\* game\_state)

{

if (act\_pos.is\_legal()){//当用户点击坐标合法时

BLOCK\* pblock = &map[act\_pos.row][act\_pos.col];//取个别名，方便使用

if (pblock-> state == 0){//格子没被打开时格子才能被打开

pblock-> state = 1;

map\_open++;//被打开的格子数加1

if (pblock->num == 15)//打开了雷

\*game\_state = 1;//游戏失败

if (pblock->num == 0)//当周边雷数为0时

for (int i=0; i<8; i++){//遍历周边格子

if ((act\_pos + surround[i]).is\_legal())//当周边格子坐标合法时

OpenBlock(act\_pos + surround[i], game\_state);//递归打开周边的格子

}

}

}

}

改变格子的状态

实现十分简单，只需要把map[][].state按0,2,3循环变化即可

void ChangeState(BLOCK\* pblock)

{

switch (pblock->state){

case 0:

pblock->state = 2;//普通格变为旗帜

break;

case 2:

pblock->state = 3;//旗帜变为问号

break;

case 3:

pblock->state = 0;//问号变为普通格

break;

}

}

游戏地图的展示

只需按照地图格子的内容与状态来输出不同的字符即可（控制台）

这里就不给代码了

要求更高的话可以用图形界面

游玩过程

可以使用控制台输入指令，也可以使用图形界面用鼠标操作

注意需要判断输入的合法性

游戏结束的判断

游戏运行时gamestate==0

当game\_state==1时（由OpenBlock函数修改），游戏结束

当game\_state!=1，且“打开的格子数 等于 地图的总格数减去雷数”时，修改game\_state为2，游戏胜利

当game\_state!=0时，根据game\_state的不同值作出相应响应

以上便为扫雷游戏的基本流程了，将这些函数组合起来就能构成一个简单的扫雷程序了

1. 进阶算法——旗帜的行为与交互

我们知道，扫雷游戏过程中，旗帜这个东西是非常重要的，可以理清我们的思路。

那么，如何实现旗帜给我们带来的便利呢？

其实有关旗帜的交互只有一个，就是：当我们点击一个数字格，且该数字格周边的旗帜数与该数字本身相同，就会直接打开这个数字周边的格子

图片包含 游戏, 电子, 键盘, 游戏机

描述已自动生成 图片包含 游戏, 电子, 门, 键盘

描述已自动生成（图中点击了中央的数字1）

实现也是很直接的，直接上代码吧

void OpenSurround(POS act\_pos, int\* game\_state)

{

if (act\_pos.is\_legal()){

BLOCK\* pblock = &map[act\_pos.row][act\_pos.col];

if (pblock->state == 1 && pblock->num != 0 && pblock->num == CountFlags(act\_pos))//当且仅当格子被打开时且周边的旗帜数等于该格的数字

for (int i=0; i<8; i++)//遍历周边格子

if ((act\_pos + surround[i]).is\_legal()){//当周边格子坐标合法时

OpenBlock(act\_pos + surround[i], game\_state);//打开周边格子

}

}

}

int CountFlags(POS act\_pos)

{

int flag\_count=0;

for (int i=0; i<8; i++)

if (map[act\_pos.row + surround[i].row][act\_pos.col + surround[i].col].state == 2)

flag\_count++;

return flag\_count;

}

不过，对应旗帜的使用，我们认为，在图形界面游玩时更能发挥其便利效果，在控制台中并不会有使用旗帜的欲望

1. 总结

通过本次研讨，我们小组对扫雷这个游戏的实现算法有了深入的了解，并能用代码实现游戏功能，对我们的编程能力有了很大的提高

（感谢页）