离散数学大作业报告

——扫雷智能体的实现

指导老师：黄民烈

曾天宜

精 02

2010010530

一．程序功能与目的

1. 概述

扫雷是 Windows 系统上一个较为流行的游戏。本项目以 Win8 的推出以及 Win8 应用开发热浪为背景，希望能够开发出这样一个 Windows Store Application：使其不仅拥有原有扫雷游戏功能，而且能够在用户需要时给出不同程度的提示。

2. 基本功能

本款游戏的基本功能与原有扫雷游戏相同，所不同的仅仅只是内部游戏逻辑与界面的交互模式可能与之前 Windows 上有所不同。同时，整体界面的大小（即共有的方块[1]数目）以及雷的总数完全可调。还可以通过侧边选项栏选择比较常用的游戏设置。

基本规则如下：

- 整体雷区中按照初始化条件随机分布有一固定数目的雷；
- 若某方块中无雷，则该方块可以点下；
- 无雷方块中的数字表示该方块周围的有雷方块的个数；
- 若点下的雷周围雷方块个数为 0（即自身数字为 0），则向四周扩展直至非 0 无雷方块；

胜利规则：

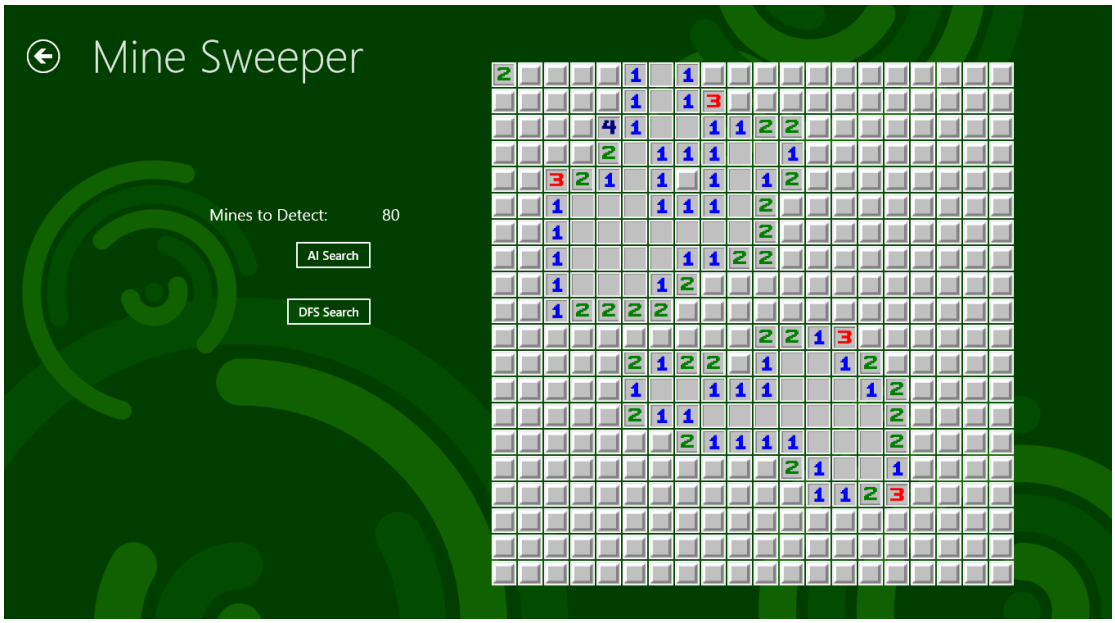
- 将所有无雷方块点下，并且标出所有雷方块（右击一次，变为有小旗的状态）

失败规则：

[1] 本文中均以方块或者 MineWindow 来表示游戏中的每一个单元格。

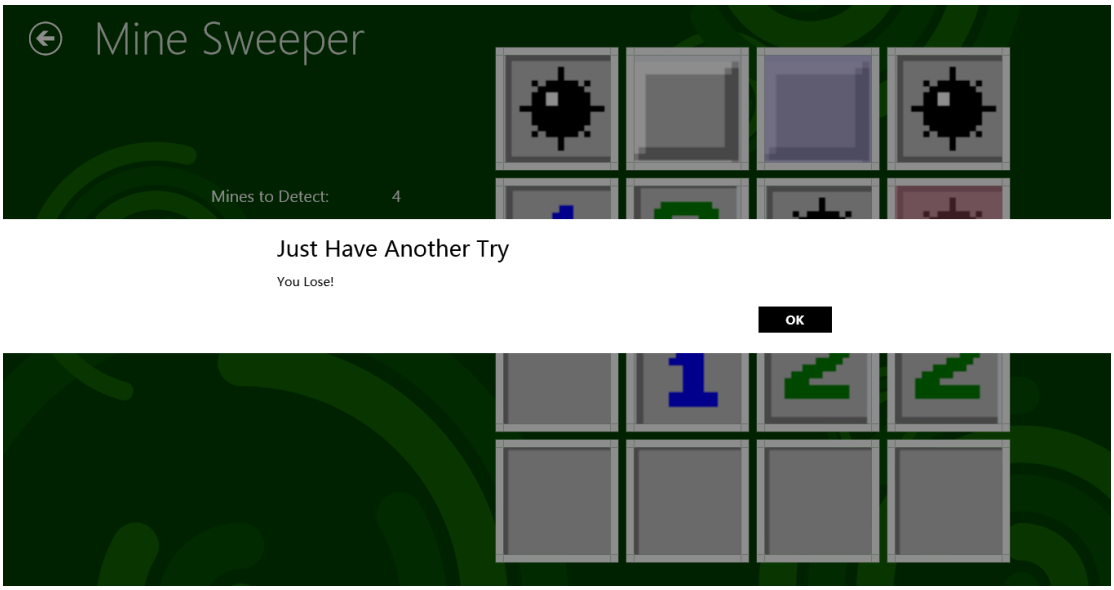
- 点中有雷方块；

正常游戏界面如图：

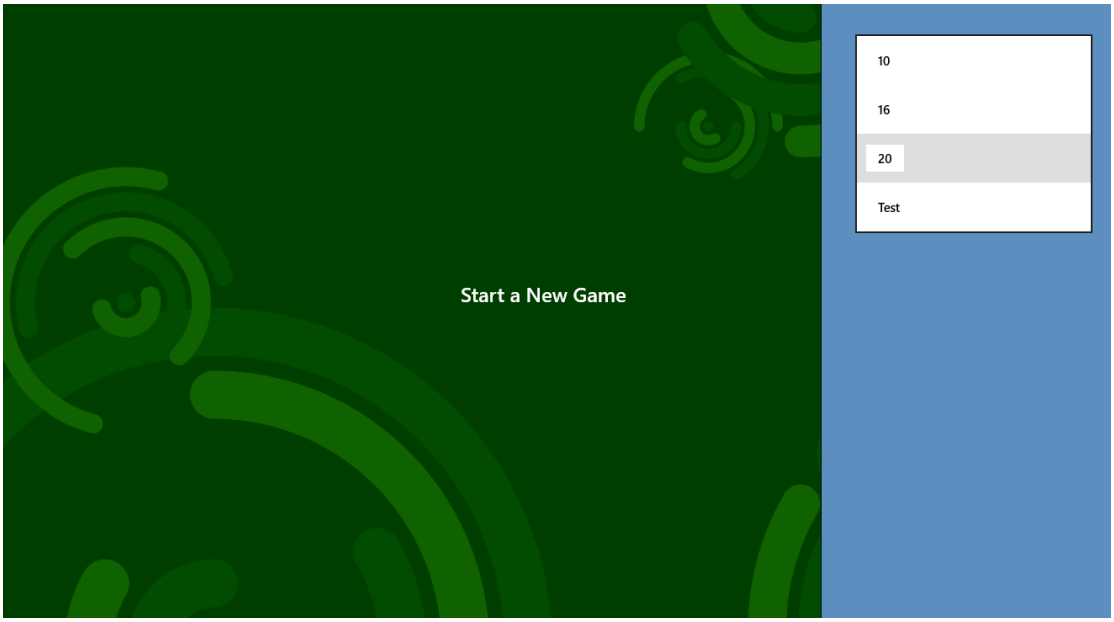


胜利界面如图（出现弹出框）：

失败界面如图（出现相同的弹出框，并且将所有的雷标识出来）：



侧边选项栏：



3. 高级功能

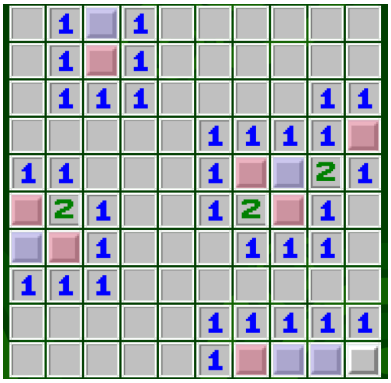
本项目最重要的创新在于，希望实现一个能够与用户能够进行很好的互动的扫雷智能体。该智能体的感知区域与用户可见的区域是相同的，因而该智能体的所有推断均是由玩家可推出的结果，因而可用于指导玩家提升思维能力与游戏水平。

基于不同层次玩家的需要，共编写了简单与高级两个智能体，通过点击屏幕左边的两个智能体按钮进行调用。

调用按钮示意图（”AI Search”表示简单智能体，“DFS Search” 表示高级智能体）：

简单智能体通过大多数初级玩家掌握的两个最基本的扫雷规则，给出用户提示。提示的具体内容包括一定为雷的区域和一定为空的区域。其中，前者用红色标识，后者用深蓝色标识。意图在于对刚入手扫雷的玩家进行指导，使其尽快熟悉游戏最基本的玩法。同时其推理结果也可以让初级玩家作为判定自己是否误判的依据。

简单智能体提示样例（其中红色一定为雷，深蓝色一定为空）：

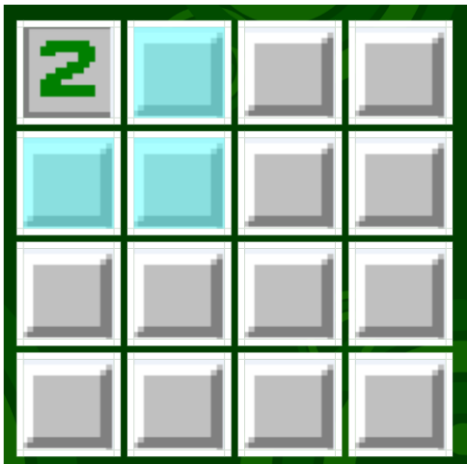


但是，在某些情况下利用初级逻辑的初级智能体是无法给出确定判断的（详细情况举例见**智能体原理**章节），相应地在那些情况下初级玩家也无法独立地仅通过初级方法找出下一步的确定方法。而在这种情况下，就需要高级智能体给出判断，并对玩家予以指导。

高级智能体可以通过更完备的算法，得到通过已知信息得到的所有可以确

定的为空的方块，并且给玩家以显示（仍用深蓝色显示）。同时，在当前已知信息无法推断出哪些方块一定为空时，则给出最优解（可能性最高为空的一系列方块的坐标），并且在图上予以显示（用天蓝色显示）。

高级智能体提示样例（显示的是无确定解时最高可能性为空的方格）：



当然，该智能体也能够全自动地完成扫雷功能，只需要对两个智能体的判断进行鼠标点击事件的模拟即可。但是成功率会有波动，因为众所周知扫雷初始值对后面问题结果的影响很大。

二． 程序实现平台

1. .Net Framework 4.5

.NET Framework4.5 是微软发行于 2012 年 8 月 16 日的平台无关性的网络开发平台，支持生成和运行下一代应用程序和 Web 服务的内部 Windows 组件。其关键组件为公共语言运行时(CLR)和.NET Framework 类库（包括 ADO .NET、ASP .NET、Windows 窗体和 Windows Presentation Foundation(WPF)和 Windows Workflow Foundation (WF)），并且提供了托管执行环境、简化的开发和部署以及与各种编程语言的集成。

.NET for Windows Store apps 是.NET Framework 4.5 的子集，并且可通过使

用 C# 或 Visual Basic，在该框架的基础上生成 Windows 的 Metro 风格应用程序。

2. 编程语言： XAML 以及 C#

XAML 是 eXtensible Application Markup Language 的英文缩写，相应的中文名称为可扩展应用程序标记语言，它是微软公司为构建应用程序用户界面而创建的一种新的描述性语言。XAML 提供了一种便于扩展和定位的语法来定义和程序逻辑分离的用户界面，而这种实现方式为复杂程序的编制、调试以及分工提供了极大的便利。XAML 是一种解析性的语言，尽管它也可以被编译。它的优点是简化编程式上的用户创建过程，应用时要添加代码等。

C#是微软公司发布的一种面向对象的、运行于.NET Framework 之上的高级程序设计语言。它是一种安全的、稳定的、简单的、优雅的，由 C 和 C++衍生出来的面向对象的编程语言，它在继承 C 和 C++强大功能的同时去掉了一些它们的复杂特性。

可以说，C#是微软.Net windows 网络时代框架的真正主角，因此利用 C#与 XAML 一起合作 ,共同完成程序逻辑与界面的编制 ,可以说是开发 Windows 应用的最佳选择。

3. IDE: Visual Studio 2012

VS2012 针对 Windows Store Application 的开发进行了特别的优化，提供了很多便利的功能 ,例如 :界面设计的集成(利用 Blend) ,各种类库的集成(包括.Net Framework 4.5)以及方便的调试功能和强大的单元体测试功能。因此，在编制代码的时候，选择利用 VS2012 作为开发环境。

三． 程序结构

1. Model-View-ViewModel (MVVM)模式

整体项目将采用分层架构来实现功能,即 Model-View-ViewModel (MVVM)模式。具体来说，就是将界面显示、内在逻辑以及人机互动这三块分成相对独立的三个部分，这样既方便了进行大项目时的分工，也对当项目变得复杂时的调试与分块改进提供了可能性与便利，同时也使程序有了重用或者重写的可能性。

就语言方面的区分来说，View 主要是由 XAML 语言编写，而其余两个设计函数调用以及事件处理的部分则由 C#编写。

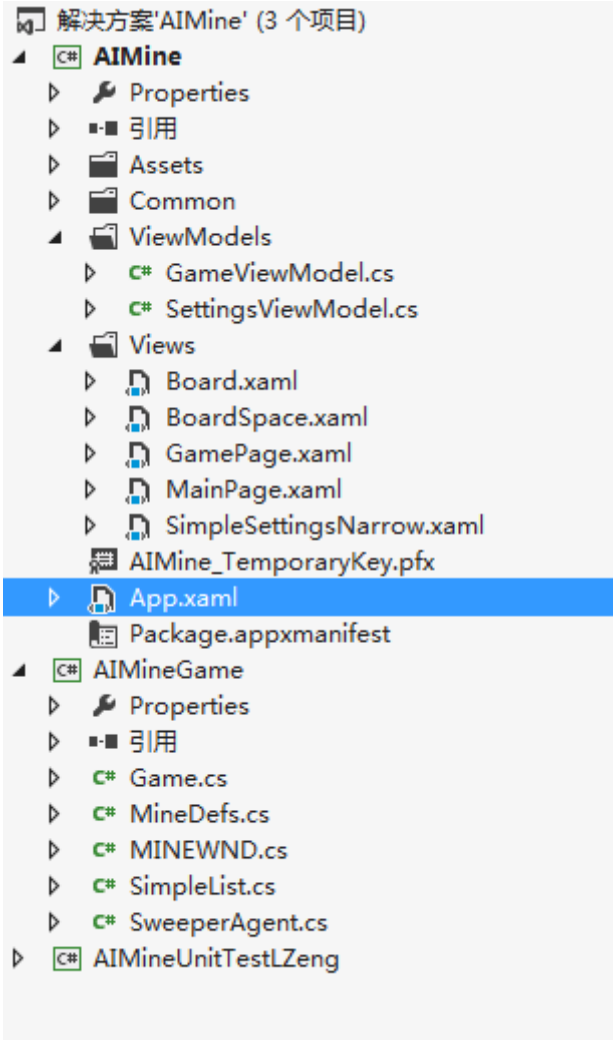
2. 三个工程

1) 各工程概述

整个程序主要由 AIMine、AIMineGame 以及 AIMineUnitTest 三个项目组成，很好地进行了程序不同功能的模块化与分治。

其中，AIMine 负责程序的界面以及界面事件的检测，即 MVVM 中的 View 以及 ViewModel 部分。AIMineGame 主要负责整个游戏的逻辑（包括游戏初始化，点击事件的处理，输赢的判断以及智能体的代码等等）。最后一个工程 AIMineUnitTestZeng 则是一个单元测试工程，用于在编写整个程序时进行调试。除非运行“单元体测试”功能，否则编译程序时不会编译最后一个工程，因而并不会影响游戏功能的发挥与游戏运行时间。

各工程以及文件、类包含关系如图所示：



2) 程序运行流程简述

整个程序的入口在上图高亮显示的 App.xaml 中。

当程序开始时，会进入 App.xaml 并利用其中的参数新建 SettingsViewModel 这个类的一个实例。同时，程序进入开始界面（由 MainPage.xaml 定义）。

当点击”Start a new game”的时候，程序会利用 SettingsViewModel 中的行数、列数以及雷数三个参数新建一个 Game 类以及 GamePage 类的实例，分别负责游戏逻辑以及游戏界面。之后，界面进入游戏界面。

GamePage 的显示以及事件的检测主要由负责（View），而事件的处

理则在 GameViewModel.cs 负责，并且对于游戏中各个方块的影响和智能体判断的调用，都是在 GameViewModel.cs 中进行处理。即，GamePage 负责界面，Game 负责逻辑，GameViewModels 是两者间的桥梁。

进入游戏界面后，每次事件的发生都由如下流程进行解决：GamePage->GameViewModel->Game，直至游戏结束（胜利、失败或者退出）。

若需要重新进行游戏，点击左上角的返回按钮之后再点击”Start a new game”即可。此时会重新建立 Game 以及 GamePage 实例，并且重新开始另一盘游戏。程序通过侧边选项改变选项的时候，改变的是 SettingsViewModel 中的数值，每一次进入游戏页面，GamePage 通过 SettingsViewModel 中的数值初始化。

3) 重点介绍 AIMineGame 结构

AIMineGame 工程下共有 5 个.cs 文件，其中四个为类，一个为定义性文件。

AIMineGame 类图大致如下：

其中，Game 类是整个游戏的类，每一盘游戏其实就是一个 Game 类的实体。

MINEWND 是每个方块的类，在 Game 类新建的时候，会同时新建

[X × Y]个 MINEWND 的实例，作为 Game 实例的属性。

MINEWND 类定义如下（为了简化，已经去除跟界面有关的绑定代码部分）：

```
public    class MINEWND: INotifyPropertyChanged
{
    public int uRow { get; set; }              //所在雷区二维数组的行
    public int uCol { get; set; }       //所在雷区二位数组的列
    public Estimation uEstimation; //每个方块的预测值
    private State uState; //每个方块当前状态
    public Attrib uAttrib { get; set; }   //方块属性
    public State uOldState { get; set; }//历史状态
}
```

SweeperAgent 则是智能体的类，智能体的所有算法均在该类中实现。与 MINEWND 类相同，Game 的实体新建时，会有一个 SweeperAgent 的实体作为其属性同时被新建。

SimpleList 是自定义的链表类型数据结构，可以将任意长度的整型数组作为元素，并且重写了 Equals 的算法，便于程序中进行数组比较。该类型在 SweeperAgent 的函数中多有使用。

在 AIMineGame 工程中还有一个 MineDef.cs 的文件，其中是定义了各种枚举类型，用以表示游戏以及每个方块的各种状态。

各结构体及状态列举如下（每个属性的功能及意义见每行代码绿色注释，下同）：

```
public enum State //方块状态
{
    /// <summary>
    /// Empty  至  Num8均为按下去之后的状态
    ///  右击事件转换顺序为  Normal -> Flag -> Dicey
    /// </summary>
    Normal = 16, //正常，即初始什么都不知道的状态
    Pressed = 15, //  被按下
    Flag = 14,    //标志为0
```

```
        Dicey = 13, //unknown status 0，带问号的状态
        Blast = 12, //爆炸状态，游戏结束后才会出现
        Error = 11, //  错误状态，游戏结束后才出现
        Mine = 10, //雷状态，被标志出来为雷，游戏结束后才出现
        Dicey_down = 9, // unknown status 1
        Num8 = 8, //周围有8雷
        Num7 = 7,
        Num6 = 6,
        Num5 = 5,
        Num4 = 4,
        Num3 = 3,
        Num2 = 2,
        Num1 = 1,
        Empty = 0, //周围无雷
    }

    public enum Estimation //智能体预测状态
    {
        None, //未预测，绑定无色
        Mine, //预测为雷，绑定红色
        Empty,//预测为空，绑定深蓝色
        TempMine,//假设为雷，未绑定
        TempEmpty,//假设为空，未绑定
        HighestEmpty //最高可能性为空，绑定天蓝色
    }
    public enum Attrib //方块属性，本质，游戏一开始后不会变
    {
        Empty, //  空
        Mine //  雷
    }
    public enum GameState //游戏状态
    {
        Normal,    //正常
        Victory, //  胜利
        Lose //失败
    }
```

每个方块显示的是数字还是小旗或者是问号都是与其uState这个属性进行绑定。其中，每个方块的uEstimation属性用于存放智能体对该方块进行的各种推断结果，并且与每个方块的色调进行绑定（具体绑定颜色见MineDef.cs中注释）。uState以及uEstimation所有取值均在之前的MineDef文件中有定义。

四．智能体原理

1. 简单智能体

1) 智能体运算步骤：

a) 首先寻找所有为数字的方块，放入 KB 中；

b) 寻找所有与数字方块相邻的未被定性的方格，放入 Requested 中；

c) 对每个 KB 中的方块进行如下判断，均只需要遍历**一次**：

i. **推理规则一：**

（周围的 Normal 状态方格的数目==自身的 Number ）？ ，若是: 周围所有的 Normal 状态的方格均为雷（即相应方块 uEstimation=Estimation.Mine）;

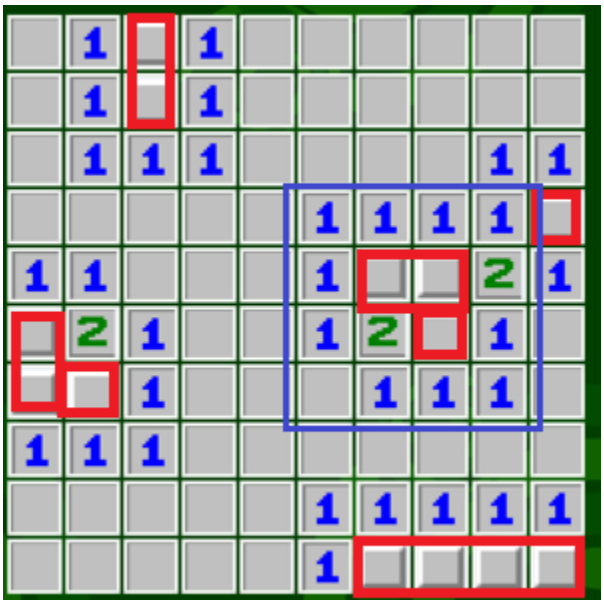
ii. **推理规则二：**

（周围的雷数（通过第一个判断得出来） == 自身的 Number ）？ ，若是: 周围所有未被预测为雷的都是 Empty（即相应方块 uEstimation=Estimation.Empty）;

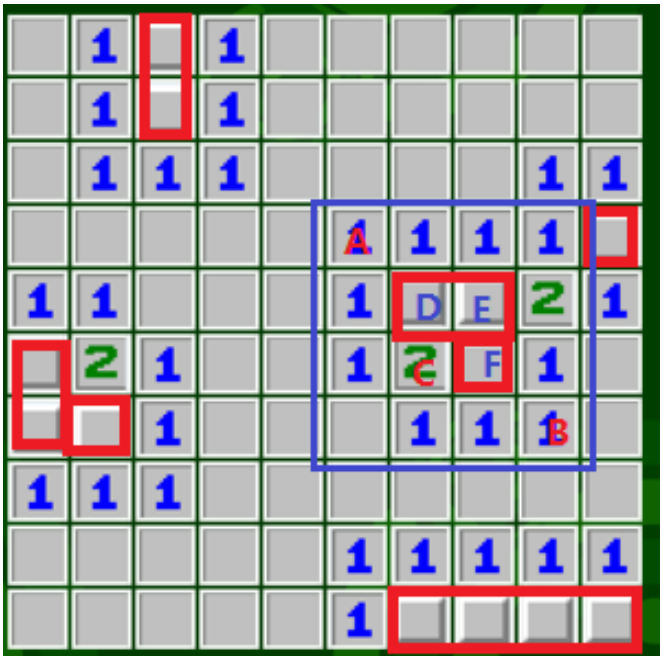
2) 运算示例：

a) 找出 KB 以及 Requested;

如图，红色方框圈出的方块即为 Requested 中存放的方块集合；而所有带数字的方块则都存于 KB 中：



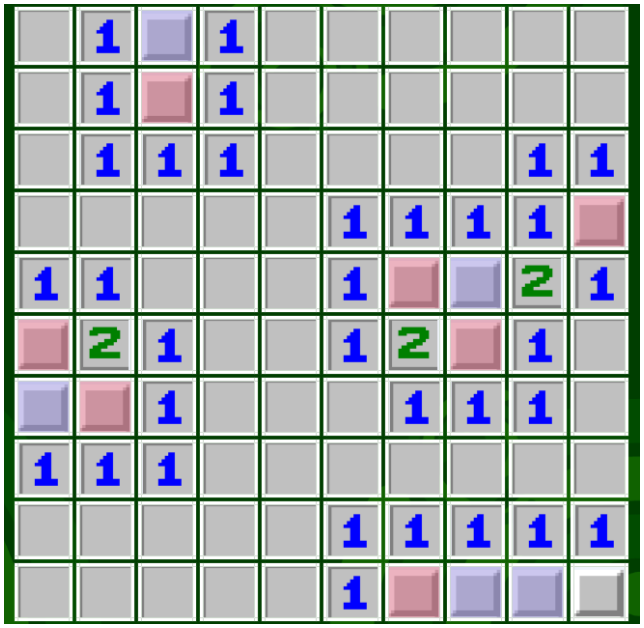
b) 针对蓝色方框中的区域进行判断：



首先，应用推理规则一。对于 A、B 两个方块，周围的未知方格数目=1，而且自身周围应该有的雷的个数=1，两者相等。因此周围的两个方块(D、F)都应该是雷。

其次，应用推理规则二。对于 C 方块，因为在之前的判断中，已经判断 D 和 F 都是雷，因此 C 周围判断的雷数=2=自身周围应该有的雷数。因而，周围未知的方块（E）应该是空。

智能体判断后，结果如下（红色表示判断为雷，深蓝色表示判断为空）：



2. 高级智能体

1）属性定义：

由于高级智能体有可能找不到确定的解，因而将其预测属性暂时定义为 Estimation.TempMine 或者 Estimation.TempEmpty。与之前的 Estimation.Mine 以及 Estimation.Empty 不同的是，TempMine 和 TempEmpty 是两种不确定的状态，只是智能体在进行逻辑判断时所作的假设，因而不能作为推理依据。而 Mine 以及 Empty 则表示，智能体可以 100%断定该方块为雷或者是空，因而可以作为智能体后续逻辑推断的依据。

2）智能体运算步骤：

a) 首先将 Requested 中可以由简单智能体推断出来的方块剔除；

b) 对剩余的方块按顺序进行深度优先搜索，将**所有合理**的情况存放

入 TemporaryMines 中；**注意：因为要搜索出所有的结果，并不是搜索到一个结果就返回。**

c) 再对每一个格子进行统计所有结果中可能为空的次数。即若{i,j}在 TemporaryMines 中有 15 次的结果为 Empty，则其对应的预测值为 15；

d) 寻找预测值最高的所有格子，输出属性为 Estimation.TempEmtpy（呈现天蓝色）;**特别的，如果在所有可能性中预测值均为 Empty，则智能体能够完全确定其为空。输出属性也要变为 Estimatin.Empty（呈现深蓝色）；**

3）深度优先搜索实现：

- 利用递归算法进行实现；
- 递归深度为 Requested 中剩余节点的个数；
- 递归返回条件为当前深度达到了递归总深度；
- 每次递归深入一层，将该层对应的方块赋值，并进行判断，如果当前的假设符合要求，则深入递归，不符合要求，则返回上一层；当到达最后一层时，如果所有假设都符合要求，则将这一假设加入 TemporaryMines 中；

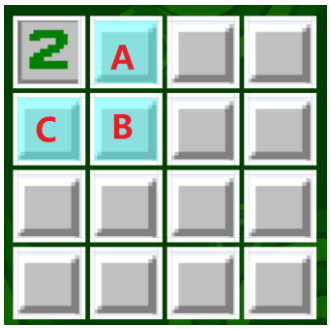
递归实现核心代码如下：

```
DFSMine( int CurrentDepth, int MaxDepth, List<SimpleList> TemporaryMines)
{
    int[] a = (int[]) Requested[CurrentDepth];
    int x = a[0];    int y = a[1];
    if (CurrentDepth < MaxDepth - 1)
        {
            //假设这一个节点为空TempEmpty
            m_pMines[x][y].uEstimation = Estimation.TempEmpty;
```

```
                //若这个假设满足，则继续深入，否则返回
                if (isLegitimateWithAssumption(x, y)) DFSMine(cd, MaxDepth++, TemporaryMines);
                //假设次节点为雷，TempMine
                m_pMines[x][y].uEstimation = Estimation.TempMine;
              //若这个假设满足，则继续深入，否则返回
                if (isLegitimateWithAssumption(x, y)) DFSMine(cd, MaxDepth++, TemporaryMines);
            }
            else if (CurrentDepth == MaxDepth - 1)//达到最后一层递归
            {
                m_pMines[x][y].uEstimation = Estimation.TempEmpty;
                if (isLegitimateWithAssumption(x, y))
                    AddToTemporaryMines(TemporaryMines);
                //将当前结果加入TemporaryMines中
                m_pMines[x][y].uEstimation = Estimation.TempMine;
                if (isLegitimateWithAssumption(x, y))
                    AddToTemporaryMines(TemporaryMines);
                //将当前结果加入TemporaryMines中
            //每次不管是成功还是失败退出函数，均要还原。
            m_pMines[x][y].uEstimation = Estimation.None;
        }
}
```

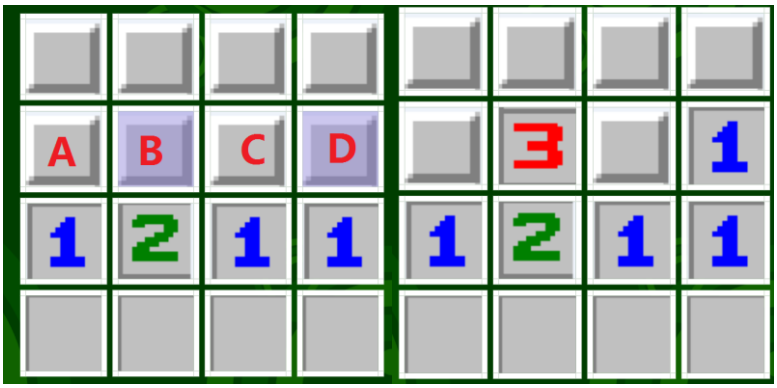
4）运算示例：

a) 无法得到确定解的情况：



若问题如上图所示，在左边角得到了一个数字方块2，则显然得不出完全正确的判断。但是可以判断，共有$C_3^2 = 3$中可能性，而所有可能性中，ABC三个方块均只有一种可能性为空，因而三者并列被列出为最有可能为空的方块。

b) 解出简单智能体无法解决的确定解得情况：



若需要解决上图左图中的情况。显然，该情况无法通过简单的逻辑智能体得出确定的解。但是通过高级智能体，对周围的4个待定的格子进行搜索，并比较结果，可以得出图二内标深蓝色的方块在所有的可能结果中均为空，则可以断定，该方块一定为空。

为了形象地表示智能体的推理过程，列表说明。四个待定方块分别按图一中所示进行标号（1代表是雷，0反之，NA表示该情况无需再考虑，因为之前的方块结果已经不满足要求，相当于提前剪枝）。若每个方块均有2种可能性，则表中应该共列出16种可能性，可是由于提前剪枝，表中列出的小于16。

A	**B**	**C**	**D**	**Satisfied**
0	0	NA	NA	NA
0	1	0	NA	NA
0	1	1	NA	NA
1	0	0	NA	NA
1	**0**	**1**	**0**	**True**
1	0	1	1	False
1	1	NA	NA	NA

可见，唯有一种情况下符合条件，则TemporaryMines中只会存储一种条件，则显然B、D两点的预测值=TemporaryMines中的所有可能值，因此被判为空（显示为深蓝色）。

图三则是点下后的状态，可见高级智能体预测准确。

3. 数据结构

KB 的类型是 SimpleList（自定义结构，见**重点介绍 AIMineGame 结构**）。其中存放着许多三元素整形数组$\{i, j, k\}$(int[3]),分别表示该方块的横纵坐标，以及方块是否为雷。

Requested 与 KB 类似，不同的是其元素为二元整型数组$\{i, j\}$，因为记录的都是未知方块的信息，所以不需要第三个元素。

TemporaryMines 的类型为 List<SimpleList>，其中的每个元素 TemporaryMine[2]都表示一个可能的预测结果。由于不确定可能预测结果的数目，即 TemporaryMines 的大小不确定，因而使用动态数组 List。每个 TemporaryMine 都是一个 SimpleList，其中存放着 Requested.Count 个数（即需要高级智能体预测的雷的总数）的需要预测的雷的信息，雷的信息用一个三元素整型数组$\{i, j, k\}$表示，i、j 表示方块的坐标，k 表示方块的预测属性。

4. 复杂度

对于简单智能体来说：

- 只是对线性数目的格子进行两次遍历；
- 空间复杂度 O(n);
- 时间复杂度 O(n);

对于高级智能体来说：

- 对一定深度的二叉树进行全搜索
- 空间复杂度 O(n)；

[2] 暂且用该符号表示 TemporaryMines 中的任意一个元素

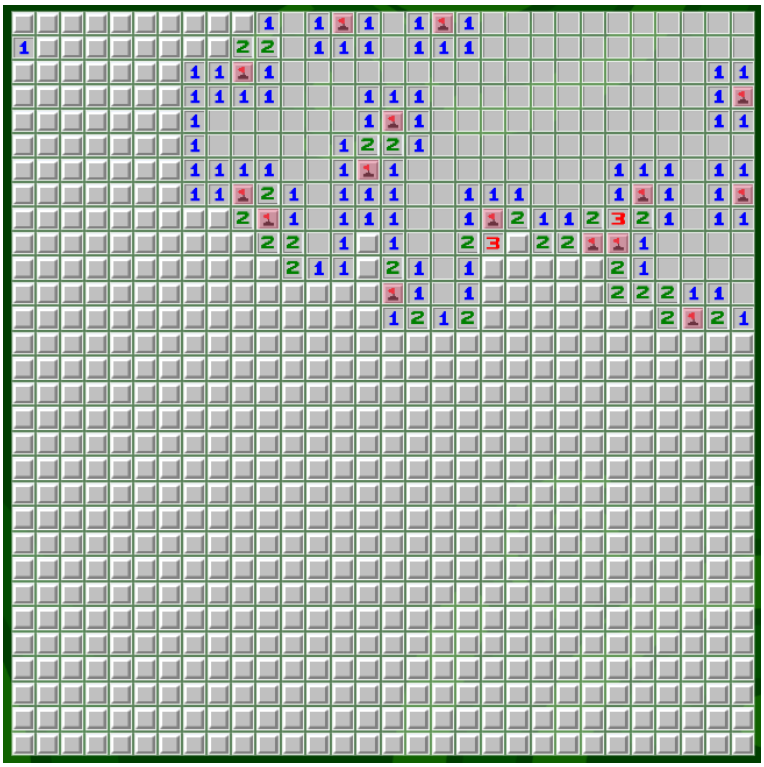
- 时间复杂度 O(2^n)；
- 但是有非常好的剪枝条件和函数，时间复杂度近似为 O($n!$)；

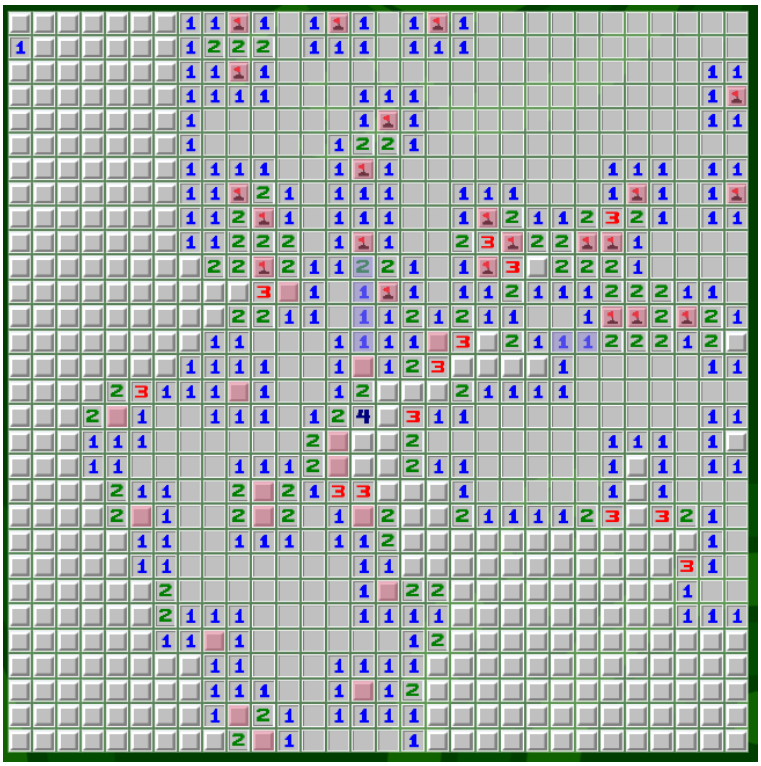
五． 测试结果

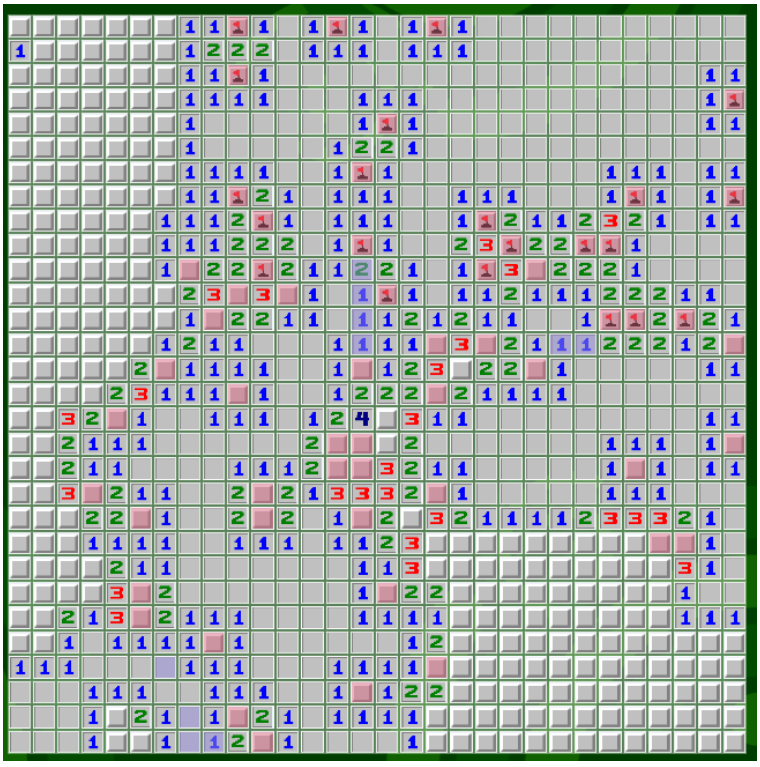
1. 运用智能体提示功能时的测试

运行参数如下：棋盘大小30×30，雷数 100。（若雷数为 200，人完成一局游戏时间太长）

以下是其中一次测试的截图：







按照智能体提示一步一步进行点击，当简单智能体无法给出解答时，利用高级智能体进行解答。最后能够非常顺畅地完成游戏。并且在所有测试中，高级智能体给出提示的时间最长为 1.5s，大多数情况下都是立即就能给出解

答。

具体测试情况如下表：

	简单智能体		高级智能体	
棋盘大小/雷数	最短时间/ms	最长时间/ms	最短时间/s	最长时间/s
$10 \times 10/8$	1	2	0.001	0.1
$16 \times 16/40$	1	5	0.001	0.5
$20 \times 20/80$	1	10	0.001	1.5
$30 \times 30/200$	1	20	0.001	2

注：

1) 以上测试结果均是结合了界面显示时间的结果，并且表中时间表示的是测试的智能体给出提示的时间；

2) 由于高级智能体会先调用简单智能体，找出无法解出的方格再调用自己的算法；但是如果不存在解不出的方格，则直接退出；因而高级智能体的最短时间和简单智能体相同；

3) 每次均将智能体给出的**所有**结果处理完（左击或右击）再让智能体进行下一次推理

2. 进行全自动扫雷时的性能测试

在测试类中编写了一个全自动扫雷的函数，大致流程如下：

1） 在四角随机进行一次左击事件；

2） 调用简单智能体；

3） 若简单智能体有结果输出，则对**所有**输出结果进行左击（对于空预测）和右击（对于雷预测）事件处理，之后转至 2）；否则转至 4）；

4） 调用高级智能体，若高级智能体有确定输出，则对所有输出进行处

理；若没有确定输出，则随机选取一个对输出进行处理；之后转至2）；

5） 不断循环，直到胜利或者失败；

以下是不同棋盘的测试结果：

	扫雷智能体		
棋盘大小/雷数	胜率	最短时间/s	最长时间/s
$10 \times 10/8$	98/100	0.5	0.8
$16 \times 16/40$	95/100	1	2
$20 \times 20/80$	90/100	5	8
$30 \times 30/200$	80/100	6	15

六． 总结与收获

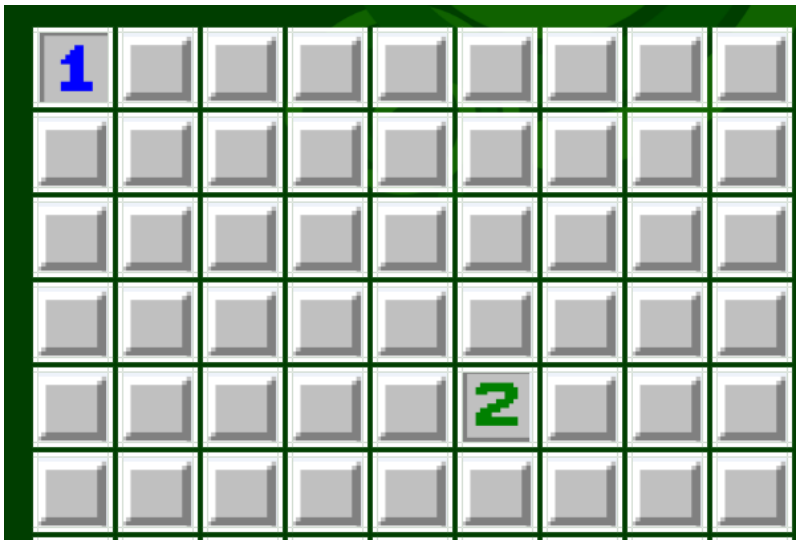
1. 项目不足：

1) 未能将范式与归结法的方法运用于智能体的扫雷，而是将过多的精力投入到了抽象游戏的逻辑构架，智能体逻辑判断规则的推导，以及规范程序的结构比如：整个程序的人机交互，以及内在逻辑与界面的关联上；

2) 未能排除一些智能体算法上的不足：

a) 完全分立的未知方块可以单独考虑：

举一个最简单的情形，如下图所示状况：



该状况下，肯定需要用高级智能体进行求解。但是，1 与 2 周围的未知方块完全独立，因而可以分开求解。若分开求解，对于 1 周围的方块有$c_3^1 = 3$种可能，则任一方块为空的概率为$^2/_3$。同理，对于 2 周围的方块，有$c_8^2 = 28$种可能，对于每一个方块，为空的概率为$c_7^2/c_8^2 = {}^3/_4$。因而最后的结果为 2 周围出现天蓝色方块。则在 RequestedMines 中共需要记录并遍历$28 + 3 = 31$种可能。

但是如果不分开处理，按照程序中现有的方法进行处理的话，虽然结果一样，但是却需要记录以及遍历28×3=84 种可能。当所需求解的方块数目尚小时，此差异不易显现。但是当问题规模变大的时候，这个差异会相当致命。

解决办法是在智能体刚开始执行时先判断出独立的方块，再分开处理即可。可以想见，因为分区数目不定，因而应该用动态数据结构。

b) 自动扫雷智能体的不足

其实对于扫雷智能体来说，可以点击的不仅仅只有 Requested 中的方块（即与数字相邻的方块）。

当然，有确定解时肯定可以去点击确定解。但是当没有确定解而

只有估计的最优解时，Requested 中的最优解有可能并没有优于其之外的其它可能为空的方格。因而，为了能够进一步提高准确度，因该将 Requested 中最优解的概率与其之外的格子为空的概率(具体计算很简单：*剩余的空格数/剩余的未知方块数* ）进行比较，并作出决断即可。

2. 收获

1) 从实际问题中抽象出逻辑表达式

扫雷问题是一个看似规则简单，其实很复杂的问题。尤其在这个项目中，为了能够让界面与用户有良好的交互，给每一个雷方块设置了很多属性，因此会让简单的逻辑表达式变得很复杂。

但是，正是在这种总结复杂表达式的过程中我锻炼了自己全方位缜密思考，以及从实际问题中抽象出逻辑表达式的能力。同时，也对“任意”“存在”这两个谓词在逻辑中的作用以及在编程中的实现有了很深刻的理解。

比如，在高级智能体搜索中，每次对新的方块进行假设都需要判断是否符合条件，而这个是否符合条件的意义是：

*对于新增加的方块周边的**任意**方块都必须满足以下条件：*

a) 周围的已经存在的雷的数目<=自身显示的数目；

b) 周围已经存在的雷的数目+周围未知方块数目>=自身显示的数目；

而代码实现时，却需要先假设新添加方块满足条件。然后对其周围所有（**任意**）方块进行检测，只要**存在**一个方块不满足上述 a)、b)条件，就说明该方块假设不满足。（具体判断代码见 *SweeperAgent.cs 下 bool*

isLegitimateWithAssumption(x,y)函数）。

可以说，这个项目让我对逻辑、代码以及现实问题间的联系和不同有了深刻的理解，并且能够有意识地在程序中运用逻辑的原理对程序结构进行思考，甚至对算法进行优化。

3. **逻辑推断的程序化**

如上所说，能够抽象出逻辑结构，和能够将逻辑结构程序化并使其符合语言规范根本就不是一回事。其中最大的问题就是如何循环以及数据的存储问题。

就像之前讨论过的 KB，Required 以及 TemporaryMines 的数据结构问题，都是经过很多次试验以及失败之后才寻找到的方法。

4. **程序分块与规范编程**

当今社会，已经没有一个人做事的程序员了。因为随着用户的需求日益提升，以及竞争的日益激烈，计算机应用领域正朝着越来越专业化、分块越来越明显的趋势发展。

而合作的基础，就是程序的分块与不同部分之间的规范与通信。在本项目中，实施了界面与逻辑完全独立的编程方式，极大地降低了调试与两个部分的编写难度。这次项目让我对大规模程序的分工与规范编程有了初步的理解，并且大幅度提升了自身调试、构思程序框架以及对各个数据结构的运用和理解能力。

最后，感谢黄老师以及赵助教一个学期以来的辛苦与努力！真的在离散数学这门课上学到了很多，收获了很多！