

---

# 扫雷 Auto & Manual MineSweep

22112020002 蔡志杰

## 目录

扫雷 Auto & Manual MineSweep ..... 1

    概述.....2

    一、底层数据结构.....2

        1.1、游戏底层数据结构分析.....2

        1.2、对底层数据的处理方式.....3

    二、扫雷功能与算法实现.....4

        2.1、随机数生成.....4

        2.2、基本扫雷操作.....4

            2.2.1、对普通网格的探索.....4

            2.2.2、标记雷.....5

            2.2.3、去除雷标.....6

        2.3、自动扫雷算法.....6

            2.3.1、随机扫雷.....6

            2.3.2、单格推断.....7

            2.3.3、多格推断.....7

            2.3.4、概率扫雷.....8

            2.3.5、自动扫雷.....9

        2.4、状态判断.....9

    三、游戏交互的实现..... 10

        3.1、交互逻辑..... 10

        3.2、终端模式的实现..... 11

        3.3、GUI 模式的实现 ..... 11

            3.3.1、GUI 库 ..... 11

            3.3.2、GUI 实现 ..... 11

    四、文件结构与说明..... 13

## 概述

本次 haskell 期末实现了一个扫雷游戏，有基于字符的终端（Terminal）版本（通过键盘输入进行交互）和基于图像的 GUI 版本（通过鼠标的点击进行交互）。两个版本底层的算法和逻辑都是一致的，只是最终呈现的形式不同。扫雷游戏除了支持用户的常规操作（选中格子和标注棋子）还支持一系列自动扫雷的操作用以辅助用户。

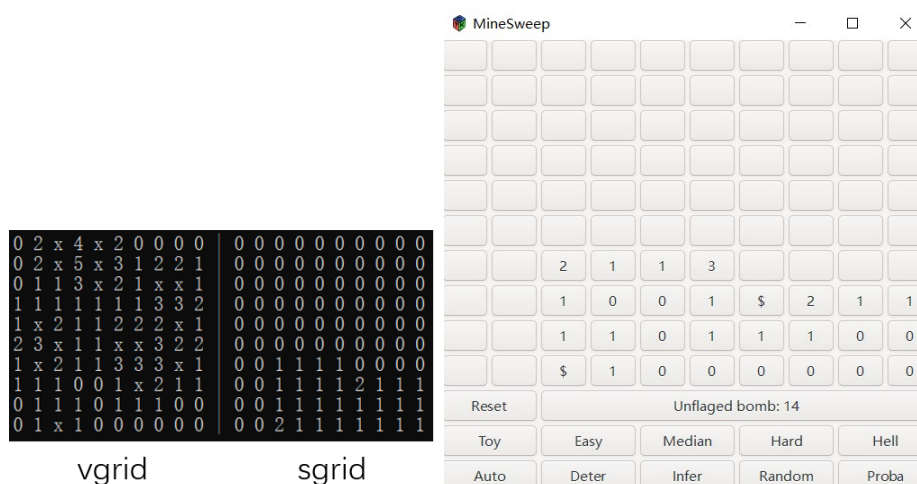
接下来将分别介绍扫雷游戏底层的数据结构实现，扫雷中各种操作效果及其算法实现，最后再介绍终端和 GUI 中游戏交互的实现。

## 一、底层数据结构

### 1.1、游戏底层数据结构分析

扫雷游戏本质上是对一个二维数组的操作，其中可以看成两个二维数组 vgrid 和 sgrid，分别存储扫雷数据层（value grid，简称 vgrid）和状态掩膜层（state grid，简称 sgrid）。vgrid 将存储扫雷中的数据，这在初始随机生成雷的位置后便直接确定，不会在游戏过程中发生变化，其中每一个格子中的数量对于这格子周围 8 个格子中包含的雷的总数，如果格子本身为雷则数值为-1。sgrid 将存储游戏中每个格子的状态，其中包含未访问（nonVisited），已访问（Visited）和炸弹标记（Flaged），sgrid 将会随着游戏的进行不断变化。

下图给出了一个 vgrid 和 sgrid 的例子（vgrid 中的 'x' 为雷）：



---

## 1.2、对底层数据的处理方式

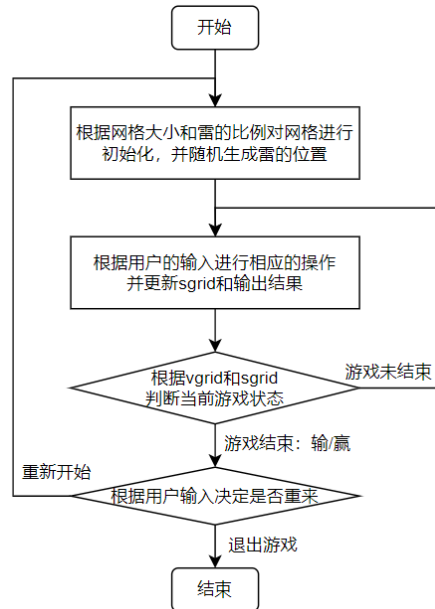
为了方便对二维数组的访问，减少参数的传递，同时方便后续的一些算法能方便地套用一些高阶函数，将二维矩阵中每一个格点采用一维索引进行访问。其中一维索引值与将二维数组进行 `concat` 后得到的一位数组的索引值相对应。此外还实现了二维索引与一维索引相互转换的函数（`coor2index`，`index2coor`）其中处理了访问越界的问题。

对二维数组的元素的访问采用 `!!` 操作符（`getElementById`）。对二维数组某一元素的更新则较为麻烦，采用 `Data.Sequence` 中的 `fromList` 函数（`fromList :: [a] -> Seq a`）将列表转换成序列，再使用 `update` 函数（`update :: Int -> a -> Seq a -> Seq a`）对序列中的某一元素进行更新，最后再通过 `toList` 函数（`toList :: Foldable t => t a -> [a]`）将序列转换回列表，更新列表元素的函数为 `updateElement`。

此外为了后续操作的方便还实现了一些基础函数，如 `getSurroundingId` 获取一个网格周围网格索引值，`countSurroundingBomb` 获取网格周围的炸弹数（用于初始化 `vgrid`）以及一些判断函数和计数函数。

## 二、扫雷功能与算法实现

整个游戏运行的流程图如下：



### 2.1、随机数生成

为了方便地获得随机数，基于 `System.Random` 的接口编写了一个随机数模块（`MyRandom`）。

`getR :: RandomGen g => Int -> g -> [Int]`，将根据给定的整数 `n` 和随机种子生成 `[0, n-1]` 之间的一个随机数。（用于辅助扫雷功能）

`getRList :: RandomGen g => Int -> Int -> g -> [Int]`，将根据给定的整数 `k`, `n` 和随机种子从 `[0, n-1]` 中随机选出 `k` 个数。（用于初始化地雷的位置）

### 2.2、基本扫雷操作

#### 2.2.1、对普通网格的探索

`sweep :: [[Int]] -> [[Int]] -> Int -> [[Int]]`，将根据给定的 `vgrid` 和 `sgrid` 以及选中的网格索引值，对该网格进行探索，返回更新后的 `sgrid`。如果该网格的值大于 0（周围 8 格中有雷）或为 -1（炸弹）就只将该网格的状态从 `nonVisited` 转换成

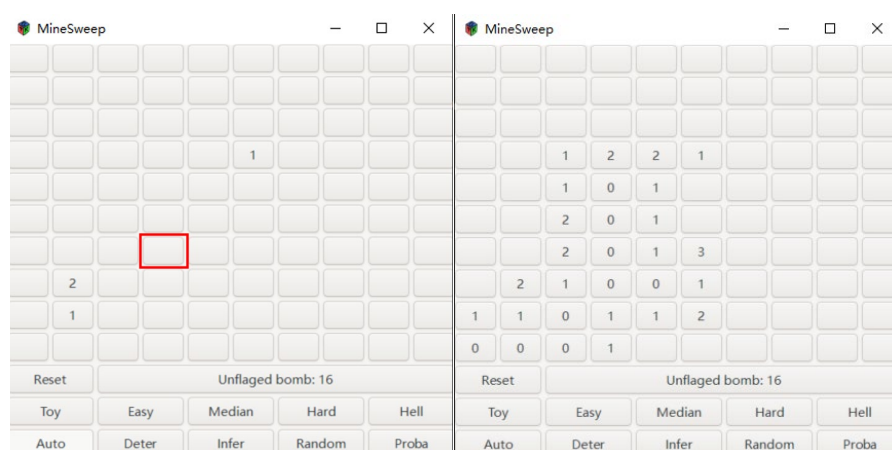
visited，否则该网格值为 0，说明周围都没有雷，那么将会递归搜索周围所有的网格。为了实现 0 值网格周围的递归搜索，通过 `bfs :: [[Int]] -> [[Int]] -> [Int] -> [[Int]]` 来完成对某一网格周围网格的广度搜索。

下图展示了两个例子（终端版和 GUI 版）

选中索引为 (10, 10) 的网格后的变化 (索引的原点在左下角)

```
$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count $: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count  
remained unmined grids : 99 remained unmined grids : 81  
total bomb/ flagged bomb : 16/0 total bomb/ flagged bomb : 16/0  
***** * *****  
* 3 ***** * 3 ***** 1 0 0 0  
***** * 2 1 0 0  
***** * 1 0 0  
***** * 2 1 1  
***** *  
***** *  
***** *  
***** *  
***** *
```

选中 (4, 4) 网格后的变化



### 2.2.2、标记雷

setFlag :: [[Int]] -> [[Int]] -> Int -> [[Int]] 将根据 vgrid 和 sgrid 以及给定的索引，若对应位置的状态为 nonVisited 就将其转换为 Flag，返回更新后的 sgrid。

将索引为 (8, 8) 的网格标为 flag ('\$'为雷标):

```

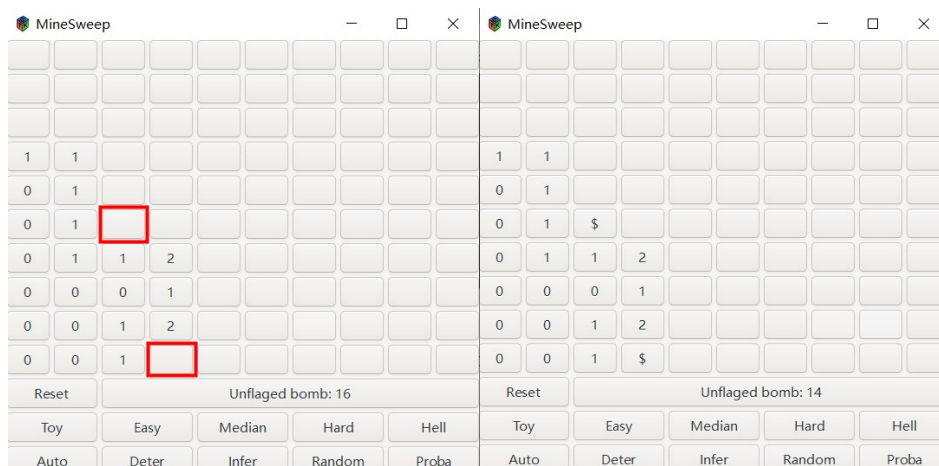
$> flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 71
total bomb/ flagged bomb : 16/0
*****
***** 3 1
***** 2 0
***** 2 1 0
***** 3 2 1 0 0
***** 1 0 0 0
***** 3 2 1 0 0
***** 3 1 0
***** 1 0
***** 1 0
*****

input operation:
[M]ine
[F]lag
[U]nflag
assistance:
[A]utomaticMine
[D]eterministicMine
[I]nferenceMine
[P]robabilisticMine
[R]andomMine
[E]xit
f
input rownumber: (1 -> 10)
8
input colnumber: (1 -> 10)

$> flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 70
total bomb/ flagged bomb : 16/1
*****
***** 3 1
***** 2 0
***** 2 1 0
***** 3 2 1 0 0
***** 1 0 0 0
***** 3 2 1 0 0
***** 3 1 0
***** 1 0
***** 1 0
*****

```

将索引为 (3, 5) (4, 1) 的网格标为 flag:



### 2.2.3、去除雷标

`unsetFlag :: [[Int]] -> [[Int]] -> Int -> [[Int]]` 将根据 `vgrid` 和 `sgrid` 以及给定的索引，若对应位置的状态为 `Flag` 就将其转换为 `nonVisited`，返回更新后的 `sgrid`。

去除索引为（8，8）的雷标：

```

input operation:
[M]ine
[F]lag
[U]nflag
assistance:
[A]utomaticMine
[D]eterministicMine
[I]nferenceMine
[P]robabilisticMine
[R]andomMine
[E]xit
input rownumber: (1 -> 10)
8
input colnumber: (1 -> 10)
8

$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 70
total bomb/ flagged bomb : 16/1
*****
***** 3 1
***** 2 0
***** 2 1 0
***** 3 2 1 0 0
***** 1 0 0 0 0
***** 3 2 1 0 0
***** 3 1 0
***** 1 0
***** 1 0

$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 71
total bomb/ flagged bomb : 16/0
*****
***** 3 1
***** 2 0
***** 2 1 0
***** 3 2 1 0 0
***** 1 0 0 0 0
***** 3 2 1 0 0
***** 3 1 0
***** 1 0
***** 1 0

```

## 2.3、自动扫雷算法

### 2.3.1、随机扫雷

`randomMine :: RandomGen g => [[Int]] -> [[Int]] -> g -> [[Int]]` 将根据 `vgrid`，`sgrid` 和随机种子，从所有 `nonVisited` 状态下的网格中随机选一个，返回更新后的 `sgrid`。

```

input operation:
[M]ine
[F]lag
[U]nflag
assistance:
[A]utomaticMine
[D]eterministicMine
[I]nferenceMine
[P]robabilisticMine
[R]andomMine
[E]xit
input rownumber: (1 -> 10)
2
input colnumber: (1 -> 10)
8

$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 100
total bomb/ flagged bomb : 16/0
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****

$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 99
total bomb/ flagged bomb : 16/0
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****

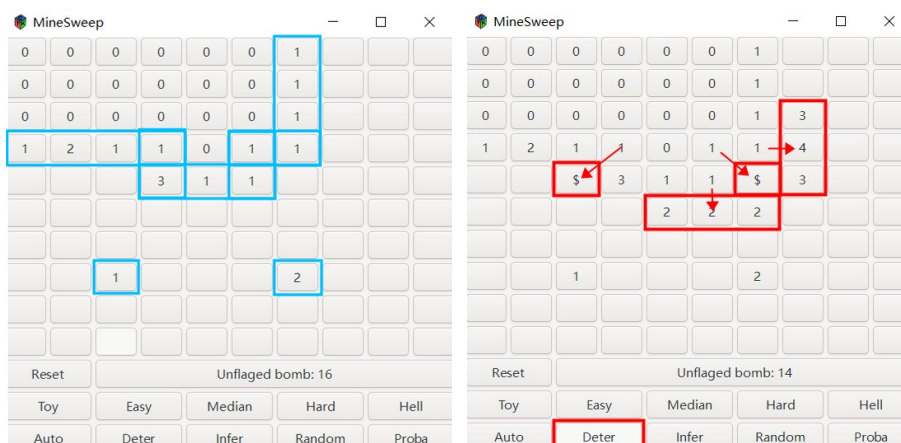
```

### 2.3.2、单格推断

`deterministicMine :: [[Int]] -> [[Int]] -> [[Int]]` 将根据 `vgrid` 和 `sgrid`，对已有格子中的每个边界网格单独进行推断，返回更新后的 `sgrid`。边界网格它们自身为 `visited` 状态，同时周围至少要有一个处于 `nonVisited` 状态的网格，下图蓝色边框展示了一个边界网格的例子，如果满足下面的条件就能确定周围 `nonVisited` 网格的状态：

#周围 `flag` == 自己的值（周围的炸弹都已确定，剩余 `nonVisited` 都可以变成 `visited`）

#周围 `nonVisited` + #周围 `flag` == 自己的值（周围剩余 `nonVisited` 网格都是炸弹，都可以转换为 `flag` 状态）



### 2.3.3、多格推断

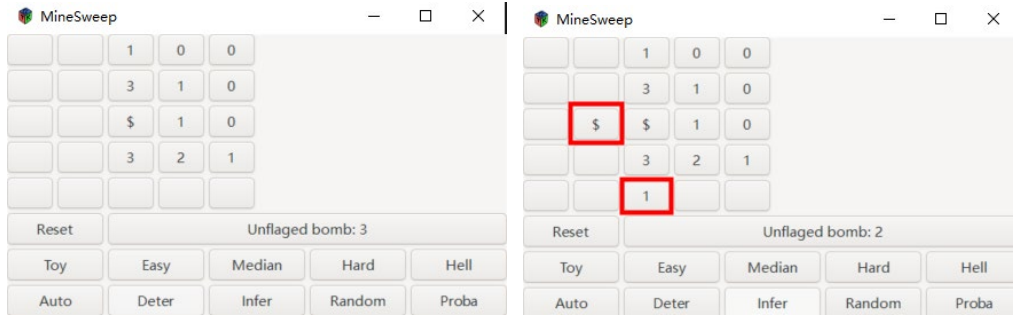
`inferenceMine :: [[Int]] -> [[Int]] -> [[Int]]` 将根据 `vgrid` 和 `sgrid`，对已有格子中边界网格进行推断，返回更新后的 `sgrid`。与单格推断不同，多格推断能结合周围几个格子的信息同时进行推断，得到单格推断得不到的信息。

首先定义轮廓网格，轮廓网格为边界网格周围处于 `nonVisited` 状态的网格。这里采用的方法为枚举所有轮廓网格的可能（是炸弹或不是炸弹，若轮廓网格有  $n$  个则有  $2^n$  种可能），然后根据枚举的可能判断这一可能是否合理，过滤掉不合理的可能，在剩余的可能中，若某个网格的状态在所有可能中都一致则将其置为该种状态（通过 `sweep` 或 `flag` 操作对状态确定的轮廓网格进行处理）。

上面提到的方法的复杂度过大，随着轮廓的大小复杂度呈指数增长，这只适用于小规模的问题，为了实现更大规模下的多格推断，采用启发式的方法。根据

分析多格推断通常只受到相邻网格的影响，因此可以每次枚举部分相邻的轮廓网格，在通过轮流切换来覆盖所有轮廓网格。具体做法如下，设置每次推测的轮廓网格数量上限 `inferenceRange::Int`（实验中设置为 10），以及两次推断中可以忽略的交叠部分 `inferenceRangeOverlap::Int`（实验中设置为 6），每次从轮廓网格列表中选一个起点，通过广度优先搜索（`bfsSurrounding`）获得与之相连前 `inferenceRange` 个轮廓网格，枚举其所有可能性，并进行推断，如果推断成功（有确定的雷或非雷网格）则结束本次多格推断，否则从轮廓列表中删除起始网格相连的前 `inferenceRangeOverlap` 个轮廓网格，并从剩余的轮廓网格中重新挑选一个网格作为源头继续前面的过程，直至有一次推断成功，或是所有轮廓网格都推断过了。（经实验，设置的参数在 30\*30 的网格下运行速度还可以接受）

下面左图中的情况使用单格推断无法得到结果，但是多格推断则能得到右边的结果：

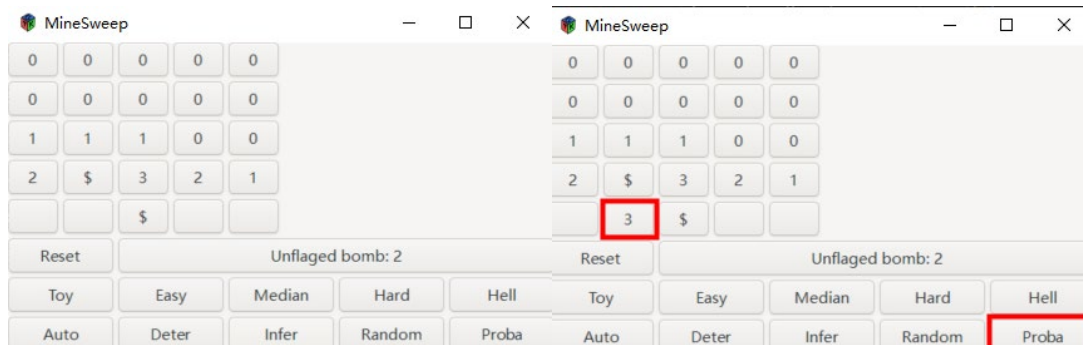


#### 2.3.4、概率扫雷

`probabilisticMine :: RandomGen g => [[Int]] -> [[Int]] -> g -> [[Int]]` 将根据 `vgrid` 和 `sgrid`，对当前的轮廓网格进行概率推断，返回更新后的 `sgrid`。与多格推断的做法类似，会先选取一段轮廓网格，尝试所有可能的状态，最后从所有可能的状态中选出非雷概率最大的网格（这里的概率就是指非雷状态在合法状态中出现的次数）进行 `sweep` 操作，如果多个网格概率相同则从中随机选一个。

经过分析概率扫雷的操作与随机扫雷大部分情况下相比失败率更大，因为概率扫雷只会考虑轮廓网格（受限于复杂度，本算法还只能考虑 `inferenceRange` 个的轮廓网格），而随机扫雷会考虑所有 `nonVisited` 网格。因此在后面综合的自动扫雷算法中遇到单格和多格都无法推断的情况下会采用随机扫雷而非概率扫雷。





### 2.3.5、自动扫雷

`automaticMine :: RandomGen g => [[Int]] -> [[Int]] -> g -> [[Int]]` 将根据 `vgrid`, `sgrid` 和随机种子, 进行综合性的自动扫雷, 返回更新后的 `sgrid`。

自动扫雷算法整合了上面几种自动扫雷的方法, 将针对不同情况采用不同的方法。策略如下:

- 首先优先采用单格推断, 复杂度中等, 且得到的结果一定正确。
- 当单格推断失败 (无法更新 `sgrid`) 则进行多格推断, 复杂度高于单格推断, 但是能够获取的信息更多。
- 当单格和多格推断都失败的时候, 进行随机扫雷, 复杂度最低, 但是有概率失败。

该自动扫雷方法能接近人的最高扫雷能力, 和人的唯一区别在于多格推断的时候为了考虑计算复杂度只能考虑部分轮廓网格, 而不是全部。但是由于推理过程还是往往局限于临近的一些单元, 故该自动扫雷方法已经能达到和人几乎一样的水平了。

## 2.4、状态判断

在游戏过程中每步操作后都需要对当前状态进行判断, 判断当前状态是输、赢还是继续, 同时需要进行一些游戏状态的统计来辅助玩家 (如剩余炸弹数, 剩余没炸弹的网格数量) 这些都由函数 `evalState :: [[Int]] -> [[Int]] -> IO ()` 实现。其中又包含 `evalWin` 和 `evalLoss` 两个函数来判断是否输赢, 输的标准是存在 `visited` 状态的网格数值为 -1 (炸弹), 赢的判断是不存在 `nonVisited` 状态的非炸弹网格 (数值大于等于 0) 且没有标记错误的 `flag` 且没有输。

---

## 三、游戏交互的实现

### 3.1、交互逻辑

不论是终端交互还是图形界面交互他们底层的交互逻辑都是一致的。

再终端交互游戏开始后会首先要求用户输入扫雷网格的大小，然后开始游戏。图形界面中则是先默认 10x10 的网格大小，可以通过更改游戏难易程度的按键（Toy, Easy, Median, Hard, Hell）来改变网格大小。

确定好网格大小后即可开始游戏，游戏过程中都有以下几种操作（不论是以点击按钮的新式还是输入操作名称和操作作用坐标的方式）：

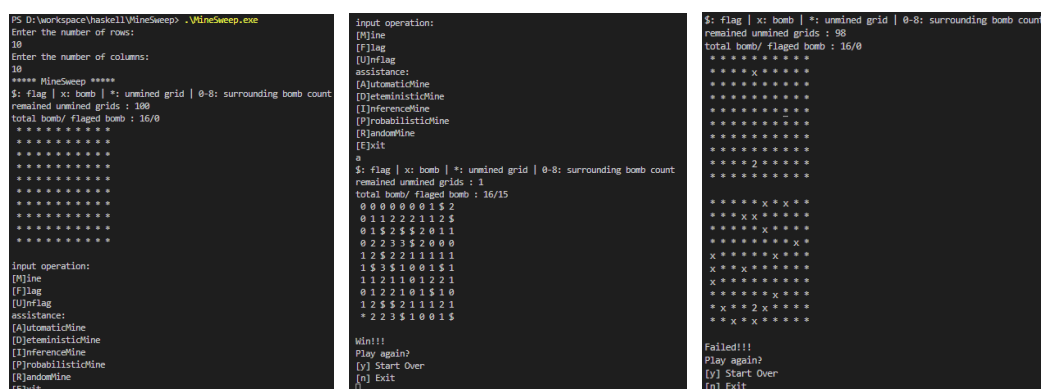
- **Mine**: 普通的扫雷操作（终端中需要先输入 M，再分别输入扫雷的坐标；图形界面中需要鼠标左键点击相应的格子两下）
- **Flag**: 标记炸弹（终端中需要先输入 F，再分别输入标记的坐标；图形界面中需要鼠标左键点击相应的格子一下）
- **Unflag**: 去除炸弹标记（终端中需要先输入 F，再分别输入标记的坐标；图形界面中即为点击以及标记为 flag 的格子，即可去除 flag 标记，同时会访问该格子）
- **AutomaticMine**: 自动扫雷（终端中输入 A；图形界面中点击 Auto 按钮）
- **DeterministicMine**: 单格推理扫雷（终端中输入 D；图形界面中点击 Deter 按钮）。其中注意当对一个以及访问过的格子（已经 Mine 操作过的网格），再次进行 Mine 操作会触发其单格推理的过程，自动确定其周围的情况并进行状态更新
- **InferenceMine**: 多格推理扫雷（终端中输入 I；图形界面中点击 Infer 按钮）。
- **ProbabilisticMine**: 概率扫雷（终端中输入 P；图形界面中点击 Proba 按钮）。
- **RandomMine**: 随机扫雷（终端中输入 R；图形界面中点击 Random 按钮）。

游戏结束后（输或赢）重新开始（终端输入 y，图形界面点击 Reset），退出游戏（终端输入 n，图形界面点击右上角的 x）。

## 3.2、终端模式的实现

为了实现终端版的扫雷游戏，首先定义显示的符号：\$为标记的 flag，\*为 nonVisited 网格，x 为炸弹，0-8 为网格周围的炸弹数。

为了只显示 visited 和 flag 状态的网格，其余显示'\*'，采用函数 `maskGrid :: [[Int]] -> [[Int]] -> [[Int]]` 将根据 `vgrid` 和 `sgrid` 生成经过掩膜处理后的数据，再通过自定义的 `int2char :: Int -> Char` 将整型转换成字符类型便于显示。最后通过 `dispGrid :: [[Int]] -> [[Int]] -> IO()` 函数将其输出到终端中。下图为一个例子：



```
PS D:\workspace\haskell\MineSweep> .\MineSweep.exe
Enter the number of rows:
10
Enter the number of columns:
10
**** MineSweep ****
$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 100
total bomb/ flagged bomb : 16/0
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
input operation:
[M]ine
[F]lag
[U]nflag
assistance:
[A]utomaticMine
[D]eterministicMine
[I]nferencemine
[P]robabilisticMine
[R]andomMine
[E]xit
0
$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 1
total bomb/ flagged bomb : 16/15
0 0 0 0 0 0 1 5 2
0 1 1 2 2 2 1 1 2 5
0 1 5 2 5 5 2 0 1 1
0 2 2 3 3 5 2 0 0 0
1 2 5 2 2 1 1 1 1 1
1 5 5 1 0 0 1 5 1
1 1 2 1 1 0 1 2 2 1
0 1 2 2 1 0 1 5 1 0
1 2 5 5 2 1 1 1 2 1
* 2 2 3 5 1 0 0 1 5
Win!!!
Play again?
[y] Start Over
[n] Exit
[]

$: flag | x: bomb | *: unmined grid | 0-8: surrounding bomb count
remained unmined grids : 98
total bomb/ flagged bomb : 16/0
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
Failed!!!
Play again?
[y] Start Over
[n] Exit
[]
```

## 3.3、GUI 模式的实现

### 3.3.1、GUI 库

使用 `gi-gtk` 库进行 GUI 界面的实现。相关的文档和配置方法可见如下链接：

<https://hackage.haskell.org/package/gi-gtk>

<https://github.com/haskell-gi/haskell-gi>

[Using haskell gi in Windows · haskell-gi/haskell-gi Wiki \(github.com\)](https://haskell-gi.github.io/haskell-gi/Wiki.html)

### 3.3.2、GUI 实现

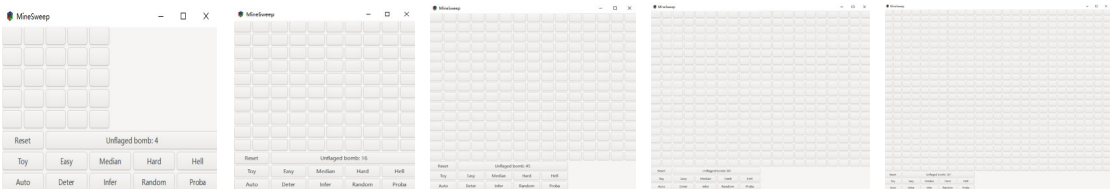
将扫雷的网格用 `Gtk.Button` 类实现。将二维矩阵的 `Gtk.Button`，绑定在一个 `Gtk.Grid` 上，通过 `Grid` 对按键进行管理。扫雷网格中按键显示符号的意义与终端版本一致：\$对应 flag，0-8 对应周围炸弹数量，空白对应 nonVisited 状态。

其中按键又分为几类：

一类是扫雷网格中的普通按键，其回调函数为 `gridButtonCallback`，会根据该

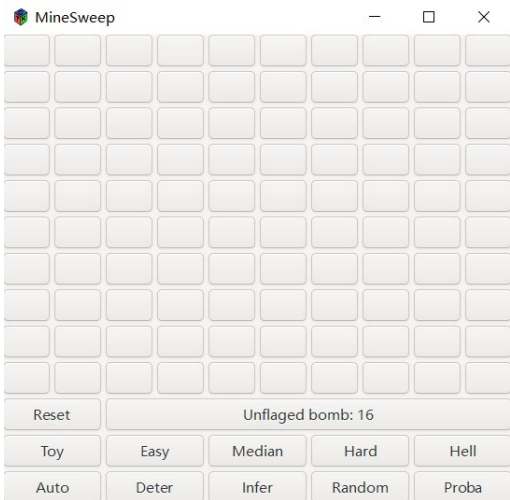
按键当前的状态更新整个地雷网格的按键状态。由于 `gi-gtk` 中的 `Button` 只有左键的属性，没有办法通过邮件来标注炸弹，故本游戏的标记炸弹的方法为左键点击一下 `nonVisited` 的按键，将其转化为 `flag` 状态，再左键点击一下 `flag` 状态的按键则会将其转换成 `visited` 状态。

一类是重置网格大小（扫雷游戏规模的按键）（`Reset`, `Toy`, `Easy`, `Median`, `Hard`, `Hell`），他们的回调函数是 `restartCallback/scaleCallback`，会重置游戏/调整网格的大小以及地雷的数量占比。（`Reset` 会保留当前网格大小重置游戏，`Toy` 5x5 网格 0.15 的地雷比例，`Easy` 10x10 网格 0.15 的地雷比例，`Median` 15x15 网格 0.16 的地雷比例，`Hard` 20x20 网格 0.2 的地雷比例，`Hell` 25x25 网格 0.3 的地雷比例）



最后一类是自动扫雷按钮（`Auto`, `Deter`, `Infer`, `Random`, `Proba`），分别对应自动扫雷、单格推断、多格推断、随机扫雷和概率扫雷，其回调函数为 `autoMineButtonCallback`。同时回调函数中包含一个 `string` 类型的变量来决定具体的扫雷操作。

此外 GUI 界面还有一个显示当前状态的区域，在游戏过程中会显示剩余炸弹（除去 `flag` 数量），游戏结束时显示输赢的状态。



---

## 四、文件结构与说明

### 文件列表

- Util.hs: 包含一些功能函数, 如生成指定长度统一内容的列表的函数 `getUnitList`, 获得列表最大值的函数 `listMax`, 列表去重复元素函数 `unique` 等。
- MyRandom.hs: 生成随机数的函数, 主要有 `getR` 和 `getRList`, 分别生成一个随机数和一个不重复的有序的随机数列表。
- Grid.hs: 包含网格基本操作的函数。
- Mine.hs: 包含扫雷基本操作的函数, `sweep`, `setFlag`, `unsetFlag`, `automaticMine`, `deterministicMine`, `randomMine`, `inferenceMine`, `probabilisticMine` 等。
- Flow.hs: 包含终端版本扫雷的流程控制函数
- MineSweep.hs: 包含终端版本扫雷的 `main` 函数。
- MineSweepGUI.hs: 包含 GUI 版本扫雷的流程函数和 `main` 函数。