基于微软“约束求解器”Z3 的人机结合扫雷程序设计

一、实验目的：

（1）理解 AI 的约束求解（Constraint Solving）技术的实际应用；

（2）理解游戏智能（Game AI）的一种实现方法；

（3）熟悉前台 GUI 程序与后台“推理程序”（Automated Reasoner）的一种集成方法；

（4）熟悉基于 C#的 GUI 程序开发、文件 I/O 操作；

（5）熟悉基于 Python 程序设计和 Python 与微软 Z3 求解器的集成应用。

二、实验内容：

（1）熟悉扫雷游戏的规则和人工玩法与智慧。

（2）使用 Visual Studio Expres 2008 等轻量级 C#开发环境，编写扫雷 GUI，实现原型类似图 1 的 GUI 界面。

图 1. 扫雷游戏程序的 GUI 界面参考

（3）游戏者（用户）点击“推理”按钮触发推理过程。用 Python 语言、Z3 求解器的 Z3py 库 API 实现对当前格局的推理功能，提示出“无地雷的格子”的行号和列号，游戏者根据提示点

击相应的格子。此过程重复，直至扫雷完成。

三、技术方案

如图 2 所示。

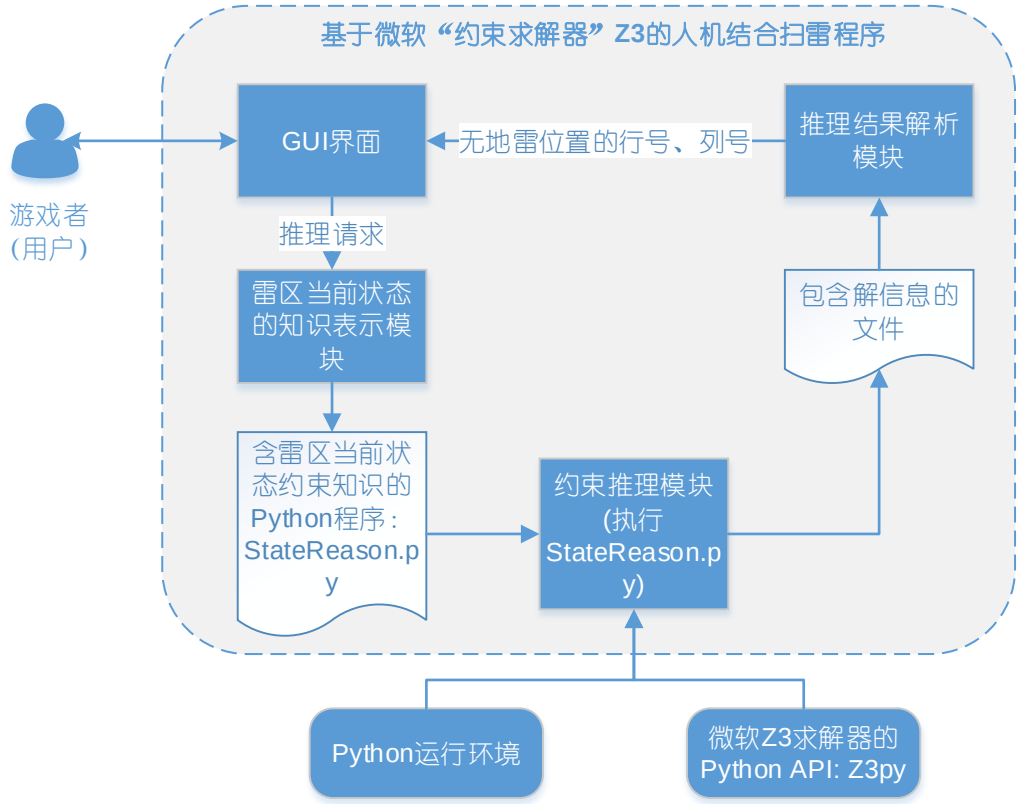

图 2. 基本技术方案

四、实验要求

（1）单人完成，单人演示答辩。

（2）撰写实验报告，含程序截图，“雷区状态”的表示方法，关键源代码，关键效果截图等。

（3）本实验项目总成绩 100 分，占课程总成绩的 15%。成绩由指导老师根据学生演示与解说结果评定。

五、技术参考

1、从 Github 仓库下载一个可用的扫雷程序的 C#源代码，借鉴：

https://github.com/hxl9654/Mine-sweeping

2、搭建 Python 编程 IDE：Anaconda2 （使用 Python 2.7，64 位版本）

3、安装 Z3Py，实现 Python 可用的 Z3 求解器版本。

参考：https://github.com/Z3Prover/z3/wiki/Using-Z3Py-on-Windows （

转到 https://github.com/Z3Prover/bin/tree/master/releases

下载目前最新的 64 位版本：z3-4.6.0-x64-win.zip）

按照：https://github.com/Z3Prover/z3/wiki/Using-Z3Py-on-Windows 的说明配置环境变量。

在 z3py 的解压目录\bin\python，找到 example.py，在 Spyder 中打开并运行，测试编程环境是否成功。

4、Z3Py 入门

https://blog.csdn.net/qq_35713009/article/details/88743410

5、将“哪些位置上有地雷”这个问题，表示为若干“约束”，使得 Z3 能够推理得出“无地雷的位置”，我们根据 Z3 的推理结果，手工点击无地雷的位置。

6、将“雷区”当前状态编码为约束的方法原理：

For those who are not very good at playing Minesweeper (like me), it's possible to predict bombs' placement without touching debugger.

Here is a clicked somewhere and I see revealed empty cells and cells with known number of "neighbours":

What we have here, actually? Hidden cells, empty cells (where bombs are not present), and empty cells with numbers, which shows how many bombs are placed nearby.

3.6.1 The method

Here is what we can do: we will try to place a bomb to all possible hidden cells and ask Z3 SMT solver, if it can disprove the very fact that the bomb can be placed there.

Take a look at this fragment. "?" mark is for hidden cell, "." is for empty cell, number is a number of neighbours.

	C1	C2	C3
R1	?	?	?
R2	?	3	.
R3	?	1	.

So there are 5 hidden cells. We will check each hidden cell by placing a bomb there. Let's first pick top/left cell:

	C1	C2	C3
R1	*	?	?
R2	?	3	.
R3	?	1	.

Then we will try to solve the following system of equations (*RrCc* is cell of row r and column c):

- R1C2+R2C1+R2C2=1 (because we placed bomb at R1C1)
- R2C1+R2C2+R3C1=1 (because we have "1" at R3C2)
- R1C1+R1C2+R1C3+R2C1+R2C2+R2C3+R3C1+R3C2+R3C3=3 (because we have "3" at R2C2)
- R1C2+R1C3+R2C2+R2C3+R3C2+R3C3=0 (because we have "." at R2C3)
- R2C2+R2C3+R3C2+R3C3=0 (because we have "." at R3C3)

As it turns out, this system of equations is satisfiable, so there could be a bomb at this cell. But this information is not interesting to us, since we want to find cells we can freely click on. And we will try another one. And if the equation will be unsatisfiable, that would imply that a bomb cannot be there and we can click on it.

3.6.2 The code

```
#!/usr/bin/python

known=[
"01?10001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"????1001?",
"????3101?",
"?????211?",
"?????????"]

from z3 import *
import sys

WIDTH=len(known[0])
HEIGHT=len(known)

print "WIDTH=", WIDTH, "HEIGHT=", HEIGHT

def chk_bomb(row, col):

    s=Solver()

    cells=[[Int('cell_r=%d_c=%d' % (r,c)) for c in range(WIDTH+2)] for r in range(HEIGHT
        +2)]

    # make border
    for c in range(WIDTH+2):
        s.add(cells[0][c]==0)
        s.add(cells[HEIGHT+1][c]==0)
    for r in range(HEIGHT+2):
```

```
        s.add(cells[r][0]==0)
        s.add(cells[r][WIDTH+1]==0)

    for r in range(1,HEIGHT+1):
        for c in range(1,WIDTH+1):

            t=known[r-1][c-1]
            if t in "012345678":
                s.add(cells[r][c]==0)
                # we need empty border so the following expression would be able to work
                    for all possible cases:
                s.add(cells[r-1][c-1] + cells[r-1][c] + cells[r-1][c+1] + cells[r][c-1]
                   + cells[r][c+1] + cells[r+1][c-1] + cells[r+1][c] + cells[r+1][c
                   +1]==int(t))

    # place bomb:
    s.add(cells[row][col]==1)

    result=str(s.check())
    if result=="unsat":
        print "row=%d col=%d, unsat!" % (row, col)

# enumerate all hidden cells:
for r in range(1,HEIGHT+1):
    for c in range(1,WIDTH+1):
        if known[r-1][c-1]=="?":
            chk_bomb(r, c)
```

The code is almost self-explanatory. We need border for the same reason, why Conway's "Game of Life" implementations also has border (to make calculation function simpler). Whenever we know that the cell is free of bomb, we put zero there. Whenever we know number of neighbours, we add a constraint, again, just like in "Game of Life": number of neighbours must be equal to the number we have seen in the Minesweeper. Then we place bomb somewhere and check.

Let's run:

```
row=1 col=3, unsat!
row=6 col=2, unsat!
row=6 col=3, unsat!
row=7 col=4, unsat!
row=7 col=9, unsat!
row=8 col=9, unsat!
```

These are cells where I can click safely, so I did:

Now we have more information, so we update input:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"???331011",
"?????2110",
"???????10"]
```

I run it again:

```
row=7 col=1, unsat!
row=7 col=2, unsat!
row=7 col=3, unsat!
row=8 col=3, unsat!
row=9 col=5, unsat!
row=9 col=6, unsat!
```

I click on these cells again:

I update it again:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"222331011",
"???2??2110",
"????22?10"]
```

```
row=8 col=2, unsat!
row=9 col=4, unsat!
```

This is last update:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"222331011",
"?22??2110",
"???322?10"]
```

...last result:

```
row=9 col=1, unsat!
row=9 col=2, unsat!
```

Voila!