Solver and GUI for Isolation

-Computational Game Theory Course Project Report

Fangshu Gao
Renmin University
gaofangshu@foxmail.com

Jiawei Li
Peking University
1600012712@pku.edu.cn

Abstract

*We implemented a solver and GUI for game Isolation in 4*4, 4*5 and 5*5 cases, and did primary analysis in 4*4 case.*

Rule of Isolation

***Isolation** is a two-player abstract strategy board game. It is played on a 7x7 board which is initially filled with squares, except at the starting positions of the pieces. Both players have one piece; it is in the middle position of the row closest to his/her side of the board.*

A move consists of two subsequent actions:

1. *Moving one's piece to a neighboring (horizontally, vertically, or diagonally) position that contains a square but not the opponent's piece*
2. *Removing any square with no piece on it, similar to the arrow effect in Game of the Amazons.*

The player who cannot make any move loses the game.

The above is copied from Wikipedia:
https://en.wikipedia.org/wiki/Isolation_(board_game)
Original game is too big to solve, we use a smaller game board instead.

Division of Work

Gao Fangshu was in charge of GUI, Li Jiawei was in charge of solver.
Both persons were in charge of testing and analysis.

Number of Positions

#Upper bound of the positions:
For a $N * M$ board: $2^{N*M} * (N * M)^2$

- 4*4: 16777216 (16.8M)
- 4*5: 419430400 (420M)
- 5*5: 20971520000 (21G)

#Actual positions:

- 4*4: 3617229 (3.6M)
- 4*5: 97976968 (98M)
- 5*5: 4967156731 (5.0G)

#Actual positions is about 4 times smaller than the upper bound. And both of them are very big!

Language & Tool

We both use Python and C++, Python for GUI and C++ for solver: use Visual Studio for C++ part and PyCharm for Python.
We developed in Windows, but also use a Linux server to run the solver.
Remark: The Solver for 5*5 case needs 20GB memory, we have to run on Linux server, but the Reader use a function(`_fseeki64`) which was only support in VS.

Implementation of Solver

It's **impossible** for Python to solve 5*5 case in one day, so we write a C++ Solver.

Hash Function:

The hash function must be efficient both in time and space. So we use a very delicate way to hash the position and the value/remoteness pair, so that the memory usage and the size of the database file is exactly same as the #Upper bound.

Hash rule:

Use mix-based number to represent a position (game map + player's location)
Game board grids were numbered as following, for an example of 4 * 5:

```
0  1  2  3  4
5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

Lowest $n * m$ is binary bit, represent the game board then is two digit of base-(n*m), represent the location of player0 and player1

For example, a gameboard like this: '.' is valid grid, 'x' is deleted grid, 0 is player0, 1 is player1

```
0...
...x
.x..
...1
```

Then hash values can be calculated as:

$$(2^7 + 2^9) + (2^{16} * 0) + (2^{16} * 16 * 15)$$

Store the Game Information:

We use an 8 bit integer to represent the information of one position., the highest bit means win/lose (win is 0, lose is 1), other 7 bits is the remoteness.
For example: (10001110)_2 means: LOSE and 14 remoteness.

Time for running the solver:

- 4*4: 3.938s
- 4*5: 106.075s
- 5*5: 114m 57.963s

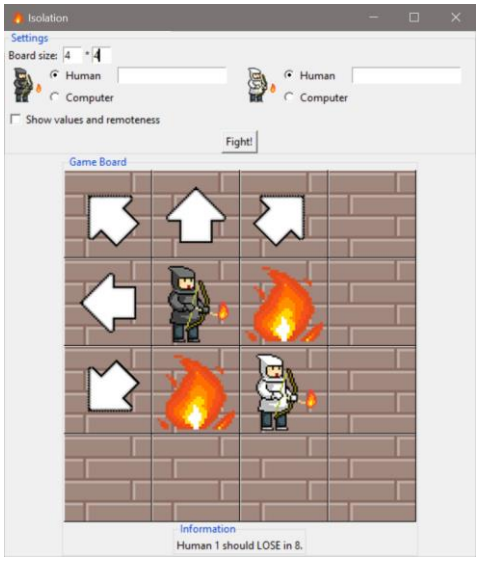
Read the Database:

The database file is really big, you can't load it to the memory during running, so a fast C++ reader is also needed. The Reader will directly put its file pointer to the location which contain the information of the required position.

Implementation of GUI

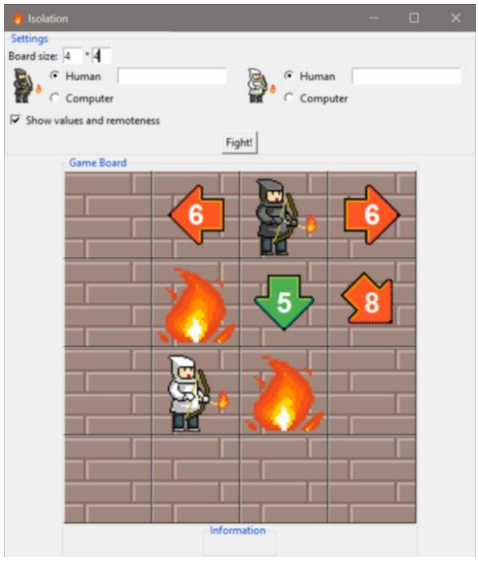
Introduction

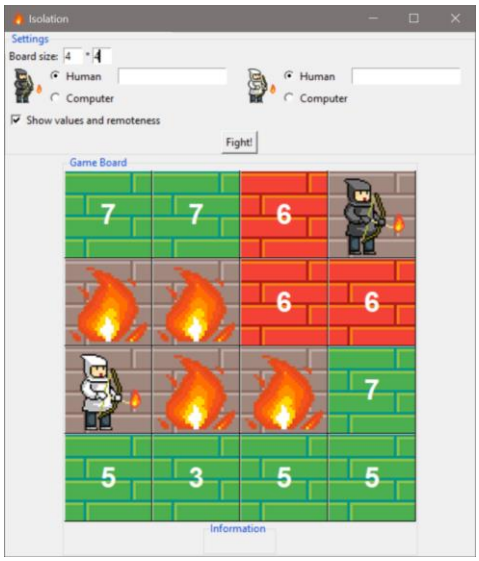
We implement GUI with Python Tkinter. This GUI can be applied to game board with any size. Under each size, if you have solved game by solver, there are four modes: human vs. human (finished), human vs. computer (finished), computer vs. human (finished), computer vs. computer (still have bugs). The layout are shown as following:



For each turn (move and delete), GUI shows possible movements and slots have been blocked (on fire).

In addition, we implement prediction function:





Clicking green arrows and slots leads to victory, and remoteness of each choice is shown in number. Computer always makes best choices (minimize remoteness when it should win, maximize remoteness when it should lose).

Code Structure

Applying OOP procedure, GUI starts with GameGUI class, which initiates Settings class, Env class and GameBoard class. User first set board size, players, prediction via GUI to Setting class, then the settings are delivered to Env. Env class stores game and dynamic variables (current turn, current position, etc.), and it is connected to Database class. Env class serves as a "transit station", it gets position from GUI, then searches it in database, feeds back corresponding values and remoteness to GUI.

Challenges

Computer can play two steps (move, delete) each time, but human only plays one step by clicking mouse. We only know values and remoteness after deleting, values and remoteness after moving are unknown. Thus, for a turn of n moves and m

deletes for each move, we need to search n*m positions in huge database. Then from n*m possible positions we go backwards to get values and remoteness of n possible moves.

The procedure introduced above happens in different classes (GUI, Env, Database), it is complicated to deliver data and command among classes. Controlling this in GUI/game environment can easily cause bugs.

However, we solved this problem by organizing classes carefully and refactoring the code.

Analysis

On 4*4 board:

- #Upper bound is 16777216, #actual position is 3617229
- Start position's value is 0(WIN), remoteness is 11, smaller than longest possible step 14.
- #Primitive position is 231480.
- #WIN position is 2737177, #LOSE position is 648572, #WIN is much larger than #LOSE, which means that go first has great advantages.
- #Average children for all position is 17.246, which is not a small number.

On 5*5 board:

- #Upper bound is 20971520000, #Actual position is 4963933511
- Start position's value is 0(WIN), remoteness is 13, only 2 steps slower than 4*4 case.
- #Primitive position is 213427647
- $\#WIN = 3712154601,\ \#LOSE = 1041574483,$ $\frac{\#WIN}{\#LOSE} = 3.56,$ go first's advantages shrinked a little, maybe because you can't kill opponent easily in a larger board
- #Average children is 32.788

How to get and play our game

Our project has a github link:
https://github.com/GaoFangshu/solve-games
Then you can follow the instruction given by the ***readme.md*** to play our game.