Classes

The application class will control the game graph generation. The application class will use an interface called IGame to initialize the game board, to generate plays for the current board situation, and to execute plays. The application class will use the IDatabase interface to write the game graph to a database. Since the prognosis of the all the resultant plays is required to determine the prognosis, the game graph will be built from the beginning moves towards the end game moves and saved from the end game moves towards the beginning moves.

A play will be a collection of modified squares. The application will execute the plays by modifying the game board using IGame::getSquare. The application will track and manage the play prognoses and manage the persistent data.

IGame Interface

|  |  |  |
| --- | --- | --- |
| Method Name | Purpose |  |
|  |  |  |
| getValidDimensions | Returns the valid dimensions for the current game. |  |
| initialize | Initialize the game |  |
| makePlay | Execute an individual play and generate the next set of available plays |  |
| undoLastPlay | Undo last play to allow execution of next available play |  |
| getScore | For display only |  |
| getSquare | Return a reference to the game square at the supplied coordinates for modification or display |  |

IDatabase Interface

|  |  |  |
| --- | --- | --- |
| Method Name | Purpose |  |
|  |  |  |
| createPlay | Writes a game graph entry and returns a key for future access | Used by additive games like Tic Tac Toe and Connect Four |
| createMove | Writes a game graph entry and returns a key for future access | Used by subtractive games like Checkers |

An abstract class called Game will implement the IGame interface.

Game Class

|  |  |  |
| --- | --- | --- |
| Method Name | Purpose |  |
|  |  |  |
| makePlay | Implements IGame method |  |
| undoLastPlay | Implements IGame method |  |
| getScore | Implements IGame method |  |
| getSquare | Implements IGame method |  |
| getAvaliablePlays | Abstract – Returns valid plays for the current board situation |  |
| getSymmetries | Abstract – Returns symmetries for the current board situation |  |

The classes derived from Game will implement IGame::initialize and IGame:: getValidDimensions. These classes will also implement the abstract method that supplies available moves for the current board situation.

The actual game logic will be supplied by Game derives classes like Checkers, Reversi, etc.

GUI Classes

The application will use an Interface called IView to display the current board situation. For each game, an IView derived class will display the current game board.

Game Graph Generation

The main application will function as a controller. After board initialization, the application will execute each play. After each play a new set of possible moves is generated and each of those plays will be executed. This continues until a game end board situation is reached and the last play gains a prognosis. When all plays for a given board situation are executed, the prognosis is propagated back to the previous play. When a prognosis is determined, the play can be written to the game graph database.

To avoid duplicate efforts and duplicate records in the game graph database, every board situation will be compared with previously analyzed board situations. If the situation is found or a symmetry of the situation has been analyzed game graph generation will stop and the game graph will be connected to the previously analyzed graph entry.

Database Records

One database, called “Metadata”, will contain metadata about the game graphs. Each record, a combination of game, board dimensions, and variable parameter, will point to the game graph database for that graph. At this writing MongoDB will be the database of choice. This easily allows different data structures for different games. One example is the difference between Checkers, a subtractive game, and Tic Tac Toe, an additive game. A JSON based document database also allows for the needed variability in the game graph edges, descendent plays or symmetrical links.

The Graph Entry records will contain only a “move” or “add” key. “move” will apply to subtractive games like Checkers and “add” will apply to additive games like Reversi, Connect Four, etc. This also allows for games with other types of plays.

The GraphEntry records will contain only a “childPlays” or “symmetry” key. “childPlays” will apply to edges that produce a new board situation and “symmetry” will apply to edges that link symmetrical board situations. If it works out that the “symmetry” keys can easily be added to the “childPlays” objects at game graph generation run time, this should be changed.

Metadata

{

"game":

{

"game name": "Checkers",

"width": 8,

"height": 8,

"games": “6543219876545,

"situations": “654987654321654987654”,

"player1wins": “654987321654”,

"player2wins": “9135787955”,

"draws": “654987”,

"database name": "Checkers8x8"

}

}

Graph Entry

{

"id": "00000000004",

"move": {

"start": [

{

"x": 1,

"y": 1

}

],

"end": [

{

"x": 2,

"y": 2

}

]

},

"add":

[

{

"x": 1,

"y": 1

}

],

"child plays": {

"ids": [

{

"id": "00000000001",

"prognosis": "Win"

},

{

"id": "00000000002",

"prognosis": "Draw"

},

{

"id": "00000000003",

"prognosis": "Lose"

}

]

},

"symmetry":

{

"id": "00000000005",

"symmetry": "HorzFlip"

}

}

The field names could be shortened to reduce the size of the database.

Threading

At this point it looks like the database records can be created with the prognosis already calculated. This will enable a design where the records are not read or updated during game graph generation. The IDatabase methods will queue up database writes for another thread to pick up and create database records.

The simplest threading model for generating the game graph will be to analyze different symmetries concurrently and terminate processing before the pristine board situation is added to the collection of previously analyzed board situations.