COMP2911 Project: Quoridor

Design Report

21 October 2012

Team Membership

Contributions (by class):

- `3390558` Chi-Chun Tiao
    - GameState
    - AIPlayer
    - HumanPlayer
    - Validator
    - Board
    - Client
    - Square
    - Wall
- `student_id` Jeffrey Loong
    - Client
    - List them

Summary

Behaviour of Provided Tests

This application passes all provided tests.

Features

1. Utility-based Intelligent Agent. Minimax implementation with alpha-beta pruning algorithm and utility (scoring) function.
2. Game Management
    - Undo/Redo
    - Game Save/Load

Details

This application relies heavily on the `GameState`. The game is said to be *cumulatively* validated in the sense that a move is passed to the mutator method of the class, and the fields of this class are only mutated if the move is valid from the current game state. If the move is invalid, the mutator does not update any fields and simply returns a boolean value of `false`.

As such, it is evident that the `Validator` class itself does not do any validating. It simply attempts to iterate through its list of strings and mutate the game state during every iteration, returning `false` as soon as any move is found to be invalid.

The `UserInterface` class acts as the game manager. It allows the user to select select game modes - human vs. human or human vs. AI. It also allows loading of saved games and saving of games either in progress or already finished, etc. Based on the user's game mode selection, the `UserInterface` will instantiate either a `HumanPlayer` or `AIPlayer`, both of which implement the `Player` interface, containing only one method that returns a move based on the `GameState` it is given. For the `HumanPlayer`, this is simple. We simply have to print the given `GameState` to standard output and read a move from standard input. The `AIPlayer` is not quite as simple for obvious reasons. Given the `GameState`, we use it as our root node and perform minimax on it by evaluating the children at the specified depth with the `AIPlayers` scoring function. The minimax algorithm has not been implemented to return the entire principal variation. Rather, it just returns the most optimal move. This AI is also somewhat adaptive, in that when all walls have been exhausted, we simply breadth-first search for the shortest path. The `UserInterface` takes the returned moves from each of these players on a turn-by-turn basis, and validates each move by the method described above. If the move is valid, the `GameState` is mutated. That is, fields updated accordingly. If not, we return an error message prompting the user to enter a valid move. This is continued until the game is over, as indicated by the `GameState`'s terminal test, at which point the winner is indicated.

The motivation for this *cumulative* validation mechanism is obvious. If we are confined to using the `Validator` class to manage our games, we would have to keep building a string on each turn and validate the entire string at each turn. A costly and redundant approach.

Having discussed all the components that rely on the `GameState`, we now turn to the details of the `GameState` itself. When a `GameState` is initialised, so is an adjacency matrix of the graph representing the squares of the board. Traversal and wall placement validation relies heavily on this graph. By way of example, when a wall is placed, a number of things occur: we remove the corresponding edges in the graph (by "edges", we really mean nodes from the list). We then do a path search for both players to see they can both reach their target rows and are not entirely trapped by walls (it goes without saying that we check walls do not cross other walls etc.) Now, using the adjacency list, it makes it very easy to check that an adjacent square is not blocked by a wall. One pitfall of this adjacency list is that validating jump moves does not involve the use of the list directly and depends a few other computations which effectively make the list redundant. This redundancy will be further discussed in the implementation reflection.

As a further note on the AI player, it can look-ahead in a reasonable amount of time up to 3-4 levels, but 4 levels might be pushing it just a bit (takes about a minute on average). It uses the alpha-beta pruning algorithm to perform minimax search and uses a relatively simple but slightly costly evaluation function:

```
MovesToFinish(MINPLAYER)-MovesToFinish(MAXPLAYER)
```

This is obviously quite intensive in the sense that we must breadth-first search a path to the finish for each player but turns out to work quite well.

As expected, the resulting AI player is quite competitive and very retaliatory. When you move, he moves. When you block him he will block you right back, and will often keep blocking you until you're right back where you started.

Since this evaluation function is quite accurate but computationally intensive, we must make the trade-off of depth of evaluation. So the minimax algorithm can really only comfortably look-ahead about 3 steps. But the accuracy of the evaluation function turns out to strike a good balance.

A number of other less accurate but significantly fast evaluation functions where experimented with, but none turned out as good as this one. They were:

1. Moves to next row
2. Rows to win
3. Number of walls left

Perhaps a hybrid of 3 above and the current evaluation function would yield a less aggressive and ill-tempered AI who will not exhaust walls quickly to his own peril.

Design Reflection

The encapsulation of data fields related to the state of the game into the `GameState` class made implementing other classes very simple. Furthermore, the responsibilities were divided into each separate class in a logical manner, so when core functionality was implemented, all else that followed became very easy.

Additionally, the `Player` interface provides a very good level abstraction for classes implementing human and AI players respectively. This means that the game manager is programmed to this interface and can thus run independent of who is a human player and who is an AI player.

While this `Player` interface was a good design decision, it is the *only* interface, and the design shortcomings can be attributed to the lack of interfaces. Practically all classes rely on other classes so any changes to the implementation can result in erratic behaviour. This is a key design flaw that has permeated since the conception of this design.

A noteworthy design shortcoming is the lack of design patterns where due. For example, the AI player, under different contexts are subject to different behaviours (e.g. when out of walls, etc.) The Strategy design pattern could have come into good use and allowed us to dynamically change AI strategies during runtime.

The Template Pattern could have been used for the game manager.

The Iterator Pattern could have been used by `GameState` to allow for iteration of child states.

These are the main design flaws that if detected earlier could have streamlined the development process and allowed for extensibility of our software components. Other key implementation limitations are described below.

Implementation Reflection

Some implementation limitations include:

- Two Player

  The implementation of this game only takes into account 2 players. Though extending the game to 4 players would not be too difficult, it is definitely non-trivial.
- Adjacency List and speed

  The adjacency list used by the `GameState` is a HashMap with Square as key and a List of Squares as values. This renders deep copying of `GameState`s quite costly, an operation heavily used by the alpha-beta pruning algorithm in child state exploration. To make a deep copy of of this list, we cannot simply pass our old HashMap to a copy constructor, as the references to the Lists would remain the same. Instead, we have to iterate through the HashMap and add all items of 81 lists individually. This adds unnecessary time complexity that makes minimax of levels deeper than 4 practically impossible.
- Node adjacency inconsistency and inaccuracy

  The adjacency list is used frequently to generate valid traversals from a given `Square`. This is clearly wrong and fails to take into account jump traversals. This method of neighbour generation is employed by the path searching algorithm provided in the `GameState` class and will cause problems with AI's that rely on this to find shortest paths. While the method which generates valid moves from the current state does not use this list to infer neighbouring nodes, the is a minor implementation flaw that also does not account for jump moves.

The adjacency list is really a huge space trade-off for a little speed, that ends up being spent on copying the list. The adjacency list only serves to encapsulate adjacency of `Square`s in the presence of walls and nothing else. Instead, the `isValidTraversal` method should simply look in a 2-neighbourhood of the square, and infer if there is a wall obstructing the destination square, along with the other operations it performs, independent of the adjacency list.

In light of the above shortcomings that the adjacency list bears, it would be better to do away with it completely, and this was touted for in the version to be submitted but has not been done, and probably will not be done in time.

Teamwork Reflection

- Did the team work together effectively?
- Was there a dominant person?
- Does one person deserve extra credit?
- Did anyone not contribute effectively at all?