CS 3110 Project Design: Quoridor

Matthew Gharrity (*mjg355*)
Sattvik Kansal (*sk2278*)
Tyler Thompson (*tt395*)
Victor Reis (*vor3*)

Abstract

Quoridor is game consisting of two or four players, starting on opposite ends of a square board. Each player wants to move their piece to the other end, and they have two options per turn: (1) move their piece to any adjacent space, or (2) place a wall on the board between spaces, the goal being to hinder the opponent's progress. This project allows for particularly interesting artificial intelligence challenges (there is currently no AI developed that can play optimally), along with the chance to experiment with making a graphical user interface in OCaml.

See the following links for more details on the rules, along with a short video explaining the basics.
https://en.wikipedia.org/wiki/Quoridor
https://www.youtube.com/watch?v=zrIiWN4cnxg

Additionally, see the following link for a lengthy analysis of AI techniques for Quoridor.
http://www.mendeley.com/download/public/2354731/3799351832/acad6962a9bb3eb3fde4272f476d6625eb0a8182/dl.pdf

Key Features

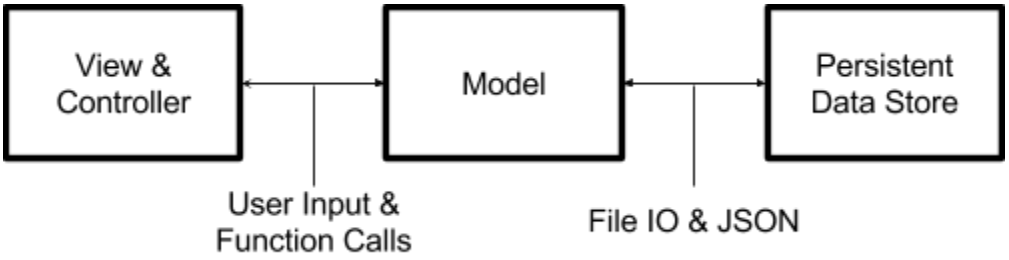
- Simple GUI showing the board, available moves, and tiles remaining
- Ability to play against another human, taking turns
- Artificial intelligence to play as opponent, using graph algorithms and other researched techniques
- Ability to save the game in a persistent file
- Stretch goal: different game types (hexagon cells or different wall tile shapes)
- Stretch goal: networked multiplayer mode using sockets

System Description

When the user starts the game, (s)he is given the option to either load the game from a previously saved state (stored in a folder meant for saved files) or to create a new game. When they create a new game they can play against the AI or another player (on the same computer).

After the game starts, the player can use the GUI to place walls or change their position on the board. Information about the number of walls left is displayed as well as the valid movements for the player. The other player/AI then makes their move. The game continues until one player wins or the other player forfeits or saves the game.

Architecture:



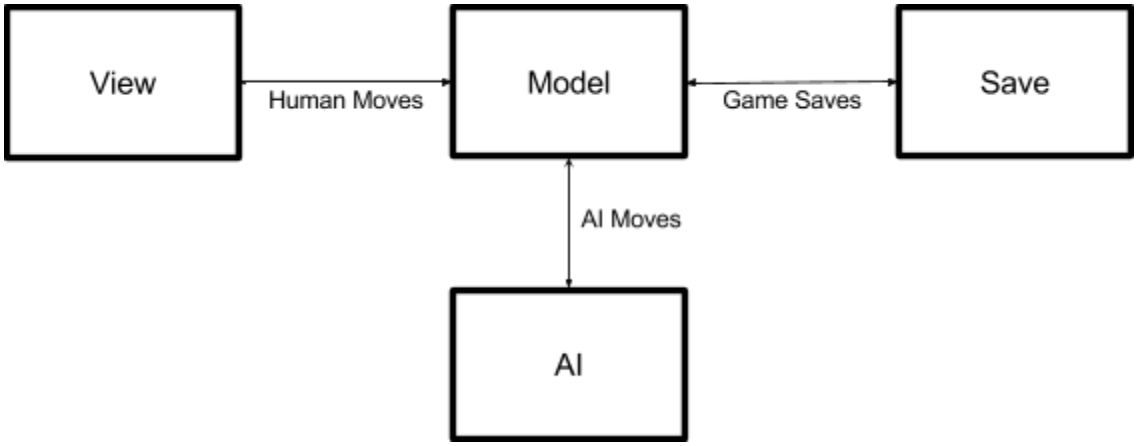
Components

- The View & Controller displays the game and processes user input
- The Model keeps track of game state
- The Persistent Data Store stores game saves

Connectors

- The View & Controller will interact with the Model through direct function calls
- The Model will interact with the Persistent Data Store using File IO & JSON.

System Design:



- **model.mli**
  The model module will provide game state. Other modules wishing to update the game in any way will need to go through its interface, which will provide functions for getting the board object (player, space, or wall) at a particular location, along with other information about the current state of the game that is needed by the GUI and the AI.

- **view.mli**
  The view module will provide a graphical user interface for the game. We are using an M(VC) pattern here, so this module will also act as a controller that responds to input from the user and that interacts with the model based on this input.

Will include a GUI and will handle player's turns, drawing the board, and handling the game state. Requesting moves from an AI and validating player moves by calling functions in model. Also will access save to allow for loading and saving calls.

- **ai.mli**
  The AI module will provide a computer-controlled opponent. It will use the current game state to estimate the best possible next move, and provide this choice to the model when requested.

  We will be trying out various strategies for the AI. We will quantify the configuration of a player with a score based on metrics such as distance to the other end of the board, number of paths available, number of branches in these paths, and the number of walls remaining for each player. Developing the AI will involve several algorithms, such as graph search, which we will design with efficiency in mind in order to avoid freezing up the game for an extended period of time. We will also be consulting the extensive literature available online on making AIs for quoridor and customizing AI techniques that we find useful. We will also try to figure out techniques based on intuition and experience playing the game, e.g. blocking the closest way to the edge of the board when the second best option is much more expensive.

- **save.mli**
  The save module will provide game state saving and loading to and from files. It will be able to convert game states to and from JSON.

  JSON files will store the contents of the game board as a two dimensional array, along with information about which player should go next.

Data Structures:

- The board will be represented with a matrix data structure (two dimensional mutable array) representing spaces, tiles, walls and player locations, so that we can lookup valid neighbors for each location in constant time. Half of the positions represent possible wall locations, one fourth represent spaces and one fourth represent corners. The data structure takes O([rows] x [cols]) space. Mutable arrays are better than functional linked lists in this case for efficiency and ease of updating the model.
- Each 'wall location' is a list of segments (coordinates) making up the wall.
- 'Move' is a variant that either represents changing the player position or placing a wall.
- JSON will be used to store game saves.

External dependencies:

- Graphics and LablGTK2 for creating the graphical user interface
- Yojson for creating and parsing game saves
- Async for networking (stretch goal)
- Ocamlgraph : contains several graph algorithms such as Dijkstra's and Bellman-Ford.

Testing plan:

We'll have the other 3 members try to break the system functionality built by one member using the tests and UI. We'll set up extensive black-box testing as well as glass-box testing for individual functions and modules as well as for the entire system, testing each of the rules of the game, including tricky cases in which a player may move across another player. We will also test the AI by checking whether the outputted moves result in a better position for the computer; better yet, we will convince family and friends try to beat the AI, and analyse the strategies that the AI performs well or poorly against.

We will try saving/loading from files and test out both the player versus player and player versus AI modes. We will also use invariants to make sure the the board always stays in valid configuration and print out debugging information about the AI algorithm and the game state.