CS 3110 Project Design: Quoridor

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

Note: this design document has been updated a lot since the first draft, so it is well worth the read!
Notable updates are highlighted in green.

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 (without making it impossible for them to win). This project provides a particularly interesting artificial intelligence challenge (the game is complex enough that there exists no AI that can play truly 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, the following link has a lengthy analysis of AI techniques for Quoridor.
http://www.mendeley.com/download/public/2354731/3799351832/acad6962a9bb3eb3fde4272f476d6625eb0a8182/dl.pdf

Key Features

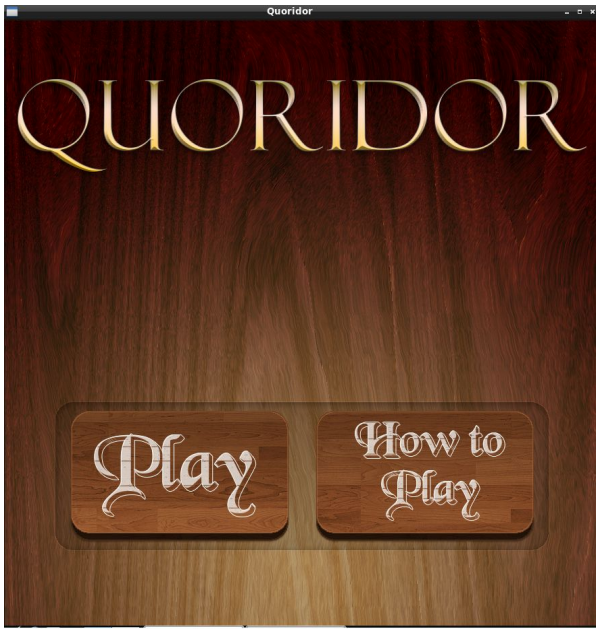
- GUI showing the board, available moves, and tiles remaining
- Ability to play against another human, taking turns
- Artificial intelligence to play as opponent, using graph algorithms and other researched techniques
- Ability to watch two artificially intelligent players face each other

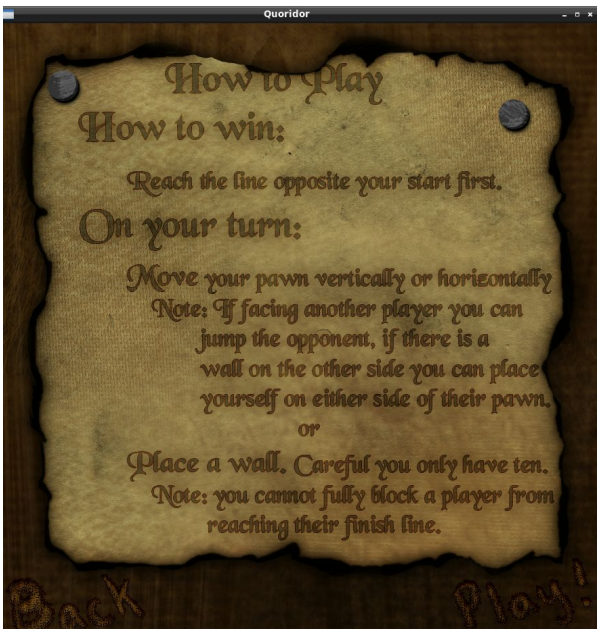
System Description

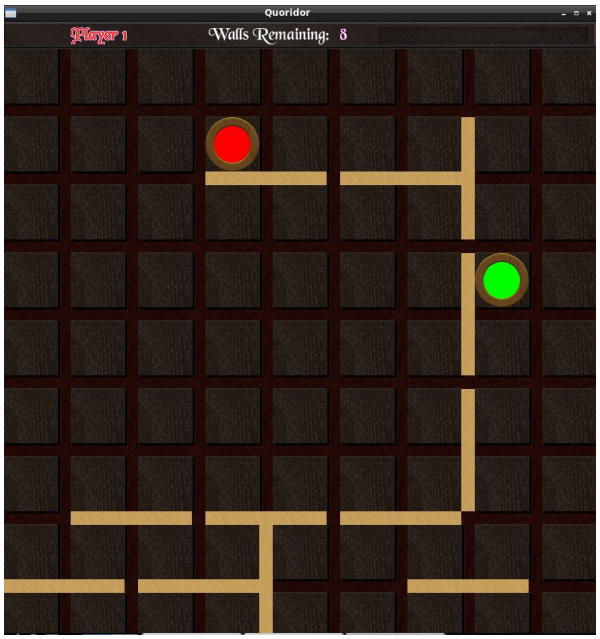
When the user starts the program, they are given the option to start a new Quoridor game with a chosen combination of human and AI players, each with a customizable color. There is also an option to display the rules of the game.

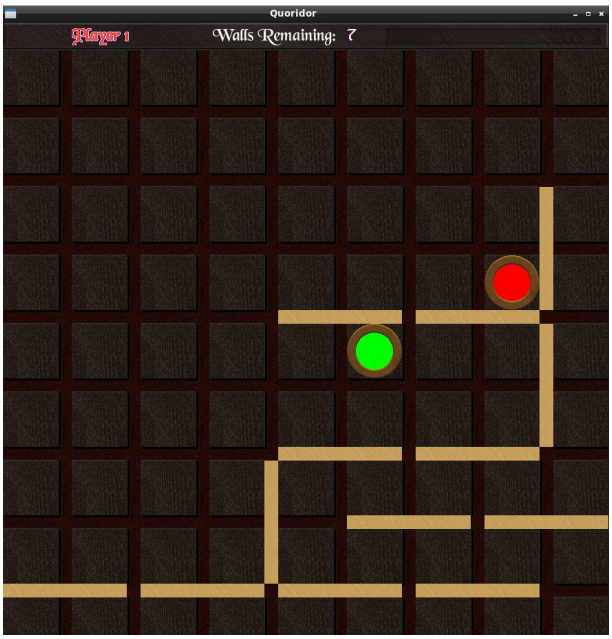
After the game starts, the player can use the GUI to place walls or change their position on the board using the mouse. Information about the number of walls left is displayed as well as messages about recent invalid move attempts. The other player/AI then makes their move. The game continues until one player wins by reaching the opposite end of the board.

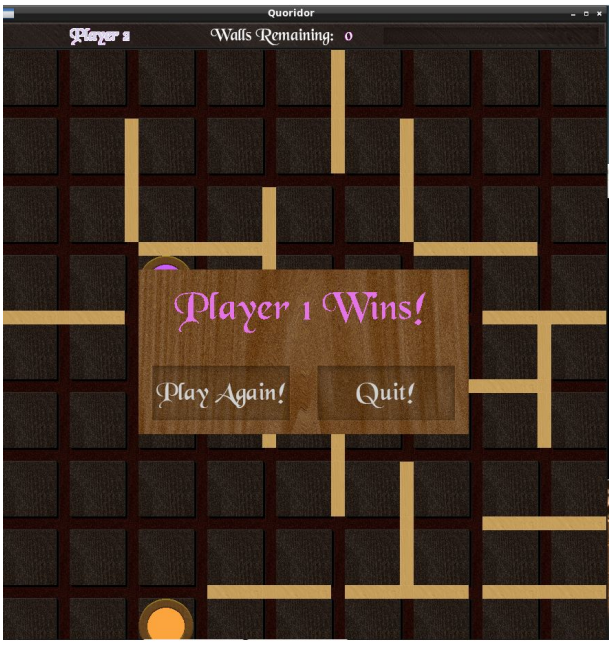
Screenshots



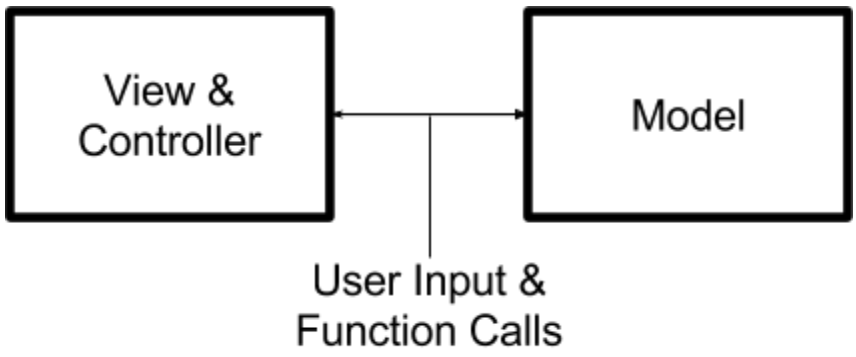








Architecture:



Components

- The View & Controller displays the game and processes user input
- The Model keeps track of game state

Connectors

- The View & Controller will interact with the Model through direct function calls

System Design:

- **model.mli**
  The model module will provide game state. Other modules wishing to update the game will need to go through its interface, which will provide functions for validating and committing moves, along with other information about the current state of the game that is needed by the GUI and the AI. We decided to make the data structure for the game board concrete so that it was easier for other modules to traverse it.

- **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.

- **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 spent a significant amount of time researching and trying different AI strategies and optimizations, and we hope you enjoy playing against it.

  The first thing we did was to quantify the state of a quoridor game, so that we know how well a particular player is doing relative to another. We experimented with several different heuristic functions, but ended up using a function involving the squared distance to the end goal for a player and the number of walls that the player has remaining. This gave the AI nice properties; for example, it is much more worried about blocking its opponent if the opponent is already close to its goal. To keep the AI interesting, we choose randomly among optimal moves, and the random number generator is seeded with the system time so that the gameplay is not deterministic across gameplays.

  The core of the AI is the minimax algorithm (https://en.wikipedia.org/wiki/Minimax), which simulates possible combinations of moves and chooses the best one. Our minimax function is recursive to an arbitrary depth, but we cap the depth to two so that the AI is reasonably fast.

  AI speed was a large priority, so we implemented several optimizations to the basic minimax algorithm and graph searching:

  - Alpha-beta pruning (https://en.wikipedia.org/wiki/Alpha–beta_pruning). By keeping track of the best sequence of moves seen so far, we can stop evaluating entire subtrees of move combinations if we find that it is impossible for the subtree to result in a better score than the best score seen so far. This approximately doubled the speed of the AI.

  - Where it matters most, we use mutable arrays to keep calculations fast and cache-friendly.

  - A large amount of time for the AI is spent computing shortest paths through the board, which are used in the heuristic function. Since some moves don't affect the shortest path for a player, we keep track of the shortest paths themselves and check if they've been

disrupted by a wall. If the path remains intact, we do not bother to do a full DFS of the board.

We originally thought about using other metrics such as number of paths available, or number of branches on these paths, but we found it more beneficial to focus on optimizing minimax.

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.

External dependencies:

- Graphics for creating the graphical user interface

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 also implemented AI v. AI gameplay so that we could watch the AI face itself and analyze the results.

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.

We do not have any known bugs.

Division of labor:

Tyler: I created the GUI (Programming and creation of the images), I also assisted in various bug fixes as necessary within both model and the ai. With the addition of testing done for the GUI and creating the images I spend around probably 30-50 hours on the project, never really kept track.

Victor: I worked mostly on the model (move validation), on optimizing the helper functions for the AI (dist_to_win, get_valid_moves) by only recomputing values when a move intersects a previous winning path,

and on trying out different heuristics for the AI. I would guess I spent around 20 hours in addition to group meetings but it's hard to give a precise number.

Sattvik: I helped with brainstorming ideas, designing the model and implementing shortest distance to the winning position for the AI. I would estimate that I put in about 10-15 hours on this project.

Matthew: I worked primarily on designing the model and optimizing the AI (esp. alpha-beta pruning and generating all possible valid moves). I also updated the design document for final project submission. It's hard to estimate how many hours I worked on this project, but I would put the estimate close to 20 hours (in addition to group meeting time).