Building a usable interface and a computer opponent for a new game

Samer Sabri
Computer Science Senior Project (CPSC 490)
Advisor: Professor James Aspnes

May 1, 2013
1 Introduction

The motivation behind this project came from three forces: the first one was my desire to deepen what I had already learned in my Computer Science classes in terms of programming techniques, algorithms, and data structures. The second was my interest in game theory and games in general, chess in particular. The third force was my curiosity about web technologies: I felt that although I had become a decent computer scientist, I would never look cool in the eyes of the mainstream if I did not know how to write a nice webpage.

I found a way to pursue these three forces through a simple tile-based game I created that is called “Angry Artists” (see description in section 2). After consulting with my advisor, Professor Aspnes, I decided to (1) write a webpage that would allow two human players to play the game against each other and (2) create different computer players with varying levels of sophistication and competence that a user could play against.

I have been able to achieve the following through this project:

- Learn HTML, CSS, Javascript and jQuery
- Put visual and programmatic elements together by building a simple interface for two players to play against each other
- Improve upon that first milestone by changing the underlying structure to accommodate computer players
- Build simple computer players and use the understanding gain to iterate and build more powerful and complex algorithms

2 Describing “Angry Artists”\(^1\)

Angry artists is a two-player perfect information tile-based game. One possible story that could motivate the game is that each player is an artist collaborating with the other on a contemporary work of art. Each turn, one

\(^1\)This section is taken from my proposal and modified to correspond to the final version of the game that I used.
of the artists places a polygonal tile on a polygonal canvas with the constraint that the color of any side of the tile being placed has to agree with the color of the side of the tile adjacent to it. Because they are angry, the artists are trying to place as many of their tiles on the canvas and they win this passive-aggressive game if they are able to place more tiles on the canvas than their partner/opponent.

2.1 The version I used (5*5 squares)

I will start with a simple version of the game where the board is a square composed of 25 smaller squares. The pieces are tiles of the size of the small squares, and the pieces’ sides are labeled in red or yellow. Pieces are randomly assigned such that player 1 receives 13 pieces and player 2 receives 12 pieces. See figure 1 for a representation of the situation before any moves are played. It is important to note that the pieces can be placed rotated on the board.

![Figure 1: Board and piece allocation before any moves are played](image)
Player one starts by placing a piece, then the players alternate. A piece can only be placed in an unoccupied square, meaning that no piece has been placed on it yet. The other constraint the player needs to respect is that two adjacent pieces need to have sides of the same color.

The game ends when any of the players is unable to place any of their pieces. The player who is unable to place a piece loses. There is one exception to that rule: if a player cannot place any piece because she is out of pieces, then she wins. See figure 4 for a situation where player 1 loses because she ran out of legal moves.

See figures 1, 2, and 3 for the first two moves of a game of Angry Artists.

![Figure 2: Situation after first move](image)

2.2 Variations and possibilities

There are several ways to expand on the simple game shown and vary its complexity. I present the main variations here:
The square figure can be generalized to an \(n\)-sided polygon.

Pieces can be composed of \(k < n\) sides of the polygon.

The rule about not being able to place a piece in an occupied square can be dropped.

Victory can be based on the largest number of pieces placed.

One can even imagine 3-dimensional version of this game where we are trying to place cubes in such a way that the faces of the cubes agree with each other. This game can be played by building the structure in a bottom-up fashion or by placing the pieces in space with no regards for gravity.

Similarly, one can imagine dropping the pieces into columns in the fashion of Connect Four.

Figure 3: Situation after second move
3 Webpage Structure

I wrote the Javascript code in a way that enables it to handle any n×n square. However, n=5 is the only n that I tested visually and that works for the webpage layout I chose.

3.1 Visual organization

I divided the page into two big floats: one containing the board (which was built using a for loop with jQuery) on the left, and another one containing information about the number of pieces remaining for each player and for each player type.

I designed the pieces using Gimp, and allowed them to be placed rotated by an angle that is determined by the user. Placing a piece effectively swaps a jpg of the old empty square for a jpg of the piece. In order to allow rotations, I use jQueryRotate (https://code.google.com/p/jqueryrotate/), a library that allows to rotate pictures and that was written by Pawel Witkowski.
3.2 Game rules logic

The heart of the webpage was the representation of moves and especially the way to check if a move is legal or not. Initially, I represented the board as a two-dimensional array of n*n squares: for each i, array[i][0] represented the type of piece that was there (a piece of type one has opposing sides of the same color while a piece of type 2 has opposing sides of opposite colors) and array[i][1] represented the orientation of the pieces (1 or 2 for a piece of type 1, and 1,2,3, or 4 for a piece of type 2). Array[i][0] and array[i][1] were both 0 when no piece was placed at this square.

In order to check if a move is legal, I wrote a loop that would check if the square the piece was placed at is legal and if it conflicted with any pieces around it. When I started implementing computer players, I realized that it would be much more helpful to follow my advisor's suggestion and have an array of legal moves that I could update (I used the word prune) every time a move is played by removing moves that are rendered illegal. This also incidentally made it extremely easy to check if a move is legal.

3.3 Putting it together: human v. human

I kept a global variable called "turn" that determined whose turn it was to play. Depending on that variable, I used jQuery to bind an event handler to the click Javascript event on that player’s pieces. Once a piece is clicked, the user can click on it again to rotate it, and click on an empty square on the board to place it. If the move is illegal, my page throws an alert; it places the piece otherwise and updates the piece counter. If a piece counter gets to 0, the piece image is dimmed and the player can no longer use it (as you can see in figure 4).

3.4 Building on it: human v. computer

After changing the underlying architecture that determines the way I represent the board (represent it as a set of remaining legal moves rather than a set of moves played), implementing the computer players became easier. My legal moves arrays are legal1 and legal2, and are each composed of n*n+1 entries. In legal1, the first n*n entries contain an array each. Legal1[i] contains an array containing all the legal orientations of piece 1 at square i (it is empty if there are none). Similarly, legal2[i] contains an array containing
all the legal orientations of piece 2 at square i. Legal1[n*n] and legal2[n*n] contain the total number of legal moves for each of piece 1 and piece 2.

I will not go over the other changes I had to implement to allow computer players to play. Allowing computers to play against each other would require even more abstraction that I would implement if I had more time.

Based on that, I implemented several computer players:

- Deterministic computer: this computer takes the first move it sees in the legal moves arrays and plays it

- Random computer: this computer places all the legal moves in one array and chooses one randomly at every turn

- Random computer with winning move: this computer plays like the random computer until there are less than 30 possible moves left (legal1[n*n] + legal2[n*n] ≥ 30), in which case it checks for each move whether it will cause the other player to lose at the next turn. This computer player allowed me to successfully build and test a fundamental building block of the minimax algorithm.

- Random computer with minimax: I have not built this player yet, but I am going to in the next few days - the deadline requires me to submit this project earlier. This player waits for the number of legal moves to go below a certain threshold and then use the minimax algorithm to determine if there is a winning strategy. There are some implementation problems left to resolve but they seem surmountable.

I also considered implementing a player that runs minimax using randomly selected branches. This works in games where positions that emerge from the same position are similar enough. However, as my advisor pointed out, there would be too much noise if such a strategy is used because victory or loss depends very closely on the sequence of moves played. I suspect that this player would not fare much better than the random or deterministic computer.

4 Implementation: progress and hurdles

4.1 Project phases

I can identify the following phases in my implementation of this project:
4 IMPLEMENTATION: PROGRESS AND HURDLES

- Designing the game and defining the scope of the project
- Learning HTML, CSS, Javascript, and jQuery: I mostly relied on Codecademy.com and Stackoverflow.com, as well as a book recommended by my friend Bay Gross called Eloquent JavaScript: A Modern Introduction to Programming by Marijn Haverbeke
- Implementing a first version where moves were played through commands in the javascript code
- Implementing a final human v. human version and debugging it
- Changing the underlying representation of the board and implementing computer players

4.2 Difficulties and obstacles

I had to overcome several obstacles and hurdles, I will try to mention some of them both for my future reference in future projects and for any reader who is interested in executing a similar project:

- It was difficult to transition from learning to use these languages to actually using them. I was waiting till I had all the pieces I needed to implement my project. I think it would have been better to start writing code earlier even if I was not sure of what I was doing
- I had some trouble transitioning from a low-level language like C to Javascript. The most difficult part of the transition was the asynchronous nature of Javascript. For example, within the click handler for my pieces, I had a for loop that caused the piece to be placed on any square I click on. i went from 0 to 25, at which point it exited the for loop. However, any time I clicked a piece, the i I used inside the for loop was evaluated at 25 instead of the value it was at when the for loop ran for the first time. It took my a lot of time to find this bug because these kinds of problems do not really exist in C.
- Changing the underlying architecture of piece representation was particularly challenging because it required a deeper understanding of my program and how the pieces fit together, which demanded that I work back from the way I had broken it down to the general problem and back down to a new way of breaking it down
Navigating between visual elements and underlying structures was difficult. One of the reasons why I was not able to implement the minimax algorithm in time is that the number of pieces left for each player is coded into the webpage itself and not the legal1 and legal2 arrays. While I have the general roadmap to implementation, it will take me some time to rewrite my previously existing code to take that into account.

5 Conclusion and possible further work

This project has been particularly challenging and informative; I learned a lot and I feel motivated to deepen my knowledge of web programming and games.

When it comes to my personal skills, I would like to work on executing faster (avoiding being paralyzed by perfectionism), improving my aesthetic sensibilities (I apologize to the reader for the poor visuals of the webpage), and writing cleaner, more modular Javascript (there is a lot of repetition in my code despite my attempts at making things as modular as possible).

There is a lot of room for improvement when it comes to the project itself. I will implement the minimax algorithm. Other ideas include building a server architecture to allow two players to play remotely. It would also be nice to be able to play computers against each other (maybe play a large number of games and report statistics). I would also love to see the webpage scaled to handle larger or smaller n’s as well as other figures (triangles, hexagons, etc). Finally, it would be interesting to look into other heuristics for computer programs or even ones that include machine learning algorithms that allow them to change themselves and become better.