The Bewildering, Bizarre BilliardBot

Project Report
CS 4100/5100: Artificial Intelligence
Prof. Margot Lhommet

Paranoid Androids (Xingchen Cui, Alex Hersh, David Pimentel, Oleg Vaskevich)

April 23, 2014

Project Overview

Initially for our final Donjon project, we proposed BilliardBot, a software simulation and an artificially intelligent billiard (pool) playing robot. The requirements of this emulated robot were that it had to be able to observe the pool table and calculate the most optimal shot on each move, and that several bots can be instantiated in order to compete against each other at the same pool table.

We developed BilliardBot from scratch using JavaScript and the 2-D graphics library *Matter.JS*. Using various mathematical algorithms, an instance of the BilliardBot AI can calculate the most optimal move by judging the quality of each class of its available shots and, if any, the opponent's expected gain from the resulting ball positions. The software simulation itself is an easily-accessible website viewable in modern browsers, and displays a pool table in a graphical user interface, showing the tables state as well as each shot taken in real-time. This AI provides options for three different types of games: *Single AI*, in which the robot simply tries to finish the game as fast as possible; *Double AI*, in which two robots play against each other, using expectimax to make decisions, and *Tests*, which provides a unit and integration testing framework.

While the notion of an AI for billiard was not unheard of, it was also not nearly as ubiquitous as in other games, such as Pac-Man. We planned to approach the problem from an unbiased and academic position, and there were certainly many challenges to address. While visual appeal and aesthetics were among be the least important goals of our project, the hardest parts to tackle within the projects short period were the physics portions and the strategy given our platform of choice. That being said, we were surprised to have made great progress and have something to show at the end of these few weeks.

Technical Description

Physics

Matter.js, a combined rendering and physics engine for JavaScript applications, served as the backbone for our project. Many of the decisions we made throughout the process were made with our physics engine in mind.

Billiards is a game that can be simulated relatively accurately, as it is a game of precise actions and specifically defined components. It is a continuous game, but it is as calculable as continuous games get. That said, a perfect -or even great- simulation is quite the undertaking. For the scope of this project, we condensed the physical components of the game to fit within the scope and focus of our Artificial Intelligence project.

We opted for strictly 2D environment. This decreased the number of physical variables to keep track of, while still allowing us to closely model ball movement. The downside to this, though, is that we could not model many techniques that involve decisions in 3D space (e.g., hopping, backspin).

While the physics engine we chose, *Matter.js*, proved more than competent in calculating 2D physics, we did encounter some limitations that affected our product. Matter.js makes calculations in discrete time, meaning that at every tick, things are moved, collisions are detected, and forces are applied. However, when you are simulating a game that usually runs on continuous time, you run into some issues. When two balls collide, the angle at which they reflect is dependent upon their point of contact. In discrete time, this point of contact may be passed over between ticks, resulting in late collision detection and unpredictable final resting states. We tried to limit the effect of this by limiting speed and decreasing the time-scale, but ultimately these kind of collision errors were inevitable and had to be treated as a stochastic variable. We had also made significant-enough progress at this point that switching to another physics engine was not feasible.

GUI

Our Graphical User Interface, defined in `gui.js`, is fairly straightforward. Since *Matter.js* has a built in renderer, physical Bodies can be calculated and drawn fairly easily. Since we did not implement human player mechanics to the simulation, the UI was fairly simple. Our screen consists of the table, turn indicators, scoring, and some simple game options. We represent solid balls with a solid color and a black outline, and striped balls as white balls with colored outlines.

Our GUI class also acts as a means of communication between the *Game Logic* and the *Physics* blocks. The GUI has listeners that will alert *Game Logic* that the balls have reached equilibrium, allowing the next player to take her turn. We also put hooks into the gui for adding or removing balls as necessary.

Game Logic

Our *Game Logic* block, defined in `gamelogic.js`, holds all the rules, game states, player coordination, and win/loss conditions. In its current state, it is optimized for American Pool, but can be adjusted to fit other billiards game types with a moderate amount of effort.

AI Agent

At its core, our `AI Agent`, defined in `ai.js`, uses an expectimax algorithm to determine the best possible shot—taking into account both its own utility and the opponent's utility of the cue ball's final resting place.

This AI Agent very much depends on prediction of future state, which proved quite the technical challenge in our framework. Due the size and architecture of Matter.js, we could not find a way to efficiently and rapidly use it to preserve, copy, and simulate an entire world-state for an indefinite

amount of time and iterations. As a result, we chose to use less robust, but less CPU intensive, solution. Instead of remodeling the entire system, we focus on the interaction of the cue, target ball, and pocket, and ignored all other ball interactions. To compensate for this oversight, we use ray casting to determine whether or not an interaction will occur, and updated our cost function appropriately.

We calculate shot vectors by first picking a target ball and a pocket. To calculate the initial velocity required to propel that ball into the pocket, we use a kinematic equation

$$V_i = (\text{friction} \times \text{distance})/\text{mass}_{\text{ball}}$$

From this velocity, we do a bit of geometric and kinematic manipulation to determine the cue ball's required direction and speed. This kinematic calculation is our substitute for direct simulation. Note that we do not actually invoke the physics engine in this calculation, so any calculation we do only emulates those executed by the engine at runtime. This error will propagate down the expectimax tree, so we will rely on the probability constants to take into account the accuracy of our predictions.

Within our expectimax implementation, we enumerate the best possible shot for each of the agent's balls on the table. For each one of these shots, we create a child node representing a successful shot (the ball was sunk) and a failure shot (the shot was missed). Each child node updates its table configuration accordingly, moving the cue ball to where we expect it to end up. For successful child nodes the current agent shoots again, and for failure nodes the the expectimax algorithm switches to the opponents perspective. When calculating the score of each shot, we multiply the value of each shot by the probability that it will happen. We calculate probability based on the angle of the shot, i.e. the more straight on the shot is, the higher the probability it will go in. Our evaluation function simply counted the number and type of balls on the table. A good state has a smaller number of the player's ball on the table, and a higher number of the opponent's balls on the table. In our implementation, we run expectimax with a depth of 2 (looking ahead 2 shots). Since our estimation of future nodes is fairly inexact, we did not want to explore too deep into the expectimax tree.

Validation

In order to facilitate test-driven development as well as be able to troubleshoot specific issues and have a way to track regressions in our code, we needed to develop a framework for testing. Since our code was a web app written with JavaScript, we needed some way of making assertions on the results of the game.

In particular, we wanted to be able to have a command-line utility that we could invoke to run our tests. Typically to achieve this in the industry, something known as a *headless browser* is used, and *PhantomJS* is a prime example. *PhantomJS* lets developers load a web page into memory - without a browser actually being visible - and perform operations with it. In order to perform the actual testing assertions, we decided to use *CasperJS* for its ease of use and JUnit-like testing framework.

We started out by creating a tests directory for our tests. Each test has its own HTML file. Using *RequireJS*, we can modify the prototypes of any functions we define in order to write the test; for instance, in `single_ai_8ball.html`, we override the `GUI.prototype.setupRack` function in order to only add a single 8-ball at a specified location.

Within the tests directory, there is also a file called *run.js* as well as another subdirectory with test-related code. This file can be run with *CasperJS* by invoking `casperjs test run.js`. In *run.js*, we then perform an assertion that the AI pockets the ball within 60 seconds, and watch for any run-time errors as well. Game end state is indicated by augmenting the GUI.prototype.endGame function using the helper function `TestUtils.notifyOnEndGame` to append a `gameEnded` element to the document once the game ends; *run.js* is then able to check if the document contains this element to know if it should stop.

Results

Was it a success?

Our AI Agent has succeeded in reaching our goal of completing a game of pool, but it is far from optimal. Subjectively, our expectimax algorithm seems to choose optimal shots. However, it appears that only about 30% of the attempted shots manage to pocket a ball, making for games that span about 30 shots. The problem is probably the result of collision detection issues mixed with our ideal kinematic predictor. Still this is promising, and with a better state prediction scheme, the expectimax algorithm has potential.

As for this rest of the project, we were able to construct a good-looking pool table as well as implement the foundation for the validation and testing framework. While we didnt have the luxury of time to implement many test cases, the validation framework works as it should. Indeed, if we wanted to, we could integrate it so that all the tests run on every commit to the repository, ensuring that we do not break anything as we perform further development.

Complications and Management

We initially allocated a week to set up our UI and Physics engine. We chose to pursue a Python environment powered by Pygame, but by the end of that week we could not get a version that worked for every member of the group, nor could we decipher much of the documentation. We made the decision to use JavaScript with Matter.js and after some refactoring, we were able to continue work on the AI Agent. The changes to our timetable are detailed as follows:

- April 11 to 15 - Working Physics, Game Logic, and GUI Display
- April 17 to 19 - Basic AI Agent Logic
- April 20 to 21 - Optimize and Validate AI
- April 23 - Deliverable Product

The performance of each block of our system is definitely correlated to the time attributed to them. While our product looks beautiful, our AI does not quite live up to our expectations and, although the validation framework is complete, the set of tests performed is limited.

We approximated distributed effort as follows:

	Alex	Dave	Oleg	Xingchen
Physics/GUI	20%	40%	20%	20%
Logic	30%	30%	15%	25%
AI	20%	60%	10%	10%
Validation	5%	5%	85%	5%

Table 1: Work Distribution

Extensions

With ample time, the first improvement to be made would probably be more robust state prediction. This would make our AI Agent a much more formidable opponent. There are also a few tricks we wanted to implement to make banking and chain-shots more effective. For example, table-mirroring allows you to easily find perfect vectors for banking the cue off of a wall. From there, we could add in human interaction and possibly more gametypes.

With unlimited resources, our AI algorithm could potentially be adapted to power an autonomous pool playing robot. Swapping out the GUI/Physics Engine for a robotic interface could allow the AI to grab state information from sensor data and make decisions that are translated into articulated movements.