Dartmouth College
Dartmouth Digital Commons

Dartmouth College Undergraduate Theses | Theses and Dissertations

5-1-2003

Billiards Adviser as a Search in a Continuous Domain with Significant Uncertainty

Thomas Mueller
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation
Mueller, Thomas, "Billiards Adviser as a Search in a Continuous Domain with Significant Uncertainty" (2003). *Dartmouth College Undergraduate Theses*. 27.
https://digitalcommons.dartmouth.edu/senior_theses/27

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Billiards Adviser as a Search in a Continuous Domain With Significant Uncertainty

Thomas Mueller
Senior Honors Thesis
Adviser: Zack Butler

Department of Computer Science
Dartmouth College

Technical Report TR2003-448

Abstract

Typical game search algorithms are limited to problems in which there is a certain number of moves for any given state, and the effect of each move is well known. In order to overcome this limitation, we consider the problem of determining the optimal shot given the positions of balls on a billiards table. Our solution includes the image recognition necessary to determine each ball's position, the calculation of the optimal shot, and the presentation of that shot to the player. The focus of the paper is on the second part — determining the angle and force with which the player should attempt to hit the cue ball for each shot in order to sink all of the other balls with the fewest shots. The solution to this problem is unique from other game search algorithms in that it must take into account the infinite number of possible shots given any configuration of balls as well as the fact that the player is not likely to hit the ball exactly how he attempts to do so. We compare the performance of our algorithm with one that ignores the latter fact to show that our modifications do in fact improve performance for a search in a continuous domain with significant uncertainty.

1 Introduction

In the field of artificial intelligence, much research has been done involving the topic of game-playing algorithms. These algorithms search over a set of possible moves in order to determine the optimal move given a certain state. However, the problems to which these algorithms are applied are limited to games in which there is a finite number of possible moves per state, and the effect each move will have on the state is well known. We attempt to modify a depth-limited search algorithm in order to apply it to problems in which this is not the case — ones in which we must search an infinite set of operators in a continuous domain with significant amount of uncertainty as to the effect of any given operator.

To solve this problem, we present a solution to the problem of determining the optimal shot in a modified version of the game of billiards and presenting that shot to the player. The game starts with a cue ball and five other balls placed in random positions on the table. The goal is to hit all of the non-cue balls into the pockets while making as few shots as possible. If the cue ball is sunk, it is placed in a random spot on the table, and the game continues.

Previous work involving the game of billiards has focused on training players to make more accurate shots [2], the mechanics of a robot designed to hit a given ball into a pocket [4], and using wearable computing as a method to display shots to a player [1]. The focus of this paper is on the actual determination of the optimal shot, taking into account the uncertainty of the angle and force with which the player will actually hit the ball when attempting a shot.

The algorithm used for determining the optimal shot builds on a depth-limited algorithm that is often for problem solving used in artificial intelligence. This algorithm involves a set of operators used to build a search tree, whose states are given a heuristic value which is used to determine the optimal shot. There are two aspects of our problem that distinguish it from the problems to which this algorithm is typically applied. First, there is a significant amount of uncertainty in the state that will be reached after applying a certain operator (in this case, a shot's attempted angle and force). Second, we are operating in the continuous domain of the physical world, so there is an infinite number of operators (angles and forces with which to hit the cue ball) that can be carried out from any given state.

The solution we reach can be generalized to many different applications. Obviously it could be applied to other problems involving billiards and other games, but it could also be applied to many other problems involving searches in a continuous domain with significant uncertainty. For example, a hopping robot [5] could use a similar algorithm to determine how to repeatedly propel itself off the ground in order to remain upright. This robot already has a planning model based on the continuous domain of the physical world, but an algorithm that takes uncertainty into account in its planning may increase its performance.

In the first section of this paper, we describe the overall setup of the problem and our solution. We then present our algorithms for image recognition, the game search, and

parameter tuning. Finally, we describe our results and some possible future extensions to this work.

2 Setup

2.1 Physical Configuration

A video camera with a wide-angle lens is mounted to the ceiling pointing downward toward a standard billiards table. This camera is connected via S-Video to an SGI Indy with capture capabilities.

2.2 Software Configuration

The capture computer acts as a webcam for another computer, which grabs individual images via HTTP and performs the actual calculations. The latter then displays the optimal shot as an on-screen animation. All of the software is written in Java in order to obtain maximum portability and maintain the possibility of alternative configurations.

2.3 Technical Problems and Resolutions

We ran into certain difficulties due to our physical setup that caused problems with our image recognition. These problems could certainly be solved using more sophisticated image recognition, but since the focus of this paper is on the planning algorithm we chose instead to work around them.

Our first difficulty stems from the fact that the billiards table is in a room with three large windows. Unfortunately, during the day, sunlight casts shadows of trees all over the table. These moving shadows make image recognition far more difficult, and successful operation therefore requires either working at night or covering the windows.

The low resolution and quality of the captured images makes distinguishing between some balls extremely difficult. For example, it is next to impossible even for a human to differentiate the cue ball and the one ball by looking at one of the images. As a result, we work only with solid-colored balls, and only with balls whose colors are easily distinguishable.

Due to the relatively small distance between the table and the ceiling, we require a wide angle lens in order to capture the whole table in one image. This results in a slight "fish eye" effect where relative distances from the center of the image are not directly proportional to the corresponding physical distances. This problem is accounted for in software by calculating a ball's position using a formula based on the perceived distance from the center of the table.

3 Algorithms

3.1 Image Recognition

Our image recognition involves a relatively simple two-step algorithm. First, an image of the empty table is compared to an image of the table in its current state to determine each

ball's position. Then the color of each ball in the image is compared to a set of previously captured images to determine which ball is which.

In the first step, we first determine the red, green, and blue values of each pixel in the image. We maintain these values for two images of the table — I_{empty}, the image without any balls on it (a portion of which is shown in **Image 1**), and $I_{current}$, the image representing the current state (a portion of which is shown in **Image 2**). The former is downloaded before the beginning of the game, and the latter is downloaded each time we calculate the optimal shot.

The first calculation involves determining the pixel representing the center of the table, P_{center}, and is facilitated by a circle that is chalked on the center of the table. A set of red, green, and blue values of the square of pixels circumscribing this circle is hard-coded in the software. A set of equally sized squares around the center of $I_{current}$ is searched to find the one whose red, green, and blue values have the minimum difference from those in the hard-coded set. P_{center} is determined to be the center of that square.

The next calculation determines the position of each ball on the table. First, we determine the set of pixels P_{diff} for which the sum of the differences between the red, green, and blue values in I_{empty} and $I_{current}$ are above a certain tolerance (colored red in **Image 3**). We then determine the set of pixels such that a certain percentage of the pixels in a circle, slightly smaller than a billiard ball and surrounding the pixel in question, are in P_{diff} (shown in **Image 4**). This results in several "clumps" of pixels in the center of each ball's position in the image. We then calculate the coordinates of the center of each of these clumps, which corresponds to the center of each ball (shown in **Image 5**). Finally, we calculate the angle and distance from the center of the table and each ball, apply a predetermined formula to the distance, and use these two values to determine the ball's position on the table. We determine each ball's average red, green, and blue values and compare them to stored values in order to find which ball is which.

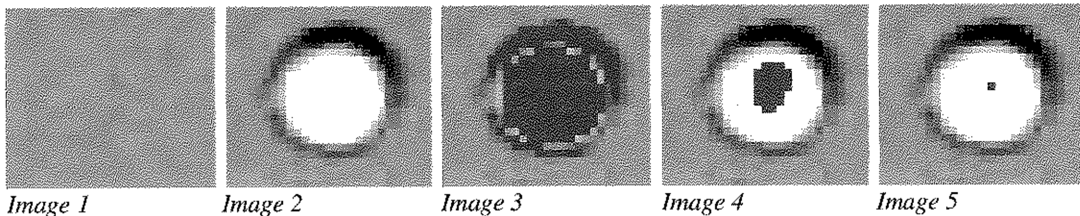

Image 1 *Image 2* *Image 3* *Image 4* *Image 5*

3.2 Physics

The simulation of a shot is computed by a modified version of a billiards game [3] written by Michael O'Shea. The modifications allow us to pass a set of ball positions, an angle, and a force to a function that then returns the ball positions after simulating the corresponding shot. Keeping the physical simulation independent from the rest of the algorithms in this way allows for the possibility of replacing the physics module with a

more accurate or faster simulator if necessary. It also allows us to use the same module to simulate games in order to test the effectiveness of our algorithm.

3.3 Search

This algorithm determines the optimal shot for a given set of ball positions and is the focus of this paper. It is a modified version of a depth-limited search, in which a search tree is generated by applying the operators of each node to the corresponding state in order to determine its children and then applying a heuristic function to the resulting state. There are several parameters involved in the heuristic function, each with a different weight with which it affects the overall score given to the state. After these calculations have been completed, we choose the child of the root node with the highest heuristic value and output the corresponding shot's angle and force. In the first subsection below, we describe what would be a first attempt at laying out this algorithm. We then outline the changes necessary in order to account for our continuous domain and the uncertainty involved in playing the game of billiards. Finally, we describe the algorithm we use in order to determine which weights are best for each parameter in the heuristic function.

3.3.1 Unmodified Algorithm

Our algorithm is a modified version of the following depth-limited search algorithm, which represents a classic approach to problem-solving. Each state consists of how many shots we have taken, the angle and force of each of these shots, whether or not the cue ball is on the table, whether or not each of the other five balls is on the table, and the position of each ball that is on the table. The initial state is with no shots taken and all six balls on the table in unique, non-overlapping, random positions. Each operator is described by an angle and force with which the player will attempt to hit the cue ball. Since the physical simulator we use contains no notion of spin, we ignore the possibility of spinning the ball in different directions. It would be relatively trivial to modify our algorithm to take this possibility into account with a physical simulator that includes it. We then build a search tree up to a constant depth with the initial state as the root. Each node's children are the states corresponding to the output of the physical simulator when passed each possible angle and force. We then apply a heuristic function to the nodes to give each a certain score based on the positions of the balls and choose the child of the root node for which this score is the highest. The angle and force corresponding to this node are returned as the optimal shot.

3.3.2 Modifications

There are two reasons why we must modify this algorithm before we can apply it to our problem. First, there is an infinite number of angles and forces with which we can hit the cue ball regardless of its position, resulting in an infinite number of operators per state and therefore an infinitely wide search tree. Second, the player cannot possibly hit the ball with the exact force and at the exact angle as the algorithm determines to be optimal.

As a result, there is a good deal of uncertainty as to the effect of applying any given operator.

In order to take these problems into account, we use the algorithm in **Figure** 1, which incorporates the following modifications. We modify the heuristic function to take into account not only how good the simulated outcome of a shot is, but also how likely our shot is to result in a similar outcome. The value of each state is partially determined by the highest heuristic value of the corresponding node's children (lines 3, 4, 6, and 7) so that a shot that leaves the cue ball in a good position will be given a higher score. The other factors in the heuristic function are the number of operators we would consider taking from that state (line 5; the more choices of shots we have, the better), whether or not the cue ball is on the table (line 8), how many other balls are on the table (lines 9-10), how many balls would remain on the table if we accidentally hit the ball at a set of angles slightly to each side of the attempted angle (lines 11-13), and the force with which we attempt to hit the ball (line 14). Each of these parameters is given a certain weight, because each affects a state's desirability to a different degree. The last two of the above factors are the ones that we use to account for the uncertainty of the shot's outcome. By simulating a few shots with angles slightly different from the attempted angle, we have a better idea what will happen if the player misses by a slight amount. Also, the amount by which the player misses is likely to be higher with a greater attempted force, and if the ball does not go into the pocket, then it is more likely to stay close to the pocket if it is traveling at a lower speed. After the nodes' heuristic values have been calculated, we determine which of the root node's children has the greatest heuristic value and output the angle and force corresponding to that node.

```
Score(depth):
1:   s ← 0
2:   if depth > 0 then
3:        s_best ← 0
4:        for each c ∈ [child nodes] do
5:             s += f_numCandidates
6:             if c.Score(depth – 1) > s_best then s_best ← c.Score(depth – 1)
7:        s += s_best
8:   if [cue ball is on table] then s += f_cue
9:   for i ← 1 to 5 do
10:       if [b_i is on table] then s += f_ball
11:  for a ← 1 to 4 do
12:       s += f_side * (5 - [balls left after hitting at (angle + a/4 )])
13:       s += f_side * (5 - [balls left after hitting at (angle – a/4 )])
14:  s += f_force X ([maximum force] – force)
15:  return s
```

Figure 1: Heuristic function

One possibility for reducing each state's set of potential operators to a finite set would be simply to split the range of angles into a certain number of degrees, perhaps searching every half of a degree, and to search a certain set of forces. This would be extremely inefficient, however, because the search would involve a large number of shots in which the cue ball does not even hit another ball. Instead, we break down the set of potential shots into a set that sink at least one ball in simulation. In order to do this, we first calculate the angle from the center of the cue ball to the center of each other ball, as well as the difference between that angle and the angle at which we would have to hit the cue ball in order to just skim each side of each other ball. This gives us the range of angles that would hit each ball, which we then split evenly into a constant number of angles. We simulate a shot with each of these angles and each of a constant set of forces, and we use the subset of these shots in which a ball is sunk as the state's potential operators. Note that for each of these shots, several similar shots will be simulated as a part of determining its heuristic value. If this set is empty, then the only potential operator is set to be a random angle and force. The algorithm for determining the set of shots to search for a given configuration of balls is shown in **Figure 2**.

Shots(ball positions):
1: $s \leftarrow \{\}$
2: **for** $i \leftarrow 1$ **to** 5 **do**
3: **if** [b_i is on the table] **then**
4: $a_{center} = \tan^{-1}(\frac{y_2 - y_1}{x_2 - x_1})$
5: $a_{diff} = \tan^{-1}(\frac{2 * radius}{distanceTo(b_i)})$
6: **for** $a \leftarrow 0$ **to** 3 **do**
7: **for** $f \leftarrow 1$ **to** 4 **do**
8: $simulateResult(a_{center} + \frac{a}{3} * a_{diff}, f)$
9: **if** [any ball is sunk] **then** s += $(a_{center} + \frac{a}{3} * a_{diff}, f)$
10: $simulateResult(a_{center} - \frac{a}{3} * a_{diff}, f)$
11: **if** [any ball is sunk] **then** s += $(a_{center} - \frac{a}{3} * a_{diff}, f)$
12: **if** $s = \{\}$ **then** $s \leftarrow \{$[random angle], [random force]$\}$
13: **return** s

Figure 2: Algorithm for determining candidate shots

3.3.3 Learning Algorithm

The heuristic function described above uses several different parameters to determine the value of a given state. Each parameter must be given a certain weight so that the more important ones will have more of an effect on a state's calculated value. However, these relative weights are somewhat arbitrary, and hand-tuning them is rather difficult. In order to further tune these parameters, we use the algorithm in **Figure 3**. It starts with the set of hand-tuned parameters, *P*, and creates a new set of parameters, *Q*, whose values are a random amount up to 10% different from those in *P*. It then simulates a game using each set to determine the optical force and angle for each shot and adding or subtracting a random amount up to one degree to the angle for each shot it makes, and it keeps the set of parameters that performed better. It repeats this process a number of times, eventually converging on an optimal set of parameters.

```
TuneParameters(P):
1:    while not finished do
2:        Q ← P
3:        for each parameter q∈Q do
4:            r ← [random number between .9 and 1.1]
5:            q ← q X r
6:        B ← [random set of positions for cue ball and 5 other balls]
7:        s ← [number of shots in a simulated game using P and B]
8:        t ← [number of shots in a simulated game using Q and B]
9:        if t < s then P ← Q
10:   return P
```

Figure 3: Learning algorithm

4 Results

In order to test the performance of our algorithm, we simulated games in software and recorded the number of shots required to sink all of the balls for each game. The simulated games were in two major categories: those testing the hand-tuned parameters and those testing the machine-tuned parameters. These were each split into three sub-categories, in which the simulator added or subtracted a random amount up to 0.5 degrees, 1 degree, or 2 degrees to the suggested angle when making each shot. In each of these sub-categories, 50 random placements of the cue ball and the other five balls were chosen. Two games were then simulated for each of these configurations with our algorithm – one in which the heuristic function ignored the number of candidate shots, how deep it was in the search, the force of the shot, and the simulated outcome of shots to either side of the shot's angle, and one in which these parameters were taken into account. This allowed us to measure the effect of including these parameters in our heuristic function on the performance of our algorithm. Testing with both the hand-tuned parameters and the machine-tuned parameters allowed us to measure the performance of our learning algorithm. Simulating games with different amounts of randomness in each shot showed us how closely the parameters were tuned to a specific skill level. The results of our simulations are shown in in **Table 1**.

Parameters	*Random Degrees*	*Algorithm*	*Total Shots*	*Games Played*	*Shots Per Game*	*Modified Algorithm's Performance*
Hand-Tuned	0.5	Unmodified	562	50	11.24	29.72% Better
		Modified	395	50	7.90	
	1.0	Unmodified	653	50	13.06	17.76% Better
		Modified	537	50	10.74	
	2.0	Unmodified	921	50	18.42	14.88% Better
		Modified	784	50	15.68	
Machine-Tuned	0.5	Unmodified	554	50	11.08	27.08% Better
		Modified	404	50	8.08	
	1.0	Unmodified	726	50	14.52	29.89% Better
		Modified	509	50	10.18	
	2.0	Unmodified	888	50	17.76	4.95% Better
		Modified	844	50	16.88	

Table 1: Results from simulated games

The results of these simulations were very positive. In every case, the algorithm performed better on average when taking into account the randomness of each shot than when ignoring it. There are several other things that can be learned from the simulated games' outcomes. First, the algorithm performed far better with the machine-tuned parameters than with the hand-tuned parameters when there was one degree of randomness. This is not surprising, considering that this was the amount of randomness to which the parameters were tuned. This shows that the algorithm's performance relies greatly on how well its parameters are tuned to a specific player, and demonstrates the importance of the learning algorithm. However, the hand-tuned parameters performed better with different amounts of randomness, whether these amounts were larger or smaller. This shows that parameters that work well for one player will not necessarily work well for another. If the adviser is to be used by many players without tuning its parameters to each individually, then the parameters should be tuned to perform well over a large set of playing abilities, rather than being tuned to just one player's performance.

Our algorithm also performed very well in games played on a physical table. Most of the time, the optimal shot according to our program was the same as the player would have attempted without it. Sometimes it suggested a different shot, however, and in these cases its suggested shot often seemed more reasonable than the shot the player had planned on making. In a few cases, attempting the shot that our program suggested resulted in sinking a different ball than the one at which the player was aiming, which

was impressive. There were cases in which the program's suggested shot did not seem optimal, but these cases were certainly not the norm.

5 Future Work

There are many interesting extensions that could be made to this project. Considering the tremendous effect that the learning algorithm has on the algorithm's performance in simulated games, it would be interesting to apply this algorithm to physically played games so that the heuristic function's parameters' weights could be tuned to best advise an individual's game play. The program is able to output the amount that each parameter affects its decision in choosing the best shot. It would be interesting to modify it to use this information in some significant way, such as outputting its reasons for choosing that shot over another or offering two different suggestions along with their explanations. Improving the image recognition would allow for games with more balls, closer resembling a regular game of billiards. The algorithm could also be extended to work with a two player game, enhancing its intended role as a billiards adviser. Such an extension would involve modifying the algorithm to search for shots in which the cue ball ends up in a place that is good for the player making the shot but bad for the other player. It would be interesting to combine this project with others involving billiards, using a laser [2] to project the best shot onto the table, for example, so that it could be better used to help a player improve his game play. Lastly, similar algorithms could be applied to different problems involving a continuous domain and significant uncertainty to test its effectiveness in the general case.

Acknowledgments

We are grateful to Research Computing at Dartmouth College for providing the computer to perform the image recognition, and especially to Justin Latham for his work in setting up the software on that machine.

References

[1] Tony Jebara, Cyrus Eyster, Josh Weaver, Thad Starner, and Alex Pentland. Stochasticks: Augmenting the Billiards Experience with Probabilistic Vision and Wearable Computers. In *Proceedings of the International Symposium on Wearable Computers*, Cambridge, MA, October 1997.

[2] L. B. Larsen, M. D. Jenson, and W. K. Vodzi. Multi Modal User Interaction in an Automatic Pool Trainer. In *Fourth IEEE International Conference on Multimodal Interfaces*, pages 361-366, Pittsburgh, PA, October 2002.

[3] Michael O'Shea. Billiards.java. http://206.63.61.36/japps/Billiards.java.

[4] Bingchen Qi and Yoshikuni Okawa. Building An Intelligent Mobile Robot For Billiards. In *30th International Symposium on Robotics*, pages 605–612, Tokyo, Japan, October 1999.

[5] Garth Zeglin and Ben Brown. Control of a Bow Leg Hopping Robot. In *IEEE International Conference on Robotics and Automation*, 1998.