Developing AI for a Robotics Billiards Player

15-780 Graduate Artificial Intelligence (S13)

ASHOK K. ELLUSWAMY, The Robotics Institute, Carnegie Mellon University
Andrew ID: aelluswa

1. INTRODUCTION

The past few years saw significant amount of research in the personal robotics space. Slowly but steadily progress is made in areas of perception, manipulation and mobility. These robots must cope with environments that are partially observable, stochastic, dynamic and continuous. Many AI concepts such as probabilistic state estimation, perception, planning, unsupervised learning and reinforcement learning are used by these robots to solve the above problems.

In my pursuit of personal robotics domain, I have chosen to study algorithms and AI methods that would help enable me to contribute towards research in personal robotics space. I'm working on a project (as a requirement for the class 16-662 Robot Autonomy, Prof. Siddhartha Srinivasa) where three other team mates and I are working on the PR2 robot of the Search Based Planning Lab in the Robotics Institute. The aim of the project is to make the PR2 play a simplified version of 8-ball billiards game. This involves perception, cognition, manipulation and motion planning. In addition to this, the mission requires the AI required for the game itself. The latter is the main focus of this project while the former is the focus of the Robot Autonomy class project.

The AI engine guides the robot as to what action to execute next, which shots of the game are suitable for the robot etc. Even though general purpose AI billiards engines exist[1], they do not take in to consideration the constraints and the physical capabilities of the robot. It requires artificial intelligence to select the 'best' shot taking into account the accuracy of the robot, the noise inherent in the domain, the continuous nature of the search space, the difficulty of the shot, and the goal of maximizing the chances of winning.

It would be interesting to study how the constraints of the robot limit the game playable by the robot. In addition it would also be useful to make observations of shot selection and representing shot difficulty and probability of success of the shot.

.

2. BACKGROUND AND SYSTEM DESCRIPTION

2.1 Billiards Physics Simulation

The outcome of any billiards shot depends on the physical interactions between the balls moving and colliding on the table. The physics of billiards are quite complex, as the motion of balls and results of collisions depend on the spin of the ball(s) involved as well as their direction and velocity. Alon Altman's FastFiz[5] is a physics simulator that, given an initial table state and a shot to execute, finds the resulting table state after the shot completes. Simulation results are deterministic, whereas the outcomes of shots made by a human or robot player on a physical table are non-deterministic. To capture this stochastic element, the input shot parameters to FastFiz are perturbed by a noise model at game time. This results in a slightly different shot outcome every time for a given set of input parameters. The noise

model is described in detail later. The AI engine uses FastFiz to build search trees. From a given state, candidate shots are simulated to find successor states. For a sufficiently realistic simulator, shots that work well in simulation will work well in the real world.

2.2 Lever Stick-Cue Stick-Bridge assembly

A special assembly was required in order to make the robot play the game. The PR2 robot's fastest joint is its wrist joint. In order to convert the angular velocity of the wrist into a linear velocity at the cue stick tip, a lever stick was connected to the end of the cue stick using a revolute joint. This also facilitates the robot to apply an impulse force on the ball it strikes rather than a gradual force. The assembly is shown in Fig. 1.

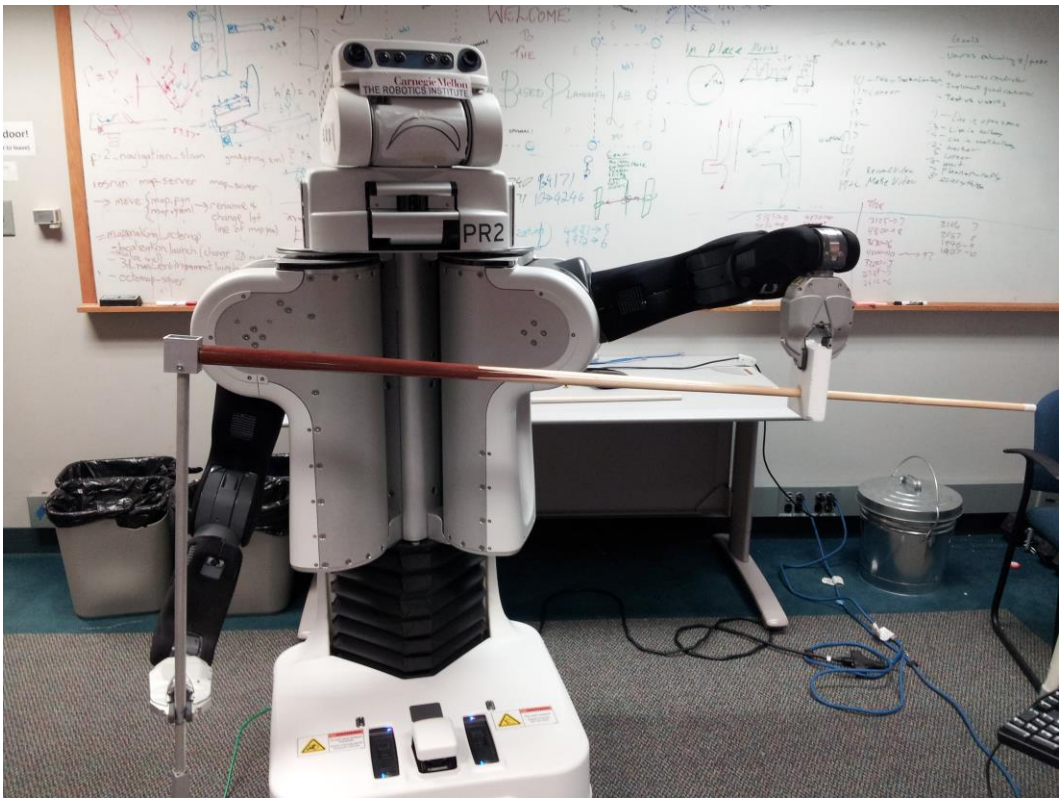

Fig. 1. The custom made assembly for the robot to play a shot. The left gripper holds the bridge, while the right gripper holds the lever stick connected to the cue stick.

2.3 Perception Noise

The robot perceives the table state using its high resolution camera system. The camera was modeled as a pin-hole camera. First the balls were found in the image and by re-projecting a ray through the camera center through the pixel in the image, we get the physical ray of light that caused the pixel. This ray intersects a plane parallel to the billiards table plane, offset from it by the radius of the ball. This point corresponds to the ball's physical location in the world.

This measurement however has a lot of noise to it. In order to account for the noise, an error model was created. Several measurements were taken with the balls in random positions with the PR2 placed at several locations around the table. Based on the data collected, a bivariate Gaussian sensor model was created. The mean of the distribution was 0 and the standard deviation was 5 cm.

3. IMPLEMENTATION OF BILLIARDS AI

3.1 Move Generation

A move generator provides, for a given game state, a set of moves for the search algorithm to consider. For deterministic games like chess, this is often as simple as enumerating all legal moves. For games with a continuous action space, it is impossible to enumerate all moves; a set of the most relevant ones must be selectively generated.

Every billiards shot is defined by five continuous parameters [3], illustrated in Fig. 2:

- φ, the aiming angle,
- V , the initial cue stick impact velocity,
- θ , the cue stick elevation angle, and
- a and b, the x and y offsets of the cue stick impact position from the cue ball center.

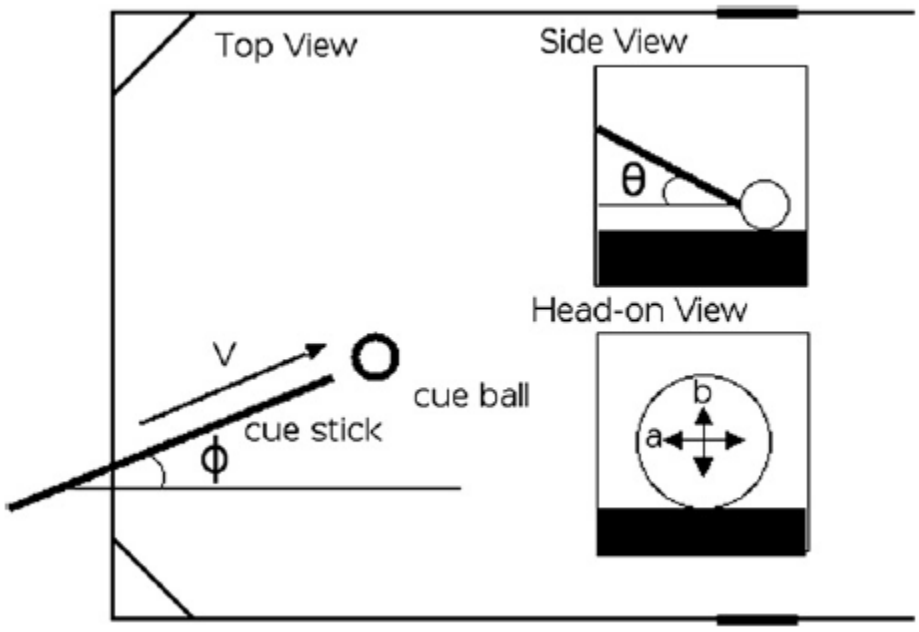

Fig. 2. Parameters defining billiards shot

However, of all the five parameters only the aiming angle φ and the cue stick impact velocity V, are relevant for the robot. The elevation angle θ is fixed at approximately 5° since the joint angles of the robot itself are fixed. The offsets a and b are ignored

because they rarely affect the shot itself and with the current accuracy of the robot, it is not trivial to implement these offsets while taking the shot.

When the current state of the table is given, the move generator generates all the possible shots by discretizing the continuous variables φ and V. The aiming angle was discretized to 0.1° and the velocity was discretized to steps of 0.5 m/s, starting at a minimum velocity.

3.2 Constraints of the Robot

The robot has many constraints that need to be satisfied in order to make a shot. The constraints are shown in Fig 3. There are two main constraints that have to be respected. The robot base constraint is to avoid any collisions of the robot with the table. However the arms remain outstretched and can come into collision. But the fixed joint angles are such that the arm remains above the table's height.

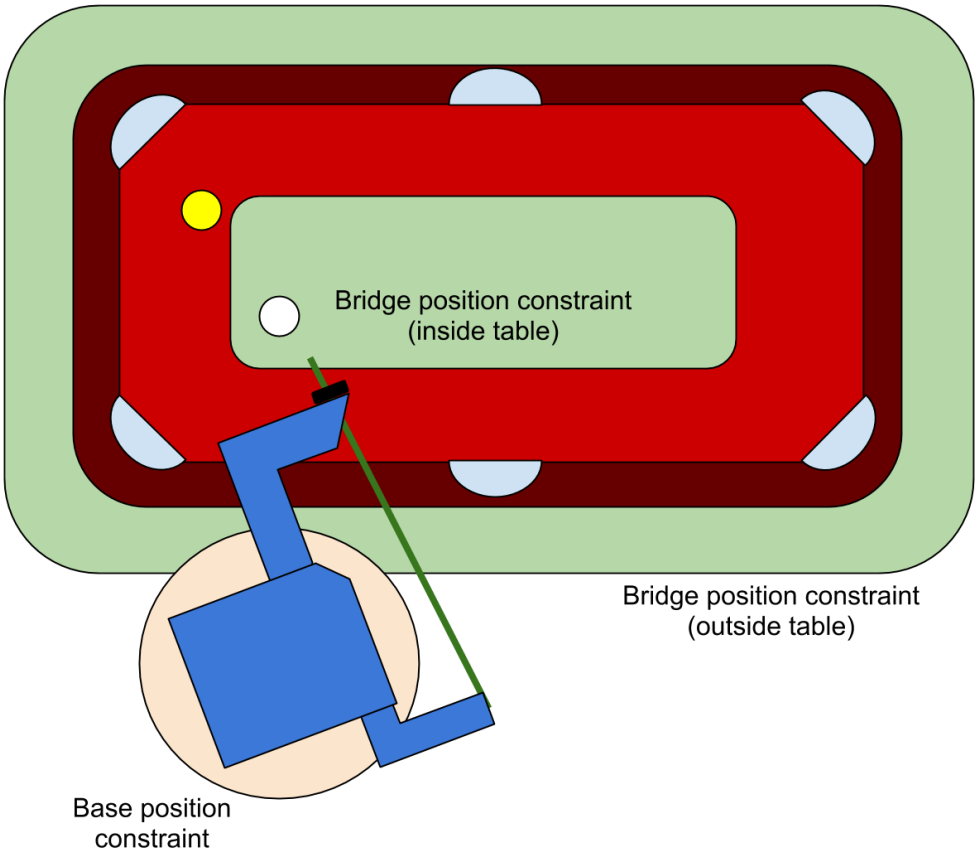

Fig. 3. The constraints of the robot are shown. The light orange circle represents the base radius of the robot. The green areas represent the areas inaccessible to the robot for placing the bridge.

The second constraint is of the bridge. The left arm of the robot holds a bridge in its gripper. The extension or collapsing of the arm limits where the bridge can be placed. The bridge has to be placed on the table in order to stabilize the cue stick. Hence any

placement of bridge outside of the table is not accepted. Similarly any placement of the bridge inside the table but unreachable by the arm is also removed.

Based on these constraints the shots generated before are filtered and only the shots that respect these constraints are considered for the search and evaluation described later.

3.3 Evaluation Function

An evaluation function was developed to assign a cost or reward to each of the shots. When the generated shots are passed to the FastFiz simulator, the simulator gives the resultant state of the table along with the events that occurred during the execution of the shot. The two main events that concern us are potting the object ball and potting the cue ball. Potting the object ball was given a score of +1, whereas potting the cue ball was assigned a score of -10.

In addition to this scoring system, wide angle shots were preferred over normal angle shots. This was easier for the robot to perceive the ball and the bridge thus reducing the uncertainty in the action. Hence the scores shots with wide angles were boosted over the shots with a more normal angle.

3.4 Monte-Carlo Search

A Monte-Carlo sampling is a randomly determined set of instances over a range of possibilities. Their values are then averaged to provide an approximation of the value of the entire range. This makes the vastness of these domains tractable. This suggests sampling is a good candidate for billiards[4].

In this implementation, sampling is done over the range of possible shot outcomes. At each node, for each generated shot, a set of num_samples instances of that shot are randomly perturbed by the noise model, and then simulated. Each of the num_samples resulting table states becomes a child node. The score of the original shot is then the average of the scores of its child nodes. This sampling captures the breadth of possible shot outcomes.

The larger num_samples is, the better the actual underlying distribution of shot outcomes is approximated. However, tree size grows exponentially with num_samples. This results in searches beyond 2 levels being intractable for reasonable values of num_samples. Fig. 4 shows pseudo-code for the Monte-Carlo approach. PerturbShot() randomly perturbs the shot parameters according to the noise model.

In addition to adding noise to the shot parameters itself, noise is also added to the table state according to the sensor model described earlier. This addition of noise to the table state makes the shot computed more immune to the uncertainty in perception of the table state.

The search depth was fixed as one for the shots tried because the strategy was not of concern initially since it was only one person play for our trials. However, the search depth can be increased based on the need.

```
Function Monte_Carlo_Search(TableState state, int depth)
{
        //If it is a leaf node, evaluate and return
        if(depth == 0) return Evaluate(state);

        //else, generate shots for this table state
        shots[] = Move_Generator(state);
        shots[] = Check_Constraints(shots[])

        best_score = -1;
        TableState nextState;
        Shot thisShot;

        // search each generated shot
        foreach(shots[i])
        {
                sample_sum = 0;
                for(k = 1 to num_state_samples)
                {
                sum = 0;
                thiState = PerturbState(state);
                        for(j = 1 to num_shot_samples)
                        {
                                thisShot = PerturbShot(shots[i]);
                                nextState = Simulate(thisShot, thisState);
                         if(!ShotSuccess()) continue;
                         sum += Monte_Carlo_Search(nextState, depth - 1);
                }
                sampleSum += sum / num_shot_samples;
        }
        score = sampleSum/num_state_samples;
        if(score > bestScore) bestScore = score;
        }
        return bestScore;
}
```

Fig. 4. Monte Carlo Search Algorithm

4. PERFORMANCE EVALUATION

When the implementation was complete the entire system was tested using a simulated robot in a simulated environment. The "perceived" table state was passed to the AI engine. It computed the shot with the maximum expected success and returned to the robot. Not only did it present the shot with the maximum expected success, it also returned top ten shots in its search. This way the robot can choose to take the shot which requires it to move the minimum distance in configuration space.

Testing with the real robot has been affected by some issues in perception and navigation and it still ongoing. However, the AI system is ready for integration with the rest of the system. When some manual intervention was provided in perception

sub-system, the robot was able to take the shot computed by the AI engine. Using only one level of search, the search took approximately 100 s to compute for 10 perturbed table states and 10 perturbed shots in each of them.

5. CONCLUSIONS

This article described an adaption of game search techniques to the continuous, stochastic domain of billiards. The program's approach to move generation, constraint filtering, evaluation function, and Monte-Carlo search algorithms were described. There certainly is room for improvement. One obvious improvement is a pre-computed shot table that computes scores for different cue ball object ball combinations. Then the search would actually just lookup rather than online simulation thus reducing search time. However, it has not been implemented since there are more bottlenecks in the robot's other sub-systems.

REFERENCES

[1] Smith, Michael. "PickPocket: A computer billiards shark." *Artificial Intelligence* 171.16 (2007): 1069-1091.

[2] Nierhoff, Thomas, Kerstin Heunisch, and Sandra Hirche. "Strategic play for a pool-playing robot." *Advanced Robotics and its Social Impacts (ARSO), 2012 IEEE Workshop on*. IEEE, 2012.

[3] Greenspan, Michael, et al. "Toward a competitive pool-playing robot." *Computer* 41.1 (2008): 46-53.

[4] Landry, Jean-François, Jean-Pierre Dussault, and Philippe Mahey. "Billiards: an optimization challenge." *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering*. ACM, 2011.

[5] Altman A., Computational Billiards Library (http://www.stanford.edu/group/billiards/FastFiz/)

[6] Willow Garage, The PR2 Robot Plays Pool (http://www.willowgarage.com/blog/2010/06/15/pr2-plays-pool)