Running the Table: An AI for Computer Billiards

Michael Smith
Department of Computing Science, University of Alberta
Edmonton, Canada T6G 2E8
smith@cs.ualberta.ca

Abstract

Billiards is a game of both strategy and physical skill. To succeed, a player must be able to select strong shots, and then execute them accurately and consistently. Several robotic billiards players have recently been developed. These systems address the task of executing shots on a physical table, but so far have incorporated little strategic reasoning. They require AI to select the 'best' shot taking into account the accuracy of the robotics, 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. This paper develops and compares several approaches to establishing a strong AI for billiards. The resulting program, PickPocket, won the first international computer billiards competition.

Introduction

Billiards refers to a family of games played on a billiards table. Players use a cue stick to strike the cue ball into an object ball, with the intent to drive the object ball into a pocket. Common billiards games include 8-ball and snooker.

Billiards games emphasize both strategy and physical skill. To succeed, a player must be able to select strong shots, and then execute them accurately and consistently. Several robotic players have recently been developed, including Deep Green (Long et al. 2004) and Yang's billiard robot (Cheng & Yang 2004). These systems address the task of executing shots on a physical table, but so far have incorporated little strategic reasoning. To compete beyond a basic level, they require AI to select the 'best' shots.

Three main factors determine the quality of a billiards shot. First, it must contribute towards the player's goals. Most shots sink an object ball, allowing the player to shoot again and progress towards clearing the table. Safety shots, giving up the turn but leaving the opponent with few viable shots, are another strategic option. The many potential extraneous shots that perform neither of these have little value. Second, the shot's difficulty is a factor. All else being equal, shots with a high probability of success are preferred. Finally, the quality of the resulting table state after the shot is a factor. A shot that leaves the player well positioned to make an easy follow-up shot on another object ball is preferred.

Skilled human billiards players make extensive use of position play. By consistently choosing shots that leave them well positioned, they minimize the frequency at which they have to make more challenging, risky shots. Strong players plan several shots ahead. The best players can frequently run the table off the break shot. The value of lookahead suggests a search-based solution for a billiards AI. Search has traditionally proven very effective for games such as chess. Like chess, billiards is a two-player, turn-based, perfect information game. Two properties of the billiards domain distinguish it, however, and make it an interesting challenge.

First, it has a continuous state and action space. A table state consists of the position of 15 object balls and the cue ball on a continuous <x,y> coordinate system. Thus, there are an infinite number of possible table states. This renders standard game-tree search enhancements inapplicable. Similarly, each shot is defined by five continuous parameters, so there are an infinite number of possible shots available in any given table state. A set of the most relevant of these must be selectively generated.

Second, it has a stochastic nature. For a given attempted shot in a given state, there are an infinite number of possible outcomes. The player can visualize their intended shot, but will always miss by a small and effectively random delta amount. A professional player, trained for accuracy, will tend to have small deltas, whereas casual players will exhibit larger deltas. Similarly, for a robotic player, deviations from the intended shot arise from limitations in the accuracy of the vision and control systems. Ambient environmental factors such as temperature and humidity can also affect collision dynamics, leading to variance in shot outcomes. This stochastic element means that a deterministic expansion of a move when building a search tree, as is done in chess, is insufficient to capture the range of possible outcomes.

Search has been applied to stochastic games such as backgammon and scrabble. However, these games are simpler than billiards in that they have a discrete state and action space. Search has also been investigated for continuous real-time strategy (RTS) games (Chung, Buro, & Schaeffer 2005). Billiards is more constrained than these because of its rigid turn-based structure. Billiards players take turns shooting, whereas in RTS games players move simultaneously. Billiards players also have the benefit of having a static table state which they can spend time analyzing before

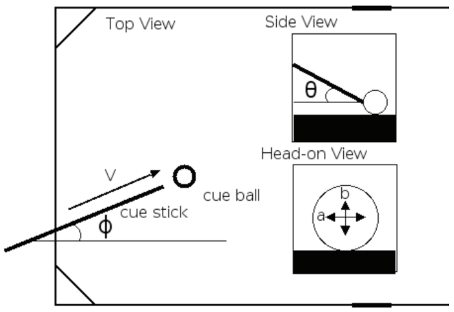

Figure 1: Parameters defining a billiards shot

selecting a shot. In contrast, the environment in an RTS is constantly changing. As a perfect information, turn-based, continuous, stochastic game, billiards bridges the complexity gap, bringing together elements of traditional deterministic perfect information games like chess, stochastic games like backgammon, and realtime continuous games.

There is a family of turn-based continuous stochastic games, which include croquet, lawn bowling, shuffleboard, and curling. These have not been previously examined from an AI perspective. The techniques and considerations discussed here for billiards should carry over to these domains.

Any search-based game-playing program consists of three main components: a move generator, an evaluation function, and a search algorithm. In this paper, we discuss the adaptation of each of these to a stochastic continuous domain, and describe our billiards implementation. Two search algorithms are presented: probabilistic search (a special case of expectimax) and Monte-Carlo sampling search. These algorithms have offsetting strengths and weaknesses, representing the classic trade-off between of breadth vs. depth in search. Experimental results are given showing the strength of these search algorithms under varying error conditions.

Move Generation

A move generator provides, for a given game state, a set of moves to consider. 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.

In billiards, shots vary in their difficulty. Shots range from ones players rarely miss, such as tapping in a ball in the jaws of a pocket, to very challenging, such as a long bank shot off a far rail. This difficulty is a key property of the shot itself, and thus must be captured by the move generator. With every shot generated, it must provide an assessment of its difficulty. This is needed by both the search algorithm and the evaluation function.

Every billiards shot is defined by five continuous parameters: ϕ, the aiming angle; V, the 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 centre. Figure 1 illustrates these parameters.

Shots that accomplish the goal of sinking a given object ball into a given pocket can be divided into several classes. In order of increasing difficulty, they are: The straight-in shot, where the cue ball directly hits the object ball into the pocket; the bank shot, where the object ball is banked off a rail into the pocket; the kick shot, where the cue ball is banked off a rail before hitting the object ball into the pocket; and the combination shot, where the cue ball first hits a secondary object ball, which in turn hits the target object ball into the pocket. These can be combined to arbitrary complexity. Difficulty increases with each additional collision.

Consider a straight-in shot. As a result of billiard ball dynamics, ϕ largely determines the shape of the shot up until the cue ball's first collision with the target object ball. This object ball will have a similar post-collision trajectory regardless of the values of the other parameters. However, the cue ball's post-collision trajectory can be altered by varying V, a, and b, which affect the cue ball's spin at the time of collision. V at the same time affects the distance travelled by the cue and object balls. It must be sufficiently large to sink the desired object ball, while variations above this threshold determine how far the cue ball travels post-collision. θ is constrained by having to be large enough that the cue stick is not in collision with either any object balls on the table or the rails around the table's edge. High θ values impart curvature on the cue ball's initial trajectory.

PickPocket generates shots one class at a time, starting with straight-in shots. For every legal object ball, for every pocket, a straight-in shot sinking that object ball in that pocket is considered. Sometimes this shot is not physically possible. When the shot is possible, ϕ is chosen such that the object ball is aimed at the exact centre of the pocket. V is retrieved from a precomputed table of minimum velocities necessary to get the object ball to the pocket. θ is set to a minimum physically possible value. a and b are set to zero.

This generates one shot sinking the target ball in the target pocket. An infinite set of these could be generated by varying the shot parameters, especially V, a, and b, such that the altered shot still sinks the target ball in the target pocket. Each variation on the shot leaves the table in a different follow-up state. For position play, it is important to generate a set of shots that captures the range of possible follow-up states. PickPocket discretely varies V, a, and b to generate additional shots. For example, V is increased in $1m/s$ increments up to the maximum $4.5m/s$.

If straight-in shots are found, move generation is complete. If not, PickPocket falls back on the other shot classes in order of increasing complexity until shots are found.

Shot Difficulty

To model shot outcomes, the *poolfiz* physics simulator (Leckie & Greenspan 2005) is used. Given an initial state and a set of shot parameters, *poolfiz* applies dynamics equations to find the resulting state. Under simulation, the result of a particular shot is deterministic. To capture the inaccuracies inherent in playing on a physical table, an error model is applied. This randomly perturbs the shot parameters before they are input to the simulator, resulting in a slightly different shot every time. The error model is described later.

The difficulty of a straight-in shot is a function of several parameters. A subset of these depend entirely on the position of the object ball and cue ball, independent of the rest of the table state. These are: the cut angle α, the object

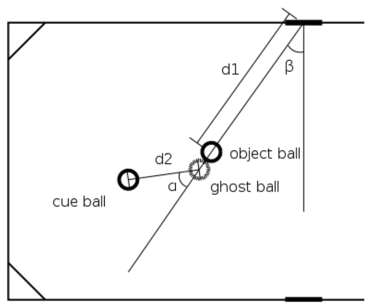

Figure 2: Shot difficulty parameters

ball-pocket distance $d1$, the cue-object ball distance $d2$, and object ball-pocket angle β (Figure 2). α and $d2$ are calculated relative to a ghost ball, representing the position of the cue ball when it impacts the object ball. The function is also different between shots into corner and side pockets, due to differing interactions between the object ball and rails.

In previous work, fuzzy (Chua et al. 2002) and grey (Lin & Yang 2004) logic have been used to approximate this function. PickPocket uses a different approach, taking advantage of the *poolfiz* simulator to capture the shot difficulty function in a table. The table is filled with accurate approximations of absolute probability values. Previous techniques generated arbitrary, relative values.

PickPocket precomputes a table to capture the difficulty function, as the calculation is too costly to perform for each shot generated at runtime. The shot difficulty parameters are discretized and sampling is used to fill each table entry. For each set of parameter values $\{\alpha, d1, d2, \beta\}$, a table state and shot are generated. The shot is simulated s times in *poolfiz*, and the percentage of these that the ball is successfully pocketed is recorded in the table. PickPocket uses $s = 200$. At runtime, a table lookup for each generated shot provides a quick estimate of its probability of success.

In a game situation, the actual probability of success of a shot depends on dynamic factors that cannot be captured in this lookup table. Other object balls on the table can interfere with the shot when they are near the intended trajectories of the object and cue ball. The exact value of the five shot parameters also has a small effect on success chances.

The granularity of the discretization used for the table has an impact on its accuracy, as well as its memory footprint and computation time. Thus it must be chosen with these factors in mind. PickPocket builds $30 \times 30 \times 30 \times 30$ tables.

Bank, kick, and combination shots have too many parameters to construct a success probability table of manageable size. For each collision there is an additional distance and angle parameter. To assign a probability value to these shots, each one is mapped to a corresponding straight-in shot. A discount factor is applied to the straight-in probability to account for the additional complexity of the shot class.

Safety Shots

With a safety shot the goal is not to sink a ball, but rather to leave the opponent with no viable shots. Ideally the opponent will then give up ball-in-hand,[1] leaving the player in a strong situation. Unlike the previously discussed shot classes, there is no way to generate a safety directly from parameters. What makes a good safety is wholly dependent on the table state. The goal is abstract (leave the table in a 'safe state') rather than concrete (sink ball x in pocket y).

One way to account for the entire table state is the use of sampling. For safety shots, a wide range of ϕ and V values are sampled, leaving θ, a, and b alone to make the sampling space manageable. For each set of ϕ and V, a shot with these parameters is sampled i times, evaluating the resulting state from the opponent's perspective. The overall value of this shot is then the average of these evaluations.

Since sampling is a costly operation, if safeties were generated per-node then the cost of searching would quickly become excessive. To get around this, safety shots are only considered at the root. Sampling as a one-time cost has a relatively minor impact on performance. At the root, if the best shot from the search has an estimated success probability below a threshold t_0, safety shots are generated. If the value for the opponent of the best safety is below another threshold t_1, this shot is selected instead of the search result shot. The thresholds t_0 and t_1 can be adjusted to alter the program's safety strategy. PickPocket uses $t_0 = 65\%$ and $t_1 = 0.5$ (evaluation values range from 0 to 1.48).

Evaluation Function

An evaluation function generates, for a game state, a value corresponding to the worth of that state for the player to act. In search, the evaluation function is applied to the game states at the leaf nodes, and the generated values are propagated up the tree.

In PickPocket's billiards evaluation function, the dominant term is related to the number and quality of the shots available to the player. This is similar to the mobility term used in games like chess, extended to account for the uncertainty of the stochastic domain. Intuitively, the more high success probability shots available to the player, the more ways he can clear the table without giving up the turn. Even if there is no easy path to clearing the table, the more good options the player has, the greater the chances are that one of them will leave the table in a more favourable state. Similarly, the higher the probability of success of the available shots, the more likely the player is to successfully execute one and continue shooting. Unintentionally giving up the shot is one of the worst outcomes in all billiards games.

To implement this term, the move generator is used to generate shots for the state being evaluated. These shots are sorted by their probability estimates, highest first. Duplicate shots for the same ball on the same pocket are eliminated, as these all have the same estimate. The first n shots are considered, and the function $d_1 p_1 + d_2 p_2 + d_3 p_3 + ... + d_n p_n$ is applied. d_n is the discount factor for the nth shot and p_n is the estimated probability for the nth shot. Values are chosen for each d_n such that they decrease as n increases.

The discount factor is applied to account for diminishing returns of adding additional shots. Consider two situa-

[1]When a player fails to execute a legal shot, his opponent gets *ball-in-hand*, and may place the cue ball anywhere on the table.

tions for a state: three shots with 90%, 10%, 10% success chances, and three shots with 70%, 70%, and 70% chances. These are of roughly equal value to the player, as the former has an easy best shot, whereas the latter has several decent shots with more options for position play. With equal weighting, however, the second would evaluate to nearly twice the value of the first state. Applying a discount factor for shots beyond the first maintains a sensible ordering of evaluations. PickPocket uses $n = 3$, $d_1 = 1.0$, $d_2 = 0.33$, and $d_3 = 0.15$. These weights have been set manually, and could benefit from tuning by machine learning.

Search Algorithms

A search algorithm defines how moves at a node are expanded and how their resulting values are propagated up the resulting search tree. For traditional games like chess, $\alpha\beta$ is the standard algorithm. For stochastic games, the search algorithm must also account for inherent randomness in the availability or outcome of actions. In billiards, players cannot execute their intended shots perfectly. The outcome of a given shot varies, effectively randomly, based on the accuracy of the shooter. For any stochastic game, the search algorithm should choose the action that has the highest expectation over the range of possible outcomes.

When searching billiards, a physics simulation is used to expand the shots available at a node to the next ply. The per-node overhead of simulation reduces the maximum tree size that can be searched in a fixed time period. Whereas top chess programs can search millions of nodes per second, PickPocket searches hundreds of nodes per second.

Expectimax, and its *-Minimax optimizations, are natural candidates for searching stochastic domains (Hauk 2004). In expectimax, chance nodes represent points in the search where the outcome is non-deterministic. The value of a chance node is the sum of all possible outcomes, each weighted by its probability of occuring. This approach does not apply directly to billiards, as there is a continuous range of possible outcomes for any given shot. The chance node would be a sum over an infinite number of outcomes, each with a miniscule probability of occuring. To practically apply expectimax, similar shot results have to be abstracted into a finite set of states capturing the range of plausible outcomes. In general, abstracting billiards states in this way is a challenging unsolved problem.

A simple abstraction that can be made, however, is the classification of every shot as either a success or failure. Either the target object ball is legally pocketed and the current player continues shooting, or not. From the move generator, p_s, an estimate of the probability of success, is provided for every generated shot. Expectimax-like trees can be constructed for billiards, where every shot corresponds to a chance node. Successful shots are expanded by simulation without applying the error model. For a shot to succeed, the deviation from the intended shot must be sufficiently small for the target ball to be pocketed, so the outcome table state under noisy execution should be similar to the outcome under perfect execution. For unsuccessful shots, there is no single typical resulting state. The deviation was large enough that the shot failed, so the table could be in any state after the shot. To make search practical, the value of a failed shot is set to zero. This avoids the need to generate a set of failure states to continue searching from. It also captures the negative value to the player of missing their shot.

```
float Prob_Search(TableState state, int depth){
  if(depth == 0) return Evaluate(state);
  shots[] = Move_Generator(state);
  bestScore = -1; TableState nextState;
  foreach(shots[i]){
    nextState = Simulate(shots[i], state);
    if(!ShotSuccess()) continue;
    score = shots[i].probabilityEstimate
            * Prob_Search(nextState, depth - 1);
    if(score > bestScore) bestScore = score;
  }
  return bestScore;
}
```

Figure 3: Probabilistic search algorithm

Probabilistic search, an expectimax-based algorithm suitable for billiards is shown in Figure 3. It has a `depth` parameter, limiting how far ahead the player searches. `Simulate()` calls the physics library to expand the shot, without perturbing the requested shot parameters according to the error model. `ShotSuccess()` checks whether the preceding shot was successful in pocketing a ball.

There are three main drawbacks to this probabilistic search. First, the probability estimate provided by the move generator will not always be accurate, as discussed earlier. Second, not all successes and failures are equal. The range of possible outcomes within these two results is not captured. Some successes may leave the cue ball well positioned for a follow-up shot, while others may leave the player with no easy shots. Some failures may leave the opponent in a good position to run the table, whereas some may leave the opponent with no shots and likely to give up ball-in-hand. Third, as the search depth increases, the relevance of the evaluation made at the leaf nodes decreases. Expansion is always done on the intended shot with no error. In practice, error is introduced with every shot that is taken. Over several ply, this error can compound to make the table state substantially different from one with no error. The player skill determines the magnitude of this effect.

Sampling is a second approach to searching stochastic domains. 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. Monte-Carlo techniques have been applied to card games including bridge and poker, as well as go. The number of deals in card games and moves from a go position are too large to search exhaustively, so instances are sampled. This makes the vastness of these domains tractable. This suggests sampling is a good candidate for billiards.

In PickPocket, 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 error 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

```
float MC_Search(TableState state, int depth){
  if(depth == 0) return Evaluate(state);
  shots[] = Move_Generator(state); bestScore = -1;
  TableState nextState; Shot thisShot;
  foreach(shots[i]){
    sum = 0;
    for(j = 1 to num_samples){
      thisShot = PerturbShot(shots[i]);
      nextState = Simulate(thisShot, state);
      if(!ShotSuccess()) continue;
      sum += MC_Search(nextState, depth - 1);
    }
    score = sum / num_samples;
    if(score > bestScore) bestScore = score;
  }
  return bestScore;
}
```

Figure 4: Monte-Carlo search algorithm

of possible shot outcomes. There will be some instances of successes with good cue ball position, some of successes with poor position, some of misses leaving the opponent with good position, and some of misses leaving the opponent in a poor position. Each instance will have a different score, based on its strength for the player. Thus when these are averaged, the distribution of outcomes will determine the overall score for the shot. 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-ply being intractable for reasonable values of $num\_samples$.

Figure 4 shows pseudo-code for the Monte-Carlo approach. `PerturbShot()` randomly perturbs the shot parameters according to the error model.

Generally, Monte-Carlo search is strong where probabilistic search is weak, and vice versa. Monte-Carlo search better captures the range of possible outcomes of shots, but is limited in search depth. Probabilistic search generates smaller trees, and therefore can search deeper, at the expense of being susceptible to error.

Both probabilistic and Monte-Carlo search algorithms can be optimized with $\alpha\beta$-like cutoffs. By applying move ordering, sorting the shots generated by their probability estimate, likely better shots will be searched first. Cutoffs can be found for subsequent shots whose score provably cannot exceed that of a shot already searched.

Experiments

Although the results of shots on a physical table are stochastic, simulator results are deterministic. To capture the range of shot results on a physical table, a random element is introduced into the simulation. In *poolfiz*, error is modeled by perturbing each of the five input shot parameters by zero-mean gaussian noise. A set of standard deviations $\{\sigma_\phi, \sigma_\theta, \sigma_V, \sigma_a, \sigma_b\}$ corresponding to the noisiness of the five parameters is specified. These σ values can be chosen with the properties of the player being simulated in mind. For a robot, σ values can be approximated experimentally.

PickPocket plays 8-ball, the game selected for the first computational billiards tournament. In 8-ball, each player is assigned a set of seven object balls: either solids or stripes. To win, the player must pocket their entire set, followed by the 8-ball. If a player's shot pockets the 8-ball prematurely, they suffer an automatic loss. Players must call their shots by specifying which object ball they intend to sink in which pocket. A player continues shooting until they fail to legally pocket a called object ball, or until they declare a safety shot.

	E_{high}			E_{low}		
Match	W	SIS	SS	W	SIS	SS
Greedy	13	246/451=55%	25/36	14	230/313=73%	27/77
Prob	37	308/496=62%	25/40	36	313/349=90%	42/112
Greedy	2	187/350=53%	25/32	0	77/115=70%	4/28
MC	48	326/481=68%	9/16	50	350/374=94%	4/17
Prob	12	205/319=64%	10/21	5	147/176=84%	9/24
MC	38	306/414=74%	9/14	45	334/360=93%	12/39

Table 1: Comparison of search algorithms

Other requirements for building an 8-ball AI are choosing a break shot and handling ball-in-hand situations. PickPocket has additional routines to handle these tasks.

Experiments were constructed to compare the search algorithms used by PickPocket. A tournament was played between the following versions of the program:

- Greedy: This baseline algorithm runs the shot generator for the table state, and executes the shot with the highest probability estimate. No search is performed. Greedy algorithms are used to select shots in both (Chua et al. 2002) and (Lin & Yang 2004).
- Prob: The 4-ply probabilistic search algorithm.
- MC: The Monte-Carlo search algorithm with $num\_samples = 15$, searching to 2-ply depth.

Prob and MC parameters were chosen such that a decision was made for each shot within 60 seconds. This is a typical speed for time-limited tournament games.

To compare purely search algorithms, other factors influencing performance were simplified. The move generator was set to generate only straight-in shots. Bank, kick, and combination shots were disabled. These occur rarely in game situations; usually there are straight-in shots available. Safety thresholds were set at $t_0 = 0\%$ and $t_1 = 1.48$, so safeties were played only when no other shots could be generated.

Fifty games were played between each pair of algorithms, under two different error models. The first, E_{high}, has parameters $\{0.185, 0.03, 0.085, 0.8, 0.8\}$. This models a strong amateur, who can consistently pocket short, easy shots, but often misses longer shots. The second, E_{low}, has parameters $\{0.0185, 0.003, 0.0085, 0.08, 0.08\}$. This was chosen to model a highly accurate professional player.

Table 1 shows the tournament results. As well as wins (W) for each match, the breakdown of players' attempted shots is given. SIS is the ratio of successful to attempted straight-in shots. A successful straight-in shot sinks the called object ball; the player continues shooting. SS is the ratio of successful to attempted safety shots. A successful safety is one where the opponent fails to pocket an object ball on their next shot, thus the player gets to shoot again. In this tournament, every shot is either a straight-in attempt, a safety attempt, or a break shot to begin a new game.

Both search algorithms defeated Greedy convincingly. This demonstrates the value of lookahead in billiards. Greedy selects the easiest shot in a state, without regard for the resulting table position after the shot. The search algorithms balance ease of execution of the current shot with potential for future shots. Thus, they are more likely to have easy follow up shots. This wins games.

Under each error model, the algorithms vary in their percentage of completed straight-in attempts. This highlights the differences in position play strength between the algorithms. Since the same error model applies to all algorithms, they would all have the same straight-in completion percentage if they were seeing table states of equal average quality. Lower completion rates correspond to weaker position play, which leaves the algorithm in states that have a more challenging 'best' shot on average. Completion rates consistently increased from Greedy to Prob to MC.

Under E_{high}, the games tended to be longer, as the lower accuracy led to more missed shots. Under E_{low}, matches completed faster with fewer misses. The change in straight-in completion rate for a given algorithm between the two error models represents this change in accuracy. Winning programs take more shots than losing programs, as they pocket balls in longer consecutive sequences.

In 8-ball, since a player may aim at any of his assigned solids or stripes, there are usually straight-in shots available. Safeties, attempted when no straight-in shots could be generated, totalled roughly 10% of all shots in the tournament. Therefore at least one straight-in shot was found in 90% of positions encountered. This demonstrates the rarity of opportunities for bank, kick, and combination shots in practice, as they would be generated only when no straight-in shots are available. Even then, safety shots would often be chosen as a better option. Safeties were effective under E_{high}, frequently returning the turn to the player. They were much less effective under E_{low}, as the increased shot accuracy led to there being fewer states from which the opponent had no good straight-in shots available.

MC is clearly the stronger of the search algorithms. Under both error models, it defeated Greedy by a wider margin than Prob, and then defeated Prob convincingly in turn. This suggests that the value of sampling and taking into account the range of possible shot outcomes outweighs the benefit of deeper search under a wide range of error models.

The $num\_samples$ used for MC search, and the search depth selected for both MC and Prob, contribute to their performance. While they were held fixed for these experiments, varying them has been seen to impact results. 2-ply search performs better than 1-ply search. There is diminishing returns for each additional ply of depth added for Prob. For MC, larger values of $num\_samples$ improve performance, but are also subject to diminishing returns.

Computer Olympiad 10

PickPocket won the first international computational 8-ball tournament at the 10^{th} Computer Olympiad (Greenspan 2005). Games were run over a *poolfiz* server, using the E_{high} error model detailed above. PickPocket used the Monte Carlo search algorithm for this tournament.

Rank	Program	1	2	3	4	Total Score
1	PickPocket	-	64	67	69	200
2	Pool Master	49	-	72	65	186
3	Elix	53	54	-	71	178
4	SkyNet	53	65	55	-	173

Table 2: 8-Ball competition results

The tournament was held in a round-robin format, each pair of programs playing an eight game match. Ten points were awarded for each game won, with the losing program receiving points equal to the number of its assigned solids or stripes it successfully pocketed. PickPocket scored more points than its opponent in all three of its matches. The results of the tournament are shown in Table 2.

Conclusions

This paper described PickPocket, an adaption of game search techniques to the continuous, stochastic domain of billiards. Its approach to move generation and its evaluation function were described. The technique of estimating shot difficulties via a lookup table was introduced. Two search algorithms for billiards were detailed.

A man-machine competition between a human player and a billiards robot will soon occur. This research goes a long way towards building an AI capable of competing strategically with strong human billiards players.

Acknowledgments

Thanks to Michael Greenspan, Will Leckie, and Jonathan Schaeffer for their support and feedback. Financial support for this research was provided by NSERC and iCORE.

References

Cheng, Bo-Ru; Li, J.-T., and Yang, J.-S. 2004. Design of the neural-fuzzy compensator for a billiard robot. In *Networking, Sensing, and Control*, volume 2, 909–913.

Chua, S., et al. 2002. Decision algorithm for pool using fuzzy system. In *Artificial Intelligence in Engineering & Technology*, 370–375.

Chung, M.; Buro, M., and Schaeffer, J. 2005. Monte carlo search for real-time strategy games. In *IEEE Symposium on Computational Intelligence and Games*, 117–124.

Greenspan, M. 2005. UofA Wins the Pool Tournament. *ICGA Journal* 28(3):191–193.

Hauk, T. 2004. Search in Trees with Chance Nodes. Master's thesis, University of Alberta.

Leckie, W., and Greenspan, M. 2005. An event-based pool physics simulator. In *Proc. of Advances in Computer Games 11*. To appear.

Lin, Z.M.; Yang, J., and Yang, C. 2004. Grey decision-making for a billiard robot. In *Systems, Man, and Cybernetics*, volume 6, 5350–5355.

Long, F., et al. 2004. Robotic pool: An experiment in automatic potting. In *IROS'04*, volume 3, 2520–2525.