Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 171 (2007) 1069–1091
Artificial Intelligence
www.elsevier.com/locate/artint

PickPocket: A computer billiards shark

Michael Smith

Department of Computing Science, University of Alberta, Edmonton, Canada, T6G 2E8

Received 13 January 2007; received in revised form 12 April 2007; accepted 16 April 2007

Available online 29 April 2007

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 on the table. 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 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. This article describes the program PickPocket, the winner of the simulated 8-ball tournaments at the 10th and 11th Computer Olympiad competitions. PickPocket is based on the traditional search framework, familiar from games such as chess, adapted to the continuous stochastic domain of billiards. Experimental results are presented exploring the properties of two search algorithms, Monte-Carlo search and Probabilistic search.

Keywords: Game-tree search; Computer games; Uncertainty; Monte-Carlo methods; Billiards

1. Introduction

Billiards refers to a family of games played on a flat, rectangular table with pockets in the corners and on the sides. Players use a cue stick to strike the cue ball into an object ball, generally with the intent to drive the object ball into a pocket. The most popular billiards games are 8-ball pool, 9-ball pool, and snooker. There are also a wide variety of other games that can be played with the same equipment, including straight pool, one-pocket, and cutthroat.

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 been developed, starting with The Snooker Machine [23], and more recently Deep Green [21] and Roboshark [3]. 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' shot to play in any given game situation.

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

E-mail address: smith@cs.ualberta.ca.

0004-3702/$ – see front matter
doi:10.1016/j.artint.2007.04.011

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 shots. This makes it easier for them to pocket long consecutive sequences of balls. Strong players plan several shots ahead. The best players can frequently run the table (pocket all balls in sequence) off the break shot. The value of lookahead for humans suggests a search-based solution for building a billiards AI. Search has traditionally proven very effective for games such as chess. Like chess, billiards is a two-player, perfect information game. Also like chess, it is turn-based, although players do not strictly alternate taking shots. A player takes consecutive shots until they miss one, at which point the other player begins their sequence of shots. Two properties of the billiards domain strongly 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 $\langle x, y \rangle$ 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: the aiming direction, velocity, cue stick elevation angle, and x and y offsets of the cue stick impact position on the cue ball, 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. Environmental factors such as imperfections in the table baize (cloth) and even the room temperature can also affect ball 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.

The decision algorithms that robotics projects have used for shot selection have so far been relatively simplistic. They focus on finding the *easiest* shot, rather than the *best* shot. That is, they attempt to find the shot with the highest probability of success, without regard for position or strategic play. The algorithms described in [1,2,6,20] are all greedy algorithms which generate a set of shots, assign a difficulty score to each shot, and select the easiest shot to execute.

This article describes the program PickPocket, which competed in the computer billiards competitions held at the 10th and 11th Computer Olympiad. In contrast with the above previous approaches, PickPocket builds search trees.[1] Lookahead allows it to select shots that are both likely to succeed, and likely to lead to good position for future shots. The approach proved successful, as PickPocket came first place in both competitions. In addition to the Olympiad results, experimental results are presented which explore how the program responds to a variety of different conditions. Specifically, this article presents: a shot generator, evaluation function, and two search algorithms for billiards; a new approach to estimating the shot difficulty function; experimental results confirming the benefit of search over the previously standard greedy approach; and experimental results exploring the relationships between search algorithm, search depth, noise conditions, and game properties for the program PickPocket. A subset of this work was published in [25].

PickPocket plays 8-ball, as this was the game selected for the Olympiad tournaments. While many of the implementation details that follow are specific to 8-ball, the overall approach could be easily applied to any billiards game. The Olympiad tournaments form steps towards the long-range goal of this research: for an AI-driven billiards robot to compete with, and ultimately defeat, a top human player in a man–machine challenge.

[1] Skynet [18], developed concurrently with PickPocket, was the first search-based approach to reach publication.

2. Background and related work

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. Leckie's and Greenspan's *poolfiz* [16,17,19] 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 outcome 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 *poolfiz* 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.

PickPocket uses *poolfiz* to build search trees. From a given state, candidate shots are simulated to find successor states. This is a costly operation; it is where PickPocket spends the vast majority of its time. Whereas top chess programs can search millions of tree nodes per second, PickPocket searches hundreds of nodes per second. To save on runtime simulation costs, PickPocket precomputes several tables for performing common tasks.

While the tournaments held so far have been played in software simulation, a major goal of PickPocket is to fill the need for AI to control physical billiards robots. This can be achieved by providing it with a highly tuned and accurate physics simulator. For a sufficiently realistic simulator, shots that work well in simulation will work well in the real world.

2.2. Search in stochastic games

Stochastic games have in common the presence of random events that effect the play of the game. These random events can come in two forms: they may determine *what actions* a player has available to them (such as the roll of the dice in backgammon, or the drawing of tiles in Scrabble), or they may influence the *outcome* of the actions the player takes (such as in billiards shots, or the drawing of cards in poker). This non-determinism complicates search. Whenever a stochastic event occurs, instead of one successor state in the search tree, there are many. The search algorithm must take into account both the value to the player of each possibility, and the likelihood of it occurring. Two established approaches to dealing with this are Expectimax search and Monte-Carlo sampling.

2.2.1. Expectimax search

Expectimax, and its *-Minimax pruning optimizations, are an adaptation of game tree search to stochastic domains [4,14]. Expectimax search operates similar to standard Minimax search, with the addition of *chance nodes* wherever a non-deterministic action is to be taken. For example, dice rolls in the game of backgammon would be represented by chance nodes. Chance nodes have a child node for each possible outcome of the non-deterministic event. In backgammon, every possible outcome of the dice roll would have a corresponding child node. The value of a chance node is the weighted sum of its child nodes, where each child is weighted by its probability of occurring. Therefore this reflects the quality of the child nodes, with the ones most likely to occur having the most influence.

2.2.2. Monte-Carlo sampling

A Monte-Carlo sampling is a randomly determined set of instances over a range of possibilities. Each instance is assigned a value, and the average of all values is computed to provide an approximation of the value of the entire range. This implicitly captures the likelihood of good and bad outcomes (as a state that has mostly strong successor states will score highly, and vice versa), as opposed to Expectimax which explicitly uses probabilities to factor in the likelihood of events occurring.

Monte-Carlo techniques have been applied to a wide range of problems. In game-playing, they have been used in card games such as bridge [11] and hearts [26], as well as Go [5] and Scrabble [22]. All of these games have in common a branching factor too large for traditional search. All except Go have a stochastic element. To explicitly build full search trees would be impossible. In card games, the number of possible deals of the cards is massive. Rather than accounting for all of them explicitly, a set of instances of deals are sampled. In Go, the number of moves available to the player is too large to search deeply, so random games are played out to estimate the values of moves. In Scrabble,

possible tile holdings are sampled from the tiles known to be left in the bag, rather than searching every possibility. The Scrabble program Maven [22] also uses *selective sampling*, using a biased rather than uniform sampling to improve play.

Monte-Carlo search has also been investigated for continuous real-time strategy (RTS) games [7]. Unlike the turn-based games mentioned so far, these are real-time games where players take their actions simultaneously. This increases the complexity of the domain dramatically. In turn-based games, players have the benefit of having a static game state which they can spend time analyzing before taking an action. In contrast, the environment in an RTS is constantly changing. Turn-based games naturally impose a rigid structure on search trees. RTS actions can occur in any sequence, or simultaneously, so knowing how to structure a search tree is a difficult problem. RTS games feature a near-infinite branching factor to compound the challenge.

Stochastic games where search has previously been investigated fall therefore into two categories: turn-based games with discrete state and action spaces like card games, backgammon, and scrabble; and continuous real-time RTS games. Unlike the former, billiards has a continuous state and action space. Unlike the latter, billiards has a rigid turn-based structure. Also unlike card games where an opponent's hand is typically hidden, billiards features perfect information. Therefore billiards bridges the complexity gap, bringing together elements of traditional deterministic perfect information games like chess, stochastic games like backgammon, and realtime continuous games. It is 'more complex' than backgammon because of its continuous nature, yet 'less complex' than RTS games because of its turn-based structure. It is the first game with this particular set of properties to be examined from an AI perspective. There is a family of such turn-based continuous stochastic games, which include croquet, lawn bowling, shuffleboard, and curling. The techniques and considerations discussed here for billiards should carry over to these domains.

3. PickPocket implementation

Any search-based game-playing program consists of three main components: a move generator, an evaluation function, and a search algorithm. This section discusses the adaptation of each of these to a stochastic continuous domain, and describes PickPocket's 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 of breadth vs. depth in search.

3.1. Move generator

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.

In deterministic games, an attempted move always succeeds. A chess player cannot 'miss' when capturing an opponent's piece. In billiards, shots vary in their difficulty. Shots range from ones that 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 used by both the evaluation function and the search algorithm to perform their respective tasks.

The need to selectively generate relevant shots, and to assign a difficulty assessment to generated shots, arise respectively from the continuous and stochastic nature of the billiards domain.

Every billiards shot is defined by five continuous parameters [17], illustrated in Fig. 1:

- 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.

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 (Fig. 2), where the cue ball directly hits the object ball into the pocket; the bank shot (Fig. 3), where the object ball is banked off a rail into the pocket; the kick shot

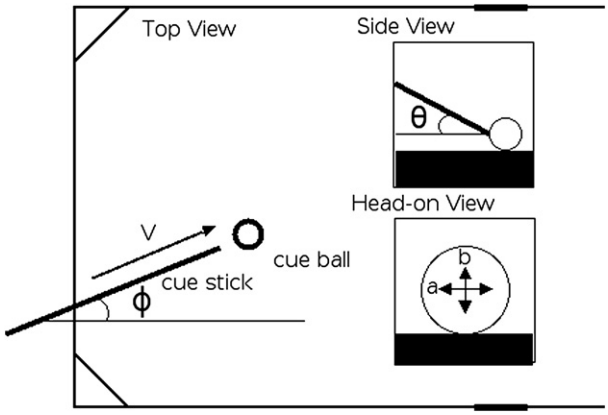

Fig. 1. Parameters defining a billiards shot.

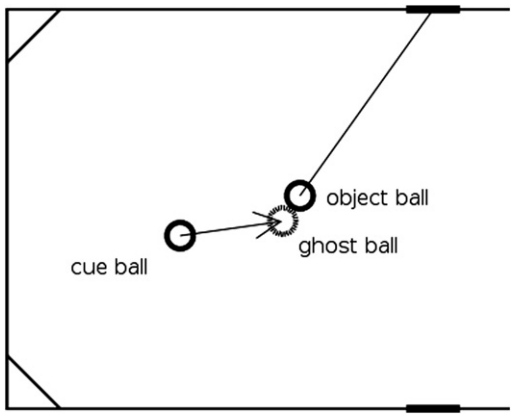

Fig. 2. A straight-in shot.

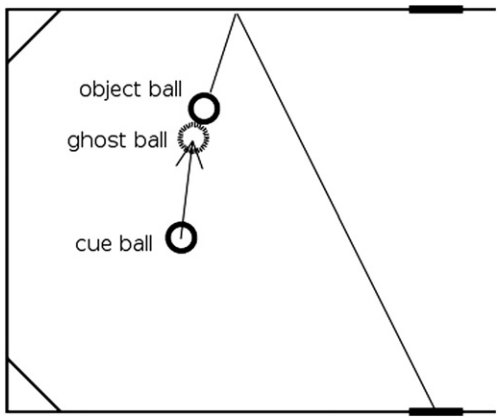

Fig. 3. A bank shot.

(Fig. 4), where the cue ball is banked off a rail before hitting the object ball into the pocket; and the combination shot (Fig. 5), where the cue ball first hits a secondary object ball, which in turn hits the target object ball into the pocket. Theoretically these can be combined to arbitrary complexity to create multi-rail bank and combination shots. In practice, difficulty increases rapidly with each additional collision, so players only attempt the more challenging types of shots when they lack easier options.

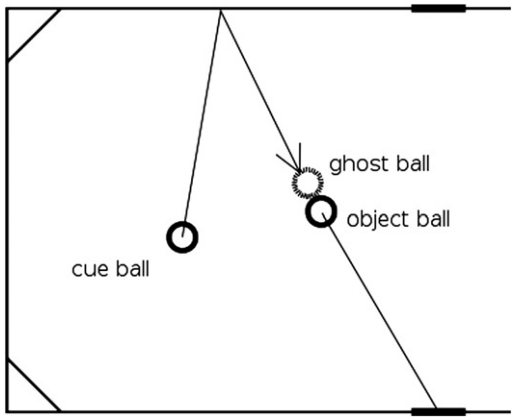

Fig. 4. A kick shot.

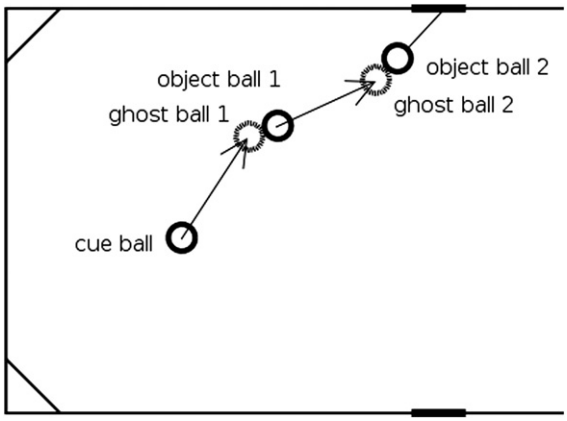

Fig. 5. A combination shot.

The target object ball is aimed with the aid of a conceptual 'ghost ball'. A line is extended from the intended target position (the center of a pocket, for example) through the center of the target object ball. The position adjacent to the object ball on the far side of this line from the target position is the ghost ball position. If the cue ball is aimed such that it is in this position when it contacts the target object ball, the target object ball will travel in the direction of the target position post-collision. Therefore, the ϕ aiming angle for the cue ball is chosen so that it is aimed directly at the ghost ball position to drive the object ball in the desired direction.

In addition to controlling the direction of the target object ball, the shooter has a significant degree of control over where the cue ball ends up after a shot. 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 traveled 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 can impart curvature on the cue ball's initial trajectory.

3.1.1. Straight-in shots

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. This can occur when the object ball is not between the cue ball and the target pocket, or when another object

ball blocks the path to the pocket. If the cue ball is very near or frozen against (in contact with) another object ball, it is impossible for the cue to hit it from some directions. Checks are made for these conditions, and impossible shots are discarded. When the shot is possible, the parameters are set as follows:

- ϕ is chosen such that the object ball is aimed at the exact center 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, found by starting at 5° and increasing in 5° increments until the cue stick is not in collision with any other object balls or the side of the table. Most of the time this turns out to be 5° or 10°. θ has the least impact on shot trajectory, so relatively large 5° increments are used to quickly find a physically possible value.
- a and b are set to zero.

This generates exactly 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 1 m/s increments up to *poolfiz's* maximum 4.5 m/s. The number of variations per ball and pocket combination has a strong impact on the branching factor when searching. These values were hand-tuned such that the 2-ply Monte-Carlo search algorithm from Section 3.3.2 would execute within the time limits imposed at the Computer Olympiad.

If one or more 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. A range of legal shots in each class can be generated in a manner similar to straight-in shots. However, straight-in shots are found a vast majority of the time, so further shot generation is generally not necessary. Experiments showed that bank, kick, and combination shots totaled under 6% of the total shots played under typical conditions [24].

3.1.2. Shot difficulty

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 ball-pocket distance $d1$, the cue-object ball distance $d2$, and object ball-pocket angle β (Fig. 6). α and $d2$ are calculated relative to a ghost ball position. 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, Dussault and Landry [8,9] based their approach on an expression of this function. Fuzzy [1,6] and grey [20] logic have been used to approximate the function. PickPocket uses a different approach, taking advantage of the *poolfiz* simulator to capture the shot difficulty function in a pre-computed table. The table is filled with accurate approximations of absolute probability values. Previous techniques generated arbitrary, relative values.

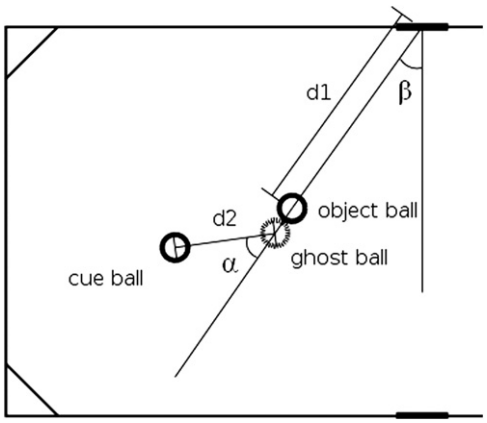

Fig. 6. Shot difficulty parameters.

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*. Due to the noise added to shots in simulation, the ball will successfully be pocketed in some samples and not in others. When simulation is complete, the percentage of the samples in which the ball was successfully pocketed is recorded in the table.

At runtime, a table lookup is made for each generated straight-in shot by finding the nearest table entry to the actual difficulty parameters for the shot. Each parameter is rounded to the granularity of the discretization used for the table. The value corresponding to these parameters is retrieved from the table, providing a quick estimate of the shot's 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. Slight deviations from these trajectories that would be insignificant on a clear table can now result in collisions with these obstacle balls. The exact value of the five shot parameters also has a small effect on the chance of success.

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 two $30 \times 30 \times 30 \times 30$ tables, one for the corner pockets and one for the side pockets, because of their differing difficulty functions. PickPocket uses $s = 200$, chosen for sub-24 hour precomputation time at the table granularity used.

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. This is an approximation that captures the increased difficulty of these shot classes, but does not have the high degree of accuracy of straight-in probability table lookups.

3.2. 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 value of a state 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 favorable 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 success 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 $d_1 p_1 + d_2 p_2 + d_3 p_3 + \cdots + d_n p_n$ 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 situations 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 $d_3 = 0.15$. Using these discount factors the first situation evaluates to 0.948, and the second to 1.036. These weights have been set manually, and could benefit from tuning via a machine learning algorithm.

Another possible weighting is $n = 1$ and $d_1 = 1.0$. This uses the score of the easiest shot only to evaluate a position. Compared to the above set of parameters, this loses information, as other shots available from the state are not considered. Intuitively, it is important to consider the presence of these other shots, as a state from which several easy shots are available has more options for position play. In practice, however, this set of parameters performed roughly equivalently to the above set. These experiments suggested that PickPocket's performance is not very sensitive to the exact values of its evaluation parameters [24].

3.3. 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.

3.3.1. Probabilistic search

Expectimax, and its *-Minimax optimizations, are natural candidates for searching stochastic domains [14]. 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 occurring. 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 minuscule probability of occurring. 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 s. 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 noise 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.

Unlike games such as chess where players strictly alternate taking moves, billiards has an open ended turn structure. A player may continue shooting as long as they legally pocket object balls. They only give up the shot when they miss, or call a safety. Because the table state after a failed shot is unknown, it is not possible to consider the opponent's moves in search. Thus, this search method only considers the shots available to the shooting player. The goal is to find a sequence of shots which is likely to clear the table, or leave the player in a good position from which to clear the table.

Probabilistic search, an Expectimax-based algorithm suitable for billiards, is shown in Fig. 7. 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 noise 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 search

```
float Prob_Search(TableState state, int depth){

  // if a leaf node, evaluate and return
  if(depth == 0) return Evaluate(state);

  // else, generate shots
  shots[] = Move_Generator(state);

  bestScore = -1;
  TableState nextState;

  // search each generated shot
  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;
}
```

Fig. 7. Probabilistic search algorithm.

depth used for the experiments reported in Section 4 was restricted more by this effect than by any time constraints relating to tree size. The player skill determines the magnitude of this effect.

3.3.2. Monte-Carlo search

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 hearts, as well as board games such 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 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. 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*.

Fig. 8 shows pseudo-code for the Monte-Carlo approach. `PerturbShot()` randomly perturbs the shot parameters according to the noise 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.

Note that in the case where there is no error, probabilistic search and Monte-Carlo search are logically identical. Searching to a given search depth, they will both generate the same result. It is entirely in how they handle the uncertainty introduced by the noise model that the two algorithms diverge.

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.

```
float MC_Search(TableState state, int depth){

  // if a leaf node, evaluate and return
  if(depth == 0) return Evaluate(state);

  // else, generate shots
  shots[] = Move_Generator(state);

  bestScore = -1;
  TableState nextState;
  Shot thisShot;

  // search each generated shot
  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;
}
```

Fig. 8. Monte-Carlo search algorithm.

An enhancement of Monte-Carlo search using Upper Confidence Trees (UCTs) [15] has recently proven effective at Monte-Carlo Go [10]. This is a selective sampling algorithm which biases sampling towards moves that are likely to be strong based on results of their previous sampling. The specific UCT algorithm is not directly applicable to billiards because it relies on a discrete state space—results of a move in a state are stored for use again later. In billiards, due to the continuous state space, it is highly unlikely that a specific state encountered somewhere in a tree will ever be seen again. However, the idea of selective sampling is sound and investigating ways of applying it to billiards search would be a good direction for future work.

3.4. Game situations

To play billiards games, an AI needs routines to handle the safety shot, break shot and ball-in-hand situations that occur regularly. This section describes the approach PickPocket uses for these situations.

3.4.1. 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, 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. i is set equal to the *num_samples* parameter described earlier, typically to a value of 15.

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, and only when there is no good straight-in shot available. Sampling as a one-time cost has a relatively minor impact on performance. At the root, if the best shot from the search has a score 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. For the 10th Computer Olympiad, these values were hand-tuned to $t_0 = 0.65$ and $t_1 = 0.5$ (evaluation values range from 0 to 1.48). Later experimentation suggested that this chose safeties too frequently; a strategy that played fewer safeties was more successful [24]. For the 11th Computer Olympiad, PickPocket used $t_0 = 0.5$ and $t_1 = 0.18$, which fared better in tests.

3.4.2. Break shot

Every billiards game begins with a break shot. This establishes the position of the object balls on the table, as well as which player gets to continue shooting. In most billiards games, including 8-ball, if a player pockets an object ball on the break shot, they may continue shooting. If they do not, their opponent gets the next shot.

Poolfiz randomizes the size of the small spaces between the object balls in the initial rack, leading to variation in the outcome of the break shot. Thus, break results are unpredictable. It is not feasible to respond dynamically to the exact details of the initial rack. The player can, however, select a shot that maximizes their chances of sinking a ball over the range of possible racks.

PickPocket uses sampling to precompute a break shot. A range of:

- Initial cue ball positions,[2]
- Velocities, and
- ϕ aiming angles

are sampled, with 200 samples taken for each set of parameters. The percentage of these that manage to sink an object ball is recorded. After sampling all positions, the set of parameters that led to the highest sinking percentage is selected. At runtime, when it is PickPocket's turn to break, this break shot is executed.

3.4.3. Ball-in-hand

A billiards player gets ball-in-hand when their opponent commits a foul. In 8-ball, a foul occurs if the cue ball is pocketed, or a player fails to hit a legal ball and rail on their shot (both must be hit to comprise a legal shot—a pocketed ball is considered to have hit the rail so this is only a consideration when a player fails to pocket one of their object balls). When a player has ball-in-hand, they are free to place the cue ball anywhere on the table. Ball-in-hand is a very strong situation, as the player can take advantage of it to set up an easy shot. Strong players often use ball-in-hand to sink 'trouble' balls that would be difficult to pocket from many positions on the table.

PickPocket must choose a position for the cue ball, as well as select a shot from that position, when it is awarded ball-in-hand. Although the cue ball could be placed anywhere on the table, it is impossible to do a full search from every position. Like with shot generation, a set of the most relevant positions for the cue ball must be generated. From each of these, search can proceed as normal by creating a table state with the cue ball in the selected position. The cue ball is ultimately placed at the position that led to the best search result, and the shot selected by that search is executed.

To generate a set of candidate positions, the table is discretized into a grid of cells. Each cell is assigned a value by generating shots as though the cue ball were at the center of that cell. Probability estimates for these shots are retrieved from the probability table. For each cell, the probability estimate of the best shot is set as the value of that cell.

This creates a map of the table, with the value of each cell corresponding to the ease of the best available shot from that cell. From this map, a set of candidate positions for the cue ball need to be retrieved. These positions should be high valued cells, from a range of different regions on the table to capture the breadth of available options. This is preferred over considering multiple neighboring cells, as the options available from neighboring cells are likely to be very similar, and could be captured by evaluating just one of them. To find the highest valued non-adjacent cells, local maxima are examined.

A randomized sampling search is used to approximate the k-best local maxima. A set of c cells on the table are randomly selected, and hill-climbing is performed from each of these to find c local maxima. Duplicates are

[2] The cue ball may be placed anywhere behind the headstring (a line across the width of the table located $1/4$ of the way along its length) on a break attempt.

```
Shot Compute_Ball_In_Hand_Shot(TableState state){

  // populate grid
  foreach(grid cell [x,y]){
   state.Cue_Position = cells(x,y).center;
   ShotSet shots[] = Generate_Shots(state);
   ShotSet.sort();
   grid[x,y] = shots[0].probabilityEstimate;
   // probability estimate for the best shot
  }

  // find best shot
  candidate_positions[] = Get_KBest_Local_Maxima(grid, k);
  Shot best_shot;
  best_shot.score = -1;
  foreach(candidate_positions[i]){
    shot = search(candidate_positions[i]);
    if(shot.score > best_shot.score){
      best_shot = shot;
    }
  }

  return best_shot;
}
```

Fig. 9. Ball-in-hand algorithm.

eliminated. The remaining values are then sorted, and the best k cell locations are returned. These are the candidate positions which are searched to find the final ball-in-hand shot.

Fig. 9 gives pseudocode for the entire ball-in-hand shot selection process. `TableState` has a parameter `Cue_Position` which controls the position of the cue ball in that table state. `Get_KBest_Local_Maxima (grid,k)` returns a set of grid positions approximating the k best local maxima on the grid.

4. Experiments

Pickpocket has a wide range of adjustable parameters and features. This section presents experimental results demonstrating the impact of these features on the program's performance. Experiments were performed with the Probabilistic search parameters, Monte-Carlo search parameters, game properties, and a comparison of the search algorithms. A description of the experimental setup and the results of these tests follow.

4.1. Noise model

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.

4.2. Sample size and statistical significance

Because of the stochastic nature of the domain, all experimental results are subject to uncertainty. This arises from the two sources of randomness: the random spacing between the balls as they are racked, and the error added to requested shot parameters by the noise model. The former leads to variations in the positions of object balls on the table after the break, and the latter leads to variations in the outcome of each requested shot. In the long run—over a very large or infinite number of games—the effects of this randomness even out as both sides are helped and hurt by

it equally. However, over the course of tens or hundreds of games as played here, the random element is significant and must be kept in mind when analyzing match results.

Running matches as experiments determines a win rate between program variants. Each match result is an approximation of an underlying long-run win rate. Ideally we would like to know which side is superior and by how much; we would like to know the exact value for this underlying win rate. However, the sample size required to approximate this to a high degree of accuracy would be very large. The experiments in this section will give an indication whether one side is superior to another, or whether they are roughly equal, but the exact underlying win rate will remain unknown.

100-game samples were chosen for the experiments in this section to provide a balance between accuracy and execution time. A significantly larger sample size would be required to substantially increase the accuracy of the experiments. Any 100-game experiment where one side wins 60 games or more is strong evidence (at least 95% confidence) of that side's superiority, as the entire 95% confidence interval for that side lies above a 50% win rate [24]. Matches with results closer to 50–50 are not conclusive one way or another to a strong degree of confidence. However, the higher above 50 a score gets, the more likely it becomes that it is the long-run winning program.

4.3. Experimental setup

PickPocket plays 8-ball, the game selected for the first computational billiards tournament. To summarize the rules, each player is assigned a set of seven object balls: either solids or stripes, determined when the first ball is legally pocketed after the break shot. The pocketing player is assigned that ball's set, and their opponent is assigned the other set. 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.

A series of matches were played to evaluate the impact of PickPocket's many parameters on its performance. Typically each match holds all but one parameter constant, to isolate the effect of that one parameter. Bank, kick, and combination shot generation is disabled, to simplify the experiments and compare purely the effects of search. Therefore, in these matches every shot is either a straight-in attempt, a safety attempt, or a break shot to begin a new game. Players alternate taking the break to eliminate any potential advantage or disadvantage of going first. For each match, the following results are given:

- W (Wins), the number of games the program won.
- SIS (Straight-in-success), the ratio of successful to attempted straight-in shots. A successful straight-in shot sinks the called object ball; the player continues shooting.
- SS (Safety success), 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.

Unless otherwise stated, PickPocket is configured as the following for the matches. A $30 \times 30 \times 30 \times 30$ granularity lookup table is used. Monte-Carlo search is used to 2-ply, with $num\_samples = 15$. Safety thresholds are set at $t_0 = 0.0$ and $t_1 = 1.48$ (the maximum possible evaluation), so safeties are played if and only if no other shots can be generated.

Parameters were chosen such that a decision was made for each shot within approximately 60 seconds. This is a typical speed for time-limited tournament games. A 10-minute per side per game hard time limit was imposed, to ensure that program variants made their decision within a reasonable amount of time. 2-ply Monte-Carlo search was the only variant that approached this limit.

Experiments were run under three noise models: E_{low}, E_{mid}, and E_{high}. E_{low}, with parameters $\{0.0185, 0.003, 0.0085, 0.08, 0.08\}$, corresponds to a top human player who can consistently make even challenging shots successfully. E_{mid}, with parameters $\{0.185, 0.03, 0.085, 0.8, 0.8\}$, was the noise model used at the 10th Computer Olympiad. It models a strong amateur, who can consistently pocket short, easy shots, but sometimes misses longer shots. E_{high}, with parameters $\{0.74, 0.12, 0.34, 3.2, 3.2\}$, corresponds to a human novice who can usually make short, easy shots, sometimes make medium difficulty shots, and rarely make challenging shots.

Table 1
Effect of search depth in Probabilistic search

Match	W	SIS	SS
		E_{low}	
Depth 1	48	446/502 = 88.8%	56/101 = 55.4%
Depth 2	52	464/519 = 89.4%	58/99 = 58.6%
Depth 1	54	502/557 = 90.1%	38/86 = 44.2%
Depth 3	46	446/523 = 85.3%	35/69 = 50.7%
Depth 1	53	450/507 = 88.7%	40/78 = 51.3%
Depth 4	47	481/532 = 90.4%	38/86 = 44.2%
		E_{mid}	
Depth 1	66	541/698 = 77.5%	90/164 = 54.9%
Depth 2	34	459/661 = 69.4%	76/137 = 55.5%
Depth 1	63	545/687 = 79.3%	83/160 = 51.9%
Depth 3	37	518/689 = 75.2%	68/142 = 47.9%
Depth 1	61	534/708 = 75.4%	83/141 = 58.7%
Depth 4	39	511/697 = 73.3%	72/139 = 51.8%
		E_{high}	
Depth 1	59	475/1009 = 47.1%	143/219 = 65.3%
Depth 2	41	463/1034 = 44.8%	122/189 = 64.6%
Depth 1	52	487/1056 = 46.1%	156/226 = 69.0%
Depth 3	48	478/1098 = 43.5%	110/186 = 59.1%
Depth 1	59	522/1082 = 48.2%	148/225 = 65.8%
Depth 4	41	490/1129 = 43.4%	96/163 = 58.9%

4.4. Probabilistic search

Matches were played between variants of the Probabilistic search algorithm searching to various depths to evaluate the relationship between search depth and performance. The results of the matches are shown in Table 1.

Under E_{low}, the various search depths all have roughly equal performance. There is no apparent gain from searching deeper, but neither is there a significant penalty. Under E_{mid} and E_{high}, on the other hand, 1-ply search clearly outperforms any deeper search, winning every match under these noise models. This suggests that, as error increases, any benefit to deep lookahead is canceled out by the compounding effect of error over several ply of search. Probabilistic search, with its deterministic node expansion, is expected to be the algorithm most susceptible to this type of error.

Leckie's and Greenspan's experiments in [18] show a similar result. They found that, for their Expectimax-based billiards search algorithm, deeper search did improve performance when the amount of error introduced was small. When they repeated the experiments with larger error, deeper search fared worse.

4.5. Monte-Carlo search

Monte-Carlo search is controlled by two parameters: search depth and *num_samples*, the number of samples to expand for each shot. Table 2 shows the effect of sample size in a 1-ply search. Table 3 shows the effect of sample size in a 2-ply search. These experiments were both run under the E_{mid} noise model. Note that this experiment was not run for *num_samples* = 30 because that variant did not run within tournament time constraints. Table 4 shows the effect of search depth on performance, with *num_samples* fixed at 15. Again, 2-ply was the deepest variant that would run within time constraints.

All of the match results varying sample size are too close to call. This suggests that the exact parameters used in Monte-Carlo search have a relatively minor impact on performance. Interestingly, all three of the variants using a larger sample size edged out the variants using a smaller sample size. This might suggest that larger sample size confers a slight benefit. This would be expected, as the larger the sample size, the better the underlying distribution

Table 2
Effect of sample size in 1-ply Monte-Carlo search (E_{mid})

Match	W	SIS	SS
Samples = 5	47	492/615 = 80.0%	25/73 = 34.2%
Samples = 15	53	504/637 = 79.1%	21/59 = 35.6%
Samples = 5	46	532/688 = 77.3%	27/73 = 37.0%
Samples = 30	54	547/707 = 77.3%	29/73 = 39.7%

Table 3
Effect of sample size in 2-ply Monte-Carlo search (E_{mid})

Match	W	SIS	SS
Samples = 5	46	500/643 = 77.8%	23/75 = 30.7%
Samples = 15	54	528/667 = 79.1%	34/66 = 51.5%

Table 4
Effect of search depth in Monte-Carlo search

Match	W	SIS	SS
		E_{low}	
Depth 1	36	381/439 = 86.8%	3/35 = 8.6%
Depth 2	64	533/593 = 89.9%	4/18 = 22.2%
		E_{mid}	
Depth 1	55	504/616 = 81.8%	25/57 = 43.9%
Depth 2	45	492/621 = 79.2%	15/41 = 36.6%
		E_{high}	
Depth 1	43	471/985 = 47.8%	50/97 = 51.5%
Depth 2	57	501/1008 = 49.7%	51/103 = 49.5%

of shot outcomes is approximated. Certainly a smaller sample size would not be expected to perform better than a larger sample size. However, because the results are all so close, it could just as easily be a result of noise that the larger sample size won every time; too much should not be read into the result. It would be interesting to see what impact using a much larger sample size has on performance. Using parallel computing this could be achieved while still maintaining acceptable performance.

The experiment with search depth also had an inconclusive result under the E_{high} and E_{mid} noise models. It is not clear whether 2-ply Monte-Carlo search performs better, worse, or approximately equal to 1-ply search under these conditions. The results suggest that they are quite close in performance. Compared to Probabilistic search, Monte-Carlo fares better at deeper depths. This is expected as it better handles the error introduced at every shot. Under the E_{low} noise model, 2-ply is stronger than 1-ply with statistical confidence. This echoes the result seen in the search depth experiments with Probabilistic search, where deeper search exhibited better performance when there was less noise relative to when there was more noise.

2-ply did generally perform fewer safety shots than 1-ply search. Since safeties are only executed when no straight-in shots are available, this means it left itself with no straight-in shots fewer times than 1-ply did. This is an expected benefit of deeper search. However, because the safety shots constitute a small percentage of the overall shots in the match, this factor likely had a minimal effect on the final result.

4.6. 8.5-ball

The experiments with search depth in 8-ball only showed a benefit to further lookahead beyond 1-ply under the E_{low} noise model, and there the benefit was only substantial for the Monte-Carlo search algorithm (there is always a large benefit to searching 1-ply versus a greedy, non-searching algorithm—see the next section). With larger error, in Probabilistic search deeper search actually performed worse than 1-ply search. In Monte-Carlo search, the match

results were too close to declare any advantage for 2-ply search over 1-ply, or vice versa. This result is surprising as traditionally search depth is strongly correlated with performance in game-playing programs. Interestingly, additional experiments showed deeper search to be beneficial under the E_{mid} noise model when using a material difference evaluation [24]. The increased search depth implicitly added information that compensated for a weak evaluation function.

There are 3 possible reasons why deeper search might not be so beneficial. Firstly, the evaluation function could be good enough that the 1-ply evaluation consistently gives the best shots the highest value. Such strong evaluations exist for backgammon, a game where relatively little additional strength comes from deeper search [14]. This is unlikely to explain the results seen in 8-ball, as deeper search was clearly beneficial under the E_{low} noise model using the Monte-Carlo search algorithm.

Secondly, noise makes the states seen, and therefore the evaluations returned, less accurate as depth increases. Because of the noise model, the root of the search tree is the only state that will actually occur in-game. The other nodes that comprise the trees are states that are likely to be similar to states that will be seen. However, inaccuracies can compound over several ply. This is likely why deeper search is actually a worse performer in Probabilistic search. Deeper Monte-Carlo search does not clearly fare worse likely because the algorithm is better suited to accounting for this type of error. As noise increased from E_{low} to E_{high}, both algorithms saw a decrease in the effectiveness of deeper search.

The third possible reason for the lack of benefit to additional search depth has to do with the properties of 8-ball. When playing, the player has the option of shooting at any of their objects balls. At the beginning of the game there are seven balls they can select to shoot at. As the table clears, they have fewer potential object balls to target, but at the same time there are fewer object balls obscuring potential shots. The player will very frequently have shots available. Further, this feature makes position play easier because the player can set up for position on any of their remaining object balls. The potential shots in 8-ball are not very constrained. Because of this, it is likely that there will be good shots available from a wide range of table states. It is relatively hard for a player to 'run themselves into a corner' and be left with no good shots available. This can be seen from the relative infrequency at which safety shots are played in PickPocket's 8-ball matches.

In contrast, some billiards games more strongly constrain the shots available to the player. In 9-ball, the player must shoot at the lowest numbered ball remaining on the table. When shooting, they are aiming at one specific ball, and trying to gain position on one other specific ball (the next-lowest ball remaining on the table), rather than having the option of any ball in a set. From more table states there will not be shots available that both sink the target ball and get position on the next ball. It is easier for a player to run themselves into a corner in such games. Therefore, deeper search would be expected to confer a stronger benefit in these games.

To test whether this is the case, the game of 8.5-ball was created. The rules of 8.5-ball are identical to those of 8-ball, except that the player may only shoot at the lowest numbered object ball remaining in their set. This constrains the shots available, and the positional options available, in a manner similar to 9-ball. The search depth experiments for the Probabilistic and Monte-Carlo search algorithms were repeated for this game under the E_{mid} noise model. The results of these matches are shown in Tables 5 and 6.

Now in 8.5-ball using Probabilistic search, depth 2 search won 56–44 games against depth 1. While this is still an inconclusive match result, it is very strong evidence of an improvement over the previous 34–66 loss suffered by

Table 5
Effect of search depth in Probabilistic 8.5-ball (E_{mid})

Match	W	SIS	SS
Depth 1	44	420/666 = 63.0%	266/423 = 62.9%
Depth 2	56	428/664 = 64.5%	283/433 = 65.4%
Depth 1	65	477/693 = 68.8%	281/427 = 65.8%
Depth 3	35	399/620 = 64.4%	257/434 = 59.2%
Depth 1	56	438/669 = 65.5%	300/466 = 64.4%
Depth 4	44	447/714 = 62.6%	254/430 = 59.1%

Table 6
Effect of search depth in Monte-Carlo 8.5-ball (E_{mid})

Match	W	SIS	SS
Depth 1	40	380/509 = 74.7%	73/139 = 52.5%
Depth 2	60	378/517 = 73.1%	54/123 = 43.9%

depth 2 search in 8-ball under the same noise model. A t-test comparing the two proportions yields $\underline{t}(197)^3 = 3.21$, $\underline{p} = 0.002$—there is a 0.2% probability that the observed difference was due to chance. 3-ply and 4-ply search still fare worse than 1-ply. The 4-ply match result is close enough to be statistically inconclusive, but considering the 3-ply result and that the 4-ply player was the loser of this match, it is likely that 1-ply is indeed the stronger variant.

Using Monte-Carlo search in 8.5-ball, 2-ply search clearly and statistically significantly outperforms 1-ply search, winning 60–40. Like with Probabilistic search, this is a substantial improvement from the 8-ball result of 45–55 under the same noise model ($\underline{t}(197) = 2.15$, $\underline{p} = 0.03$). Overall, these results add up to the properties of this game favoring deeper search, much more so than the game of 8-ball. This is an interesting result, as it shows that the importance of search depth in billiards games, as well as being a function of the noise model, is a function of the properties of the particular game being played.

Note also the higher proportion and success rate of safety shots in 8.5-ball, under both algorithms. The greater proportion of safety shots is a result of the player being required to target one specific ball. More often there will be no shot available on that ball, so the player will have to perform a safety. The higher success rate also follows. Since it is known which ball the opponent will have to target, it is easier to find a safety shot that prevents that player from having a shot on that specific ball.

4.7. Search algorithm comparison

Experiments were constructed to compare the main 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 of this form were used to select shots in [1,2,6, 20].
- Prob: The Probabilistic search algorithm.
- MC: The Monte-Carlo search algorithm with *num_samples* = 15.

Matches were played between each pair of algorithms. Under each noise model, each of the search algorithms was run to the best performing search depth for that amount of noise, as determined above. Thus, under E_{low}, Prob and MC searched to 2-ply depth. Under E_{mid}, Prob and MC searched to 1-ply depth. Under E_{high}, MC searched to 2-ply depth and Prob to 1-ply depth. Table 7 shows the tournament results.

Both search algorithms defeated Greedy convincingly under all error conditions. 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 noise 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 noise 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 generally increased from Greedy to Probabilistic to Monte-Carlo search, with the difference between Greedy and Probabilistic being much greater than that between Probabilistic and Monte-Carlo search.

[3] 197 is the degrees-of-freedom.

Table 7
Comparison of search algorithms

Match	W	SIS	SS
		E_{low}	
Greedy	7	202/377 = 53.6%	43/68 = 63.2%
Prob	93	659/726 = 90.8%	74/128 = 57.8%
Greedy	3	122/249 = 49.0%	23/38 = 60.5%
MC	97	662/722 = 91.7%	29/59 = 49.2%
Prob	38	375/432 = 86.8%	15/40 = 37.5%
MC	62	510/566 = 90.1%	12/28 = 42.9%
		E_{mid}	
Greedy	19	411/745 = 55.2%	74/119 = 62.2%
Prob	81	631/829 = 76.1%	138/222 = 62.2%
Greedy	22	341/604 = 56.5%	35/81 = 43.2%
MC	78	637/806 = 79.0%	66/135 = 48.9%
Prob	44	525/675 = 77.8%	74/145 = 51.0%
MC	56	552/730 = 75.6%	63/109 = 57.8%
		E_{high}	
Greedy	27	381/1136 = 32.4%	79/119 = 66.4%
Prob	73	579/1145 = 50.6%	192/273 = 70.3%
Greedy	25	368/1039 = 35.4%	59/89 = 66.3%
MC	75	554/1023 = 54.2%	101/153 = 66.0%
Prob	33	432/972 = 44.4%	98/155 = 63.2%
MC	67	495/1040 = 47.6%	77/131 = 58.8%

Under E_{high}, here as in the earlier experiments, the games tended to be longer as the lower accuracy led to more missed shots. Towards E_{low}, matches completed increasingly faster with fewer misses. The change in straight-in completion rate for a given algorithm between noise models represents this change in accuracy. Additionally, winning programs tend to 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, totaled 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 strategic option. Safeties were more effective under E_{high}, frequently returning the turn to the player. They were generally 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.

Monte-Carlo search is clearly the strongest of the algorithms. Under all noise models, it defeated Greedy by a wide margin, and then defeated Probabilistic search in turn. The victories over Probabilistic search under E_{low} and E_{high} have statistical confidence, while the E_{mid} result is too close to call. Overall, this suggests that the value of sampling and taking into account the range of possible shot outcomes is substantial under a wide range of noise models. This is in agreement with the results of the previous experiments on search depth.

5. Computer Olympiad tournaments

PickPocket won the first international computational 8-ball tournament at the 10th Computer Olympiad [12]. Games were run over a *poolfiz* server, using the E_{mid} noise model detailed earlier. PickPocket used the Monte-Carlo search algorithm for this tournament, searching to 2-ply depth.

The tournament was held in a round-robin format, each pair of programs playing an 8-game match. Ten points were awarded for each game won, with the losing program receiving points equal to the number of its assigned solids

Table 8
Computer Olympiad 10 competition results

Rank	Program	1	2	3	4	Total score
1	PickPocket	–	64	67	69	200
2	PoolMaster	49	–	72	65	186
3	Elix	53	54	–	71	178
4	SkyNet	53	65	55	–	173

Table 9
Computer Olympiad 11 competition results

Rank	Program	1	2	3	4	5	Total	Win %age
1	PickPocket	–	38	34	37	39	148	74.0%
2	Elix	12	–	26	31	36	105	52.5%
3	SkyNet	16	24	–	28	37	105	52.5%
4	PoolMaster	13	19	22	–	22	76	38.0%
5	Snooze	11	14	13	28	–	66	33.0%

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 8.

PickPocket also won the 11th Computer Olympiad, held in Italy in May 2006. The same setup was used, with a slightly different noise model: E_{11} has parameters $\{0.125, 0.1, 0.075, 0.5, 0.5\}$. This corresponds to a somewhat more accurate player. Tests showed than under E_{11}, 75.66% of straight-in shots in random table states were successful, over a 10,000 shot sample size. Under E_{mid}, the same test resulted in 69.07% of shots being pocketed.

One shortcoming of the 10th Olympiad was that programs only played 8-game matches against one another. While the results certainly suggested that PickPocket was the top program, there was no way of knowing whether this was really the case or if PickPocket won because of fortuitous random events. Counting balls pocketed in a loss helped overcome the effects of variance somewhat, as a stronger program is likely to pocket more balls when it loses. However, the overall results cannot be claimed to be strongly statistically significant (comparing the proportion of points achieved by PickPocket to the proportion achieved by the second place program by t-test, $\underline{t}(471) = 1.62$, $\underline{p} = 0.11$—there is an 11% probability that the observed difference was due to chance).

To address this, the 11th Olympiad featured a 50-game match between each pair of competing programs. Because this is a larger sample size, balls pocketed in a loss were not counted. The scores represent the number of games out of the 50-game match won by each player. The results of the tournament are shown in Table 9 [13].

Again PickPocket scored more wins than its opponent in all of its matches, winning 68–78% of games versus all opponents. Overall it won 74% of the games it played, defeating the next best performers by a substantial margin. This is a convincing, statistically significant result (applying the equivalent t-test here, $\underline{t}(391) = 4.58$, $\underline{p} = 0.0$—there is a 0% probability that the observed difference was due to chance). The version of PickPocket that competed in this tournament was substantially tweaked and improved from the previous year.

A variety of approaches were used by the other entrants in the tournament, which were developed concurrently with PickPocket. The main distinguishing features of each program are as follows.

5.1. Elix

Elix was developed by a strong billiards player who translated his knowledge of the game into an ad-hoc, rule based approach to shot selection. A set of shots are generated, and then a sequence of rules is used to select which shot to execute. These rules operate on such inputs as the shot difficulty parameters, the results of a 1-ply Monte-Carlo type search, and other features of the table state. Because of complex interactions between rules, Elix would sometimes make poor shot selections. Contrast this with the uniform approach used by PickPocket, where every shot (save safeties) is found as a result of the same search process. Since PickPocket has few special cases, there is less risk of them being inadvertently activated.

5.2. SkyNet

Leckie's and Greenspan's SkyNet [18] used an Expectimax search-based approach, similar to the Probabilistic search detailed for PickPocket. In tournament play, PickPocket used Monte-Carlo search, so the results of the matches between PickPocket and Skynet further reinforce the findings of Section 4.7 that Monte-Carlo search is superior. In tournament play SkyNet searched to a 3-ply depth. To estimate the probability of success of a shot, a runtime Monte-Carlo sampling algorithm was used. This formed the multiplier for each child node's returned value. This contrasts with PickPocket's more efficient, but less accurate, use of a lookup table to estimate each shot's probability of success. SkyNet's leaf evaluations were based on the number of shots available at leaf nodes, it did not take into account the quality of those shots as PickPocket does.

5.3. PoolMaster

Leckie and Greenspan describe two paradigms for billiards shot generation in [18]: shot discovery and shot specification. PickPocket, Elix, and SkyNet all use shot discovery methods. Here, shots leading to position play are generated by blindly varying an initial shot to create a set of shots which will all lead to different final cue-ball positions. Search then 'discovers' the shots among these that lead to good position play. In contrast, a shot specification approach to generation explicitly chooses good final cue ball positions, and then uses sampling and optimization to find shots that leave the cue ball as near as possible to these positions. Shot discovery is computationally cheap, as no physics simulation is required. Shot specification is expensive, as extensive physics simulations are required to find a shot that best leaves the cue ball in the desired final position.

PoolMaster [9] uses a shot specification approach to move generation; positions on the table are scored to find good positions for the cue ball to come to rest after the shot. Local maxima are used as candidates, similar to PickPocket's ball-in-hand play. The optimization algorithm used to direct sampling to find shots that leave the cue ball in these positions is described in [8]. PoolMaster performs 1-ply search on the generated shots. However, it used a basic search algorithm which did not take into account the probability of success of the generated shots. PoolMaster therefore often made risky shots in tournament play, which likely would not have been chosen if their difficulty had been taken into account.

5.4. Snooze

Snooze was a late entrant to the tournament, and the version that competed was still a work-in-progress. No details of its operation are available.

6. Man–machine challenge

A major goal of any game-playing project is to be able to defeat top human players. This has been done in chess and checkers, and the Robocup project aims to defeat a top human team at soccer by the year 2050. Defeating top humans is a strong demonstration of effectiveness, and a goal that motivates the development of new techniques. Ultimately billiards robots will be strong enough to accomplish this, and may even become common as practice opponents, just as chess players now play against computer programs for practice and entertainment.

For a billiards robot to challenge a top human, three components must be in place:

(1) The robot must have shot error margins roughly as low as the top humans. The exact accuracy required to win depends on how the AI's shot selection compares to the top human's shot selection. If the AI's shot selection is weaker than the human's, the robot must be more accurate than the human competitor. If it has better shot selection than the human, then the robot may not need to be quite as accurate as the human to win the match.
(2) Physics simulation must be accurately calibrated to the physical table. The AI's shot selection is based on the results of physics simulation. If the simulation accurately predicts what will happen on a physical table as a result of a shot, then the shots found by the AI will be effective.
(3) The shot selection of the AI driving the robot must be roughly as good as the top human. Again, the exact requirements depend on the physical accuracy of the robot being driven.

Early forms of all three components are now in place. Soon full games will be held between the robot Deep Green [21] and human challengers. The physical accuracy of the robot, and calibration of the physics simulation, are still a long way from being sufficient to challenge the best humans. The current systems should provide an entertaining challenge for humans, and should have a chance of defeating casual players.

It is unclear how the current version of PickPocket would fare against a top human player, as there is no way to directly compare just shot selection. Without a robot that is similar in accuracy to strong humans, it would be difficult to tell (except for obvious mistakes) whether losses by the robot were due to weak shot selection, or a lack of sufficient physical accuracy. Having an expert human comment on PickPocket's shot selection may provide insight on the strength of its strategic play. However in some games, such as backgammon, non-conventional moves found by computers actually turned out to be better than the established wisdom, and resulted in changes over time to the strategies employed by humans.

The properties of the game chosen for a man–machine challenge may have an impact on the robot's chance of success. In 8-ball, 1-ply search is sufficient to consistently find good shots; depending on the noise model, there may be little advantage to additional lookahead. With current technology, a robotic player may therefore have a better chance of defeating a top human at 8-ball than 9-ball, where lookahead is more important, and the best human players can plan a path to clear the entire table whereas current computers cannot.

7. Conclusions

This article described PickPocket, an adaption of game search techniques to the continuous, stochastic domain of billiards. Its approach to move generation, evaluation function, and the Probabilistic and Monte-Carlo search algorithms were described. Experimental results proved the benefit of lookahead search over the previously standard greedy technique. They demonstrated that Monte-Carlo search is the strongest of the two presented search algorithms under a wide range of error conditions. Additionally, PickPocket proved itself the world's best billiards AI at the 10th and 11th Computer Olympiad competitions.

The exact benefit of search depth was shown to depend on both the amount of noise added to each shot, and the properties of the specific billiards game being played. Surprisingly, 1-ply search performed best under a wide range of conditions. Only under conditions of low error and constrained shooting was deeper search an improvement. This contradicts on the surface the conventional wisdom that deeper search depth is strongly correlated with increased performance. However, billiards' continuous stochastic nature makes evaluations at deeper depths less relevant than in games with more traditional properties. Overall, these results suggest that noise and shot constraints should be kept in mind when adapting to various robotic platforms and alternate billiards games. This should prove useful as the next Computer Olympiad billiards competition will feature a change of games to something other than 8-ball.

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 players.

Acknowledgements

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

[1] M.E. Alian, S.B. Shouraki, A fuzzy pool player robot with learning ability, WSEAS Transactions on Electronics 2 (1) (2004) 422–426.
[2] M.E. Alian, S.B. Shouraki, C. Lucas, Evolving strategies for a pool player robot, WSEAS Transactions on Information Science and Applications 5 (1) (2004) 1435–1440.
[3] M.E. Alian, S.B. Shouraki, M.M. Shalmani, P. Karimian, P. Sabzmeydani, Roboshark: A gantry pool playing robot, in: 35th International Symposium on Robotics (ISR 2004), 2004, electronic publication.
[4] B.W. Ballard, The *-minimax search procedure for trees containing chance nodes, Artificial Intelligence 21 (3) (1983) 327–350.
[5] B. Bouzy, B. Helmstetter, Monte Carlo Go developments, in: Advances in Computer Games, Many Games, Many Challenges, Springer-Verlag, 2003, pp. 159–174.
[6] S. Chua, W. Tan, E. Wong, V. Koo, Decision algorithm for pool using fuzzy system, in: Artificial Intelligence in Engineering & Technology, 2002.

[7] M. Chung, M. Buro, J. Schaeffer, Monte Carlo search for real-time strategy games, in: IEEE Symposium on Computational Intelligence and Games, 2005.
[8] J.-P. Dussault, J.-F. Landry, Optimization of a billiard player—position play, in: Advances in Computer Games, 11th International Conference, ACG 2005, Taipei, Taiwan, September 6–9, 2005, in: Lecture Notes in Computer Science, vol. 4250, Springer-Verlag, 2006. Revised Papers.
[9] J.-P. Dussault, J.-F. Landry, Optimization of a billiard player—tactical play, in: Proc. of Computers and Games 2006, Torino, Italy, May 2006, in press.
[10] S. Gelly, Y. Wang, R. Munos, O. Teytaud, Modifications of UCT with patterns in Monte-Carlo Go, Technical Report 6062, INRIA, 2006.
[11] M.L. Ginsberg, Gib: Steps toward an expert-level bridge-playing program, in: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1999.
[12] M. Greenspan, UofA wins the pool tournament, International Computer Games Association Journal 28 (3) (2005) 191–193.
[13] M. Greenspan, PickPocket wins the pool tournament, International Computer Games Association Journal 29 (3) (2006) 153–156.
[14] T. Hauk, Search in trees with chance nodes, Master's thesis, University of Alberta, January 2004, http://citeseer.ist.psu.edu/hauk04search.html.
[15] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Proceedings, Berlin, Germany, September 18–22, 2006, in: Lecture Notes in Computer Science, vol. 4212, Springer-Verlag, 2006.
[16] W. Leckie, M. Greenspan, Pool physics simulation by event prediction 1: Motion transitions, International Computer Games Association Journal 28 (4) (2005) 214–222.
[17] W. Leckie, M. Greenspan, An event-based pool physics simulator, in: Advances in Computer Games, 11th International Conference, ACG 2005, Taipei, Taiwan, September 6–9, 2005, in: Lecture Notes in Computer Science, vol. 4250, Springer-Verlag, 2006. Revised Papers.
[18] W. Leckie, M. Greenspan, Monte Carlo methods in pool strategy game trees, in: Proc. of Computers and Games 2006, Torino, Italy, May 2006, in press.
[19] W. Leckie, M. Greenspan, Pool physics simulation by event prediction 2: Collisions, International Computer Games Association Journal 29 (1) (2006) 24–31.
[20] Z. Lin, J. Yang, C. Yang, Grey decision-making for a billiard robot, in: Systems, Man, and Cybernetics, vol. 6, 2004.
[21] F. Long, J. Herland, M.-C. Tessier, D. Naulls, A. Roth, G. Roth, M. Greenspan, Robotic pool: An experiment in automatic potting, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 04), vol. 3, 2004.
[22] B. Sheppard, World-championship-caliber scrabble, Artificial Intelligence 134 (1–2) (2002) 241–275.
[23] S.W. Shu, Automating skills using a robot snooker player, PhD thesis, Bristol University, 1994.
[24] M. Smith, PickPocket: An AI for computer billiards, Master's thesis, University of Alberta, September 2006.
[25] M. Smith, Running the table: An AI for computer billiards, in: Proceedings of the National Conference on Artificial Intelligence (AAAI 06), AAAI Press, 2006.
[26] N. Sturtevant, Multiplayer games: Algorithms and approaches, PhD thesis, UCLA, 2003.