Analysis of a Winning Computational Billiards Player *

Christopher Archibald, Alon Altman and **Yoav Shoham**
Computer Science Department
Stanford University
{cja, epsalon, shoham}@stanford.edu

Abstract

We discuss CUECARD, the program that won the 2008 Computer Olympiad computational pool tournament. Beside addressing intrinsic interest in a complex competitive environment with unique features, our goal is to isolate the factors that contributed to the performance so that the lessons can be transferred to other, similar domains. Specifically, we distinguish among pure engineering factors (such as using a computer cluster), domain-specific factors (such as optimized break shots), and domain-independent factors (such as state clustering). Our conclusion is that each type of factor contributed to the performance of the program.

1 Introduction

The International Computer Games Association (ICGA) has in recent years introduced computational pool as a new game to the Computer Olympiad. Billiards games have several characteristics that make them unique among games played by computer agents, and indeed among games in general. In particular, they have continuous state and action spaces, actions are taken at discrete-time intervals, there is a turn-taking structure, and the results of actions are stochastic. This combination of features is unique and differs from other competitive AI domains such as chess [Levy and Newborn, 1991], poker [Billings *et al.*, 2002], robotic soccer [Stone, 2007], or the Trading Agent Competition [Wellman *et al.*, 2007]. Thus, the challenge of billiards is novel and invites application of techniques drawn from many AI fields such as path planning, planning under uncertainty, adversarial search [Korf, 1995] and motion planning [Latombe, 1991].

In this paper, we discuss CUECARD, the winning program from the third ICGA computational pool tournament held in 2008. The aim of this paper is not to comprehensively describe CUECARD, but rather to determine answers to questions such as the following: To what extent was CUECARD's success due to engineering and brute force? How much did domain specific innovations and optimizations contribute to victory? And finally, what generalizable techniques and principles were used by CUECARD which might be instructive or productive when applied to other AI domains?

*This work was supported by NSF grants IIS-0205633-001 and SES-0527650.

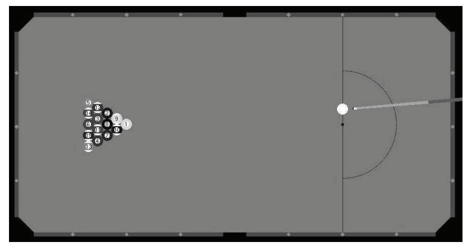

Figure 1: Pool table racked for 8-ball

This paper is structured as follows: In Section 2 we describe the official 8-ball rules and the computational pool setting. In Section 3, we give an overview of our agent design and emphasize some key elements unique to CUECARD. Section 4 contains a detailed experimental analysis of the individual contributions of CUECARD's components. Finally, in Section 5 we conclude with a discussion of the lessons learned from this domain.

2 Background

2.1 Rules of 8-ball

The game played in the ICGA computational pool tournament is 8-ball, based on the official rules of the Billiards Congress of America [1992]. 8-ball is played on a rectangular pool table with six pockets which is initially racked with 15 object balls (7 solids, 7 stripes, and one 8-ball), and a cue ball (see Figure 1). The play begins with one player's break. If a ball is sunk on the break, the breaking player keeps his or her turn, and must shoot again. Otherwise, the other player gets a chance to shoot.

The first ball legally pocketed determines which side (solid or stripes) each player is on. Players retain their turn as long as they call an object ball of their type and a pocket and proceed to legally sink the called ball into the called pocket. In case of a foul, such as sinking the cue ball into a pocket, the other player gets to place the cue-ball at any position on the table ("ball-in-hand"). After all object balls of the active player's side have been sunk that player must attempt to sink the 8-ball. At this point, calling and legally sinking the 8-ball wins the game.

2.2 Computational pool

The ICGA computational tournament is based on a client-server model where a server maintains the state of a virtual pool table and executes shots sent by client software agents on the POOLFIZ physics simulator [Greenspan, 2006]. Each agent has a 10 minute time limit per game to choose shots.

A shot is represented by five real numbers: v, φ, θ, a, and b. v represents the cue velocity upon striking the cue ball, φ represents the cue orientation, θ represents the angle of the cue stick above the table, and a and b designate the position on the cue ball where the cue stick strikes, which plays a big role in imparting spin, or "english", to the cue ball. Since the physics simulator is deterministic, and in order to simulate less than perfect skill, Gaussian noise is added to the shot parameters on the server side. The result of the noised shot is then communicated back to the clients.

In the 2008 competition CUECARD played against two opponents who have participated in previous competitions: PICKPOCKET [Smith, 2007] and ELIX[1]. PICKPOCKET was the champion of all previous competitions, and was thus the state of the art as we began designing CUECARD. In the tournament, each agent played a 39 game match against each other agent. CUECARD won the tournament with 64 wins (in 78 games), compared to only 34 wins total by PICKPOCKET.

This was not enough games to statistically establish the fact that CUECARD was the superior agent, and so, after the tournament, we ran more games between CUECARD and PICKPOCKET[2]. In this later match, CUECARD won 492-144, clearly establishing that it is the better player.

3 CUECARD description

CUECARD is a complex engineering artifact, and to describe it fully would require more space than we have here. Our goal is instead to give a good sense of how CUECARD works, to describe some of its main components, and to isolate for analysis factors that may have contributed to CUECARD's success.

3.1 CUECARD overview

Given a state of the table, CUECARD chooses the shot parameters to return by following these steps:

1. For each legal ball and pocket, a set of directions φ, each with a minimum velocity v_0, is generated in attempt to sink the ball into the pocket. In this step we generate both straight-in shots (where the object ball goes directly into the pocket), more complex shots (involving more than one collision), and special shots designed to disperse clusters of balls.
2. For each of these (φ, v_0) pairs, discrete velocity values, v_i, between v_0 for this shot and the maximum allowed velocity v_{MAX}, are generated. The (φ, v_i) pairs that are deemed feasible (i.e. pocket a ball with no Gaussian noise added) are passed to the next step.
3. For each feasible (φ, v_i) pair, variants are generated by randomly assigning feasible values to a, b and θ.
   (a) Each such variant is simulated between 25 and 100 times, depending on available time.
   (b) The resulting states (projected table state after shot simulations) are scored using an evaluation function, allowing the calculation of an average score for each shot variant. CUECARD uses the same evaluation function as PICKPOCKET, which is described in detail in [Smith, 2007].
   (c) The top two shot variants for each (φ, v_i) pair are selected.
   (d) For these top two variants, the states resulting from the simulations in Step 3a are clustered into groups of similar table states. A representative state is chosen for each cluster, and a weighted set of representative states is formed.
4. The top 20 shot variants among all shots tested are selected for further evaluation.
5. To refine the evaluation of each of these 20 shots, we execute a second level of search starting with the representative resulting states of these 20 shots. The search method used at this second level differs between the early and late stages of the games.
   (a) In the early game, the above process (Steps 1–3) is essentially repeated with smaller constants, returning the average evaluation for the best shot.
   (b) In the late game, a noiseless search is conducted up to the end of the game, as described in Section 3.4.
6. After the representative state evaluations have been adjusted, a new evaluation for each of the 20 shot variants is generated, and the best variant is chosen.
7. In case the best variant has a value below a certain threshhold, or no feasible shots were generated, CUECARD plays a last-resort shot (see below).

Last-resort shots are generated by attempting shots in all directions in an attempt to find shots that do not foul and hopefully pocket a ball. Generation of shots in this manner discovers different types of shots, such as double and triple kick shots, which were not considered during typical shot exploration.

A key facet of performance in a competitive environment is reasoning about the opponent. CUECARD considers the opponent only during evaluation of last-resort shots, while ignoring the opponent during typical shot selection. This consideration is not game-theoretic, and affects a relatively small component of CUECARD, as less than 1% of shots generated by CUECARD utilize the last-resort mechanism. When evaluating last-resort shots we consider it more likely that the opponent will have the next turn. Because of this, last-resort shots are evaluated by taking into account the value of the resulting state for both CUECARD (v_{CC}) and the opponent (v_{OPP}). The value of a last-resort shot, v_{LR}, is then

[1] ELIX was written by Marc Goddard.

[2] The authors thank Michael Smith, for making PICKPOCKET available to us.

$v_{LR} = v_{CC} - v_{OPP}$. In this manner, a shot which always sinks a ball and leaves CUECARD with good shots will be considered as valuable as a shot which never sinks a ball, but leaves the opponent with no available shots.

It is hard to analyze the contribution of the last-resort mechanism, as it is used so infrequently. Also, we cannot simply remove it, since not replacing it with another mechanism would certainly lead to worse performance, as there would be no way for the modified agent to deal with difficult situations. In light of this, analysis of the last-resort mechanism is not included in this paper. As described in the remainder of the paper, we picked components which are representative of the different design aspects (engineering, domain-specific and domain-independent), as well as amenable to analysis through testing. The following components were selected for analysis: a distributed architecture and reimplementation of the physics simulator as engineering factors, an optimized break shot as a domain-specific factor, and finally look-ahead search and a method of clustering the results of sampling as domain-independent factors. We first describe each of these components and then in Section 4 present analysis of their contributions to CUECARD's success.

3.2 Engineering components

As with any search problem, it seems clear that the more time an agent has to explore the problem space, the better the action that the agent will be able to select. "Engineering" components were designed to help CUECARD accomplish more in the same amount of time. The following factors did this by either making CUECARD more efficient, or by simply making more time available for CUECARD to use.

Distributed architecture

CUECARD was designed to have its computation distributed across multiple machines, effectively granting itself additional CPU time to compute shots. Specifically, Steps 3 and 5 are distributed among a cluster of 20 dual-core machines on Amazon's EC^2 cluster. Each sub-task is assigned a time limit and sent to a different machine with a manager coordinating the execution of the sub-tasks. Efficiency of this mechanism is tracked by CUECARD in real time and the time limits are adjusted accordingly.

Faster physics engine

Profiles of CUECARD's execution revealed that one of the major bottlenecks was the physics library, which was provided in binary format. To speed up CUECARD's execution we re-implemented the physics library based on [Leckie and Greenspan, 2006], while improving several engineering aspects of the software. This re-implementation resulted in an average 5.9× speedup when compared to the POOLFIZ library over several example games. To ensure proper performance, CUECARD only uses our physics to simulate the samples of a shot when the resulting states of the noiseless simulated shot on both simulators are sufficiently close together.

3.3 Domain specific component: break shot

A successful agent in any domain will require specific knowledge of and optimization for the domain in question. In this

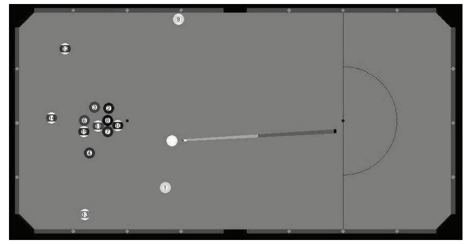

Figure 2: Typical table state after CUECARD's break

section we describe how CUECARD deals with a game situation specific to billiards: the break shot.

A break shot consists of specifying a location for the cue ball and a shot. CUECARD uses a precomputed break shot which was calculated via an extensive offline search over cue-ball locations and shot parameters. An ideal break shot would keep the turn and spread the balls out on the table. During testing it became clear that these two goals are to some degree mutually exclusive. There are break shots which keep the turn 97% of the time, but do not spread out any balls. Other break shots which spread balls out completely were at best ~60% successful. CUECARD's break shot put more weight on keeping the turn than spreading out the balls on the table, as the latter leaves the opponent in a very good position if the turn is also lost. CUECARD's break shot keeps the turn 92% of the time, and often spreads out many of the balls.

3.4 Domain-independent techniques

Some of the techniques used by CUECARD are applicable to other AI domains. These techniques are described here.

Shot sampling and state clustering

Early testing of CUECARD showed that the 15 samples used by PICKPOCKET were not enough to consistently choose successful shots. However, the consequence of sampling more, without clustering (Step 3d), is that there are a larger number of states resulting from any action (equal to the number of samples), each of which must be expanded at the second level of the search (Step 5) to refine the value estimate for the shot. Thus, any increase in the number of samples requires that more time be spent at the second level to evaluate each shot. Thus either fewer shots must be considered, or the second level of search must be eliminated completely.

To avoid these problems, CUECARD applies a clustering algorithm to the table states resulting from the different samples of a single shot. The clustering algorithm works with a state representation that includes the (x, y) coordinates of each ball on the table, as well as the evaluation of that table state. States that, according to a metric in this high- dimensional state space, are sufficiently close are combined into a "representative state" that can later be expanded. Similar sampling/clustering techniques have been previously applied to protein folding [Singhal *et al.*, 2004]. In our case, the clustering approach offers a dramatic reduction in the number of states that need to be processed at the second level of the search. Often, 50 states resulting from a single shot can be clustered into 5 or 10 states, which leaves much more time to evaluate each of those states at the next level. The additional samples of each shot lead to more robust shot selection. CUECARD samples each shot between 20 and 100 times, depending on the stage of the game.

Look-ahead search

CUECARD uses look-ahead search to plan the shots it takes. In the early game, it looks two levels deep, using the same evaluation function at each level, similar to what was done by PICKPOCKET [Smith, 2007]. In the late game, CUECARD evaluates states after the first level of search by estimating the probability that it wins the game from that state. This is done by conducting a noiseless search to the end of the game, with each shot weighted by its estimated success probability. When the end of the game is reached, the win probability for that state is 1. This value is then backed up the search tree to give value to the states where it is still CUECARD's turn. It is assumed that if a shot doesn't succeed then the same number of balls will remain on the table. The opponent's win probability for this state is estimated based on a Markov-chain model of 8-ball, indexed by the number of balls of each side in play, which was generated from thousands of self-played games.

4 Analysis

In this section the previously described components of CUECARD are analyzed and discussed to identify the importance of each to CUECARD's success.

4.1 Methodology

To isolate each individual component we created versions of CUECARD with the component either modified or removed. Each of these modified version was tested against the tournament version of CUECARD, running on a single CPU, with 600 seconds allotted per game. We will refer to this benchmark agent as CUECARD-TV. In some cases we also ran matches between the modified versions and the version of PICKPOCKET that competed in the tournament. In all cases, we report the win percentages of each program and the standard deviations[3](SD) of the results. We have also used hypothesis testing to test the statistical significance of our results when indicating that a certain version of our program performed better than another. These significance values[4] are provided when relevant.

4.2 Engineering components

In the tournament, each program was allotted 600 seconds per game for decision-making. The purpose of CUECARD's two engineering components was to allow the agent to do more computation in the time it had, either by using more machines or by making bottleneck components run faster. In order to determine the effect of these components on CUECARD's victory, we must first determine the value of this extra time.

Separate from the tournament, we ran several tests in an attempt to capture the value of time to CUECARD. Figure

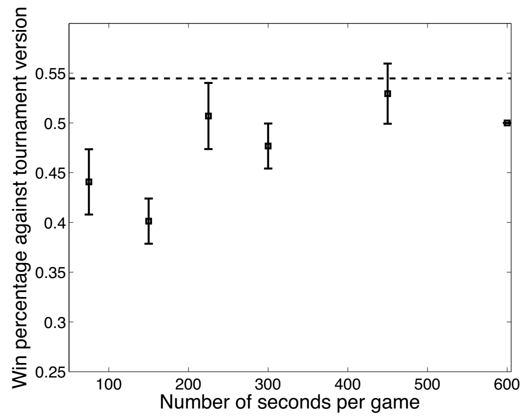

Figure 3: Effect of allocated time per game on performance

3 shows the results of these tests. Different numbers of seconds per game were allotted to the single-threaded version of our program. Each of these different versions then competed against CUECARD-TV, to which, as stated previously, was allotted the full 600 seconds. In each case the win percentage of the variant program against CUECARD-TV is reported. For reference, the win percentage of the full tournament version running on the 20-machine cluster against CUECARD-TV is shown by the dashed line.

Faster physics simulator

The reimplemented physics simulator gave an average six-fold speedup per simulation. CUECARD uses it instead of POOLFIZ for 97.95% of the shots. Since the physics simulation is the major bottleneck of the system, we would expect the overall speedup to be around $5\times$. We can see from Figure 3 that if we force CUECARD to only use POOLFIZ, that would be like giving the program only 120 seconds per game. From the data we might expect such a program to have a win percentage against CUECARD-TV between 40% and 45%. We ran a match to test this hypothesis, and indeed, CUECARD with only POOLFIZ won 42.4% ($\pm$2.68%) of the time (99.8% significance).

The computing cluster

Since CUECARD-TV won 45% of the time against the 20 CPU version, it is clear that the additional processing power of the computing cluster was not essential to CUECARD's tournament victory. To confirm this, we ran CUECARD-TV against the tournament version of PICKPOCKET, and CUECARD-TV won 77% of the time ($\pm$2%), which is similar to the win percentage from the tournament.

We conclude that CUECARD did gain benefits due to the engineering components, but it is unclear that the benefits outweigh the costs associated with the methods. Also, the improvement of CUECARD is not linear in the amount of time gained. Thus, while each of the engineering components has value when added individually, when both of the components are added together the improvement is less than the sum of the individual improvements. Due to this fact, in hindsight, we would consider carefully the benefits and cost of adding

[3] $\frac{1}{2\sqrt{N}}$, where N is the total number of games played in the match.

[4] $\varphi_{\mu=0.5,\sigma^2}(w)$, where w is the win rate of the winning program, $\sigma = \frac{1}{2\sqrt{N}}$, and φ is the normal cdf.

Agent | break vs. agent | break	Win %	SD
CC | CC vs. PP | PP	77.4%	2.0%
CC | PP vs. PP | PP	65.4%	2.4%
CC | CC vs. PP | CC	69.3%	1.6%
CC | CC vs. CC | PP	70.1%	5.1%

Table 1: Determining break shot contribution

both engineering components, and perhaps leave at least one of them out of the system.

4.3 Domain specific component: break shot

One stark contrast between PICKPOCKET and CUECARD is the break shot employed, each of which leads to very different distributions over the possible break shot outcomes. To examine the contribution of the break shot to CUECARD's success, we ran four separate test matches. In each test match, tournament versions of CUECARD and PICKPOCKET were used, with only their break shots modified. We first ran each agent with its own break shot, which recreated the conditions of the tournament. The results of this match are shown on the top line of Table 1. We next had the two agents compete against each other using the same break shot, first PICKPOCKET's and then CUECARD's. The results of these two matches are on lines 2 and 3 of Table 1. Lastly, we had CUECARD compete against itself, where one version had CUECARD's break shot, and one had PICKPOCKET's. The results of this match are shown on the bottom line of Table 1. All results are statistically significant for the winner.

At first glance, comparison of the top and bottom lines would indicate the possibility that CUECARD's win over PICKPOCKET can be accounted for almost entirely by the difference in break shots. Consideration of the other two matches quickly refutes that, however, as CUECARD was able to soundly defeat PICKPOCKET even when both programs were using the same break shot. The fact that the match on the top line, recreating tournament conditions, is the most lopsided in favor of CUECARD shows that the break shot did play a large role, accounting for around 10% of the wins. But, with break shots being equal, CUECARD was still demonstrably superior, and to explain this, we must examine other components of CUECARD.

4.4 Domain-independent techniques

Sampling and clustering

Two aspects of CUECARD's sampling were new contributions. The first, which was made possible by clustering, was simply that CUECARD performed more samples than the 15 which PICKPOCKET had previously done [Smith, 2007]. The second aspect was that CUECARD varied the number of samples performed depending on the stage of the game. To determine the effectiveness of each of these features we ran CUECARD-TV against versions of CUECARD that differed only in the number of times they sampled shots at the first level of search. In particular, each version was allotted the same amount of time per game. In each case the number of samples was held constant throughout all stages of the game.

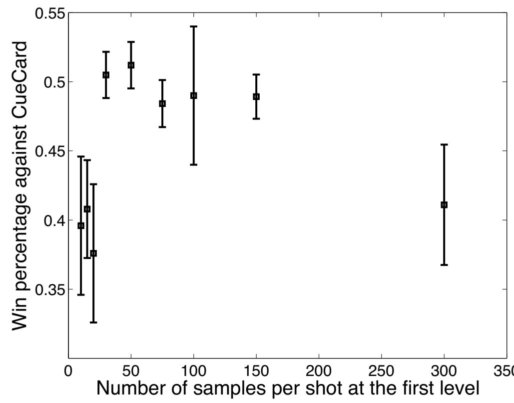

Figure 4: Effect of number of samples on performance

The results are summarized in Figure 4, where win percentages against CUECARD-TV are shown. We show the data in a graph, to emphasize the relationship between the different versions tested.

The performance of CUECARD-TV against the versions using less than 30 samples shows the effect of simply being able to sample more. CUECARD-TV won decisively against these programs, with an average win percentage of 61 %. Due to the low number of tests the statistical significance of each result is 97.7%.

For the range between 30-150 samples, which is approximately what CUECARD varies over, the results are too close to say definitely which method (constant or variable number of samples) is better. We can say that if CUECARD had a constant number of samples in the 30-100 range it would not have performed any worse. The simplicity of the constant number of samples approach argues for its use over the variable method. As the number of samples per shot increases to 300, the performance does decline. The tradeoff of sampling more is that the agent has less time to try different shots. It is clear from the data that once a certain number of samples has been reached further samples do not add to the performance of the agent. After a certain number of samples, the results of the samples seem to be an accurate enough representation of the possible outcomes of the shot, such that further samples don't give enough information to cause serious changes in decisions and play. The data indicates that this minimum number of necessary samples for our setting is somewhere between 30 and 50. We conclude that the clustering helped significantly by enabling more samples, while the variable number of samples gave us no additional advantage.

Second level of search

As described in Section 3.4, the methods used at the second level of search differ between CUECARD and PICKPOCKET. To see if one of these methods gives an advantage over another, we ran CUECARD-TV against a version of CUECARD in which the second level is the same as used by PICKPOCKET. The result of this match is shown on the top row of Table 2. Our conclusion is that neither method of searching at the second level gives significant advantage over the other, and that CUECARD's new method of searching at the second

Agent	Win % vs. CUECARD-TV	SD
L2 = PickPocket	49.88%	1.4%
Ignore level 2	45.23%	2.0 %
More level 1	51.60%	1.4%

Table 2: Effect of level 2 of search.

level did not contribute to the victory over PICKPOCKET.

This still does not address the usefulness of the level 2 of search in general. Until now, it has been generally assumed that in a search space with imperfect heuristic lookahead search would be beneficial. To see if this is true in the domain of computational pool, we did several things. First, we observed that the second level of search actually changes the decision made by the first level of the search only 18.33% of the time. We also ran two different test matches to see if running level 2 actually improves the decisions made by CUECARD. In each match, a modified version of CUECARD competed against CUECARD-TV (which utilizes level 2). In the first modified version everything remained the same, but the agent simply ignored the results of level 2, making its decision based only on the level 1 search result. The result of this match, shown on the second row of Table 2, shows that running level 2 after level 1 does improve the performance of the program (with statistical significance of 99.1%).

In the second modified version, instead of simply ignoring level 2, we took the time that would have been spent on level 2 and spent it exploring more shots at the first level of search. The results of this match are shown on the bottom row of Table 2. Interestingly, we see that running more level 1 is at least as valuable as running level 2, or at least the two methods are indistinguishable by the 1256 games we ran between them. This leaves unresolved the issue of whether a second level of search should be performed at all.

Analysis of PICKPOCKET [Smith, 2007] also failed to show conclusively that a second level of search was helpful. Interestingly, the tournament noise levels were chosen in an attempt to emphasize high-level planning and strategy. A second level of search would seemingly be invaluable in a domain which rewards good high-level planning. This choice has yet to be supported by experimental data. In the future we plan on specifically investigating this issue, seeking to either support or contradict the reasoning used to choose the tournament noise level.

5 Conclusions

Our analysis of CUECARD focused on three different types of components: engineering, domain-specific and domain-independent. We summarize our conclusions as follows:

- The two largest contributors to CUECARD's success, each with roughly the same magnitude of impact, were the break shot and increased number of samples.
- The engineering aspects, which increased the time available to CUECARD, had a smaller impact on its success.
- Look-ahead search and the variable number of shot samples had no effect, either positive or negative, on CUECARD's performance.

The lessons learned in this domain can be extended to other AI domains. Specifically, it is hard to predict, when facing a new and complex domain, which type of agent component will have the most impact. The experience with CUECARD shows that effort should be put into each different type, as they can all prove fruitful.

In the future, we plan to further explore the role of clustering in planning, especially compared to its use in other domains. We also plan to examine the relationship between noise level and the value of look-ahead search in billiards, examining what changes occur as the noise levels change. Similarly, in contrast with the idealized view of game theory, we currently do little in the way of opponent modeling. We would like to see how the need for opponent modeling changes with changes in the noise level. With all these open issues billiards presents a great opportunity to develop and apply a variety of AI techniques and methods.

References

[Billiards Congress of America, 1992] Billiards Congress of America. *Billiards: The Official Rules and Records Book*. The Lyons Press, New York, New York, 1992.

[Billings *et al.*, 2002] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence Journal*, 134:201–240, 2002.

[Greenspan, 2006] Michael Greenspan. PickPocket wins the pool tournament. *International Computer Games Association Journal*, 29:152–156, 2006.

[Korf, 1995] Rich Korf. Heuristic evaluation functions in artificial intelligence search algorithms. *Minds and Machines*, 5(4):489–498, 1995.

[Latombe, 1991] Jean-Claude Latombe. *Robot Motion Planning*. Springer, 1991.

[Leckie and Greenspan, 2006] Will Leckie and Michael Greenspan. *An Event-Based Pool Physics Simulator*, pages 247–262. Lecture Notes in Computer Science. Springer, 2006.

[Levy and Newborn, 1991] David Levy and Monty Newborn. *How Computers Play Chess*. Computer Science Press, 1991.

[Singhal *et al.*, 2004] Nina Singhal, Christopher D. Snow, and Vijay S. Pande. Using path sampling to build better Markovian state models: Predicting the folding rate and mechanism of a tryptophan zipper beta hairpin. *Journal of Chemical Physics*, 121:415–425, 2004.

[Smith, 2007] Michael Smith. PickPocket: A computer billiards shark. *Artificial Intelligence*, 171:1069–1091, 2007.

[Stone, 2007] Peter Stone. *Intelligent Autonomous Robotics: A Robot Soccer Case Study*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2007.

[Wellman *et al.*, 2007] Michael P. Wellman, Amy Greenwald, and Peter Stone. *Autonomous Bidding Agents: Strategies and Lessons from the Trading Agent Competition*. MIT Press, 2007.