MiniPool: Real-time artificial player for an 8-Ball video game

David Silva
INESC-ID and Instituto Superior Técnico,
Universidade de Lisboa
david.d.silva@tecnico.ulisboa.pt

Rui Prada
INESC-ID and Instituto Superior Técnico,
Universidade de Lisboa
rui.prada@tecnico.ulisboa.pt

ABSTRACT
The importance of artificial intelligence in games has been growing over the years due to the need of giving the player a more challenging experience. Games like 8-Ball offer many interesting challenges to both communities of AI and optimization due to the continuous and stochastic characteristics of the domain. To succeed, a player must be able to plan the best sequence of shots and execute a shot with accuracy and precision, so he does not lose the turn. There are already several artificial players, however they tend to take more than 30 seconds to select and execute a shot. Under a videogame setting, a player would give up playing the game if he had to wait that long for his turn. To address this problem, we propose a real-time solution for an 8-Ball artificial player using a Monte-Carlo Expectimax hybrid search algorithm with ray tracing techniques.

Author Keywords
Artificial Player; Billiards; 8-Ball; Stochastic Game; Video Game; Real-time

INTRODUCTION
Artificial players for 8-Ball games have been a topic of investigation due to interesting aspects that cannot be solved using the traditional methods of the classic games [5]. Having a continuous and stochastic domain makes it possible to have an infinite number of states and actions. It is also difficult to predict the resulting state of an action due to perturbations on the environment that cannot be controlled by the player (t is very unlikely that two shots with the same parameters lead to the same resulting states).

In the last years a competition called *Pool Computer Olympiads* has been taking place. In this competition participants had to develop an artificial billiards player and compete with each other. All these artificial players, which will be mentioned later in Section 3, focus on different aspects of the game and explore different points of view to overcome the design challenges found in 8-Ball.

One possible application of the state of the art in computer billiards is video games. The games industry has been growing over the years. According to the Entertainment Software Association [1] statistics in 2014, 155 million Americans play video games. Developing artificial intelligence for games brings other interesting challenges such as dealing with limited resources, providing a response in real time and balancing the skills. A professional pool player needs to have a good planning and understanding of the table state to make the best decision. For the artificial player to be able to do this and compete with the best human pool players, it will need the tools to plan the best sequence of shots, support a large variety of shot patterns, mechanisms to evaluate and find good reposition zones for the cue ball and methods to optimize the shot parameters under stochastic environments. The artificial players already developed for 8-Ball tend to take more than 30 seconds to plan the best shot to be executed. This delay on the response would probably make most players give up playing the game. In the context of this kind of games, players are expecting a real time response from the opponent.

The main focus of this work is to develop a real-time artificial 8-Ball player capable of competing with the best players while having a good balance between skill and resources used.

The rest of this document is structured as follows: first, in Section 2, we present a short background of 8-Ball and an overview of the characteristics of the game where the proposed solution will be tested. In Section 3, we introduce and discuss some of the related work already done. In Section 4, we give an overview of the proposed solution as well as some explanation of the key elements of MiniPool. The Section 5 contains all the experimental results and analysis of the individual contributions of each component of Minipool. Finally, in Section 6, there is an overall discussion and future work.

Videojogos'16, Nov 24–25, 2016, Covilhã, Portugal
Frutuoso Silva e Pedro Santos (eds.) ISBN 978-989-20-7148-0.

[1] http://www.theesa.com/

BACKGROUND

To provide a general idea of the challenges present in the 8-Ball game, we will describe the general rules used in the game, as well as the shot parameters and the most common shot patterns used by professional players.

8-Ball Rules

8-Ball belongs to the Pool-Billiards games family. It is a turn-based game played by two players on a rectangular pool table with 6 pockets and 16 balls (7 solid, 7 striped, the *8 ball* and the cue ball). The game begins with a player striking the cue ball anywhere behind the *headstring* (line with a semi-circle, illustrated in Figure 1) towards the cluster of the rest of the balls (called *break shot*).

A player can only strike the cue ball and only using the cue stick. To pocket a ball, the player can try to hit the object ball directly with the cue ball, hoping that the velocity gain from the collision is enough to make the object ball enter a pocket, or using collisions with other balls or rails to reach that object ball. The first ball to enter a pocket determines which ball type (solid or striped) the player needs to seek, having the opponent to seek the remaining type. The cue ball needs to hit a ball when stroked and that ball has to be of the type that the current player is seeking, otherwise the player is committing a *foul*. A *foul* also occurs when the cue ball enters a pocket.

When a ball enters a pocket legally, the player keeps the turn and must shoot again from the cue ball current position, otherwise the opponent gets the turn. In the case of a *foul*, the opponent also gets a *ball-in-hand*, which gives him the opportunity to place the cue ball anywhere on the table.

When the last ball of the current player enters the pocket, he needs to seek the *8 ball*. When the *8 ball* enters a pocket this way, and only at this situation, the player wins the game, but if it enters a pocket in any other situation the player loses the game immediately.

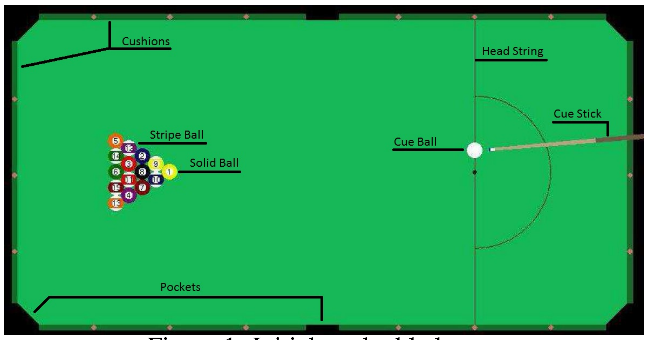


Figure 1: Initial pool table layout

Shot Parameters

In every Billiards variant, a shot is defined by five continuous parameters (illustrated in Figure 2):

- ϕ : Aiming angle,
- θ : Cue stick elevation angle,
- V : Initial cue stick impact velocity,
- a and b : Coordinates of the cue stick impact point on the cue ball.

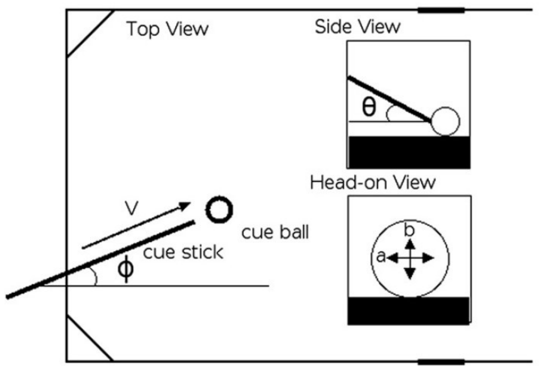


Figure 2: Parameters that define a shot in Billiards [17]

Shot Types

Pocketing, striking and kissing are some of the events that can occur in a regular 8-Ball match. The event that happens when a ball enters a pocket is called *pocketing*; a collision between a moving ball and a stationary ball with the purpose of moving the stationary ball is called *strike*; a shot whose purpose is just to adjust the trajectory of the moving ball is called a *kiss*. A shot can also consist in a combination of events. By combining them, human players can distinguish a variety of shot types. The following are the most common [5] (illustrated in Figure 3):

- **Break shot:** Initial shot, which has the purpose of dispersing the initial cluster of balls.
- **Direct shot:** The object ball is directly hit by the cue ball towards a pocket, without any other collisions involved.
- **Bank shot:** The rail is used to maneuver the object ball towards a pocket.
- **Kick shot:** The rail is used to maneuver the cue ball towards the object ball.
- **Combination shot:** A collision with another ball is used to attempt to pocket the object ball.
- **Pulk shot:** Similar to combination shot but the two object balls are very close to each other and align in a way that they are pointing to a pocket.
- **Kiss shot:** An additional ball is used to adjust the trajectory of another ball.
- **Safe shot:** This type of shot is used when the player assumes that he is more likely to lose the turn. Its purpose is to reposition the cue ball in such way that it makes it difficult for the opponent to continue.

RELATED WORK

Currently, there are already several artificial 8-Ball players; PickPocket [17, 18], CueCard [2, 1, 4] and PoolMaster [10, 11, 13, 14, 15], which participated in the *Pool Computer Olympiads*, and JPool [5].

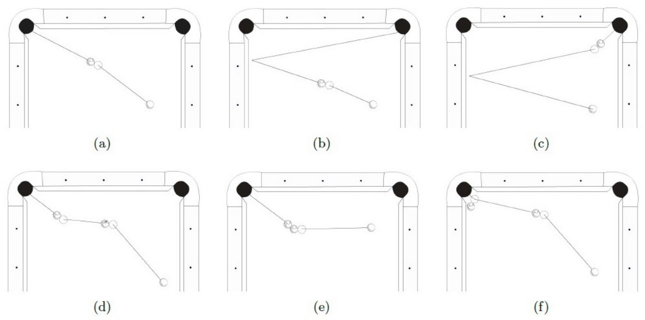


Figure 3: Shot types: (a) direct shot, (b) bank shot, (c) kick shot, (d) combination shot, (e) pulk shot, (f) kiss shot [5]

To better understand the differences among them, in this section, we will go through the most important topics, starting with a possible game model for 8-Ball found by Christopher Archibald *et al.*

Billiards Game Model

Christopher Archibald [3, 4] proved that billiards has a *pure stationary Markov perfect equilibrium* [7]. This means that when a player is selecting a shot to execute, he only needs to think about the current state of the game to get the optimal shot. At some state *s* of the game, he will not change his strategy concerning on how that state was reached, whether the game is at the beginning or at the end or whether the opponent is good or bad. When selecting a shot, we want it to pocket a ball and have the cue ball at a good reposition to continue playing. If that is not possible under our strategy, we want the cue ball to be at a position where a successful shot would be as difficult as possible for any opponent. With this in mind, we have theoretical proof that we only need to explore shots based on the current and future states (for position play) of the game to reach the optimal shot. Trying to understand who is our opponent and what are his actions on the game will not give us better results. If a strategy is optimal it will remain optimal regardless the opponent actions and strategy. However, the proposed model remains intractable due to the action space being continuous. To compute the value of a state, we would need to try all the possible actions, which are infinite. Thus, any approach still needs to perform an intelligent search space partitioning to overcome this problem.

Search Algorithms

For the particular case of 8-ball and considering stochastic environments, PickPocket [17, 18] and JPool [5] suggested *Expectimax* and *Monte-Carlo* as possible search algorithms.

Expectimax generates chance nodes for every action with a stochastic outcome. These chance nodes will be evaluated with their probability of occurrence. In the case of 8-Ball, a direct approach would mean a sum of over an infinite number of outcomes, each with a minuscule probability of occurring.

Monte-Carlo was used in almost all the artificial players that will be explained in Section 3.3, with the only exception of PoolMaster, since *Monte-Carlo* by sampling shots avoids the limitation of *Expectimax* and gives the developer more control over the algorithm complexity.

PoolMaster uses a heuristic based search algorithm to select the best sequence of shots. It clusters balls with the *K-Means* algorithm [16] to improve the search and explore less riskier shots first.

Shot Generation

8-ball has a continuous and stochastic domain nature, so it is impossible to enumerate all the possible shots. Generating only the most relevant shots for a particular situation is the key for an intelligent search space partitioning that allows to improve the overall performance of the program.

Since the shot generator algorithm differs for each one of the artificial players, we will present them individually.

PickPocket

Generates shots one type at a time in an increasing order of difficulty. Variations are generated by perturbing the original shot with the base velocity retrieved from a precomputed table. The break shot parameters are selected by sampling 200 shot variations and selecting the one which returns the best results. Safe shots are generated by perturbing V and ϕ and evaluating in the opponent's perspective. For Ball-in-hand situation, the table is discretized in a grid and every cell is assigned with the value of the best shot as if the ball was there, and then a *Hill-Climbing search* is performed in several random cells to find a local maximum.

CueCard

Similar to PickPocket, but it does not prioritize the shot types. It clusters similar resulting states with *K-Means* to reduce the state space. The Break Shot used was precomputed. The Ball-in-hand is similar to PickPocket but before discretizing the table, it tries to place the cue ball where the ghost-ball would be (see Section 3.4).

PoolMaster

The focus of PoolMaster lies on position play. First it generates all pairs ball-pocket possible given the current state of the table, then it analyzes the table for the possible next shots. Once this information is gathered, it calls an optimization algorithm to minimize an objective function that takes into account the distance to the next shot, as well as pocketing the target ball.

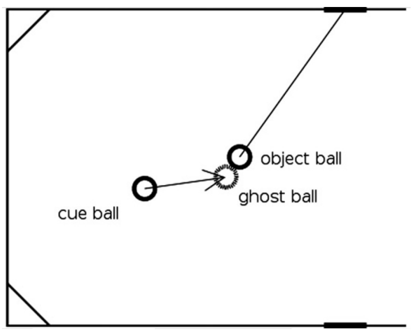
JPool

Takes a different approach, given that it models a shot as a series of steps like a tree, not limiting itself to predefined shot types. The break shot parameters were precomputed. For position play, JPool creates polygons around every ball where it would be a good place to pocket and then does a line-polygon overlap detection using the trajectory of the cue ball at maximum speed. The crossing zones are rated as if the cue ball was there and the cue ball is aimed to reach these areas. The rest of the parameters are discretized.

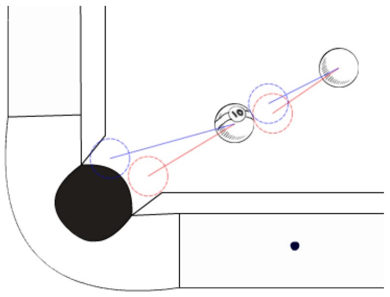
Aiming

PickPocket [17, 18] and CueCard [2, 4] use the traditional concept called *ghost ball* (illustrated in Figure 4a). If the cue ball is aimed in such a way that it hits the object ball in the position of the ghost ball, the object ball will travel in the direction of the target position.

PoolMaster [10, 11, 13, 14, 15] and JPool [5] use the same concept in a different way. Instead of aiming the object ball to the center of the pocket, they aim it to its limits, which gives them two ghost ball positions (illustrated in Figure 4). These leftmost and rightmost are adjusted to take into account the possible obstacles in the way, so, if the ball does not fit between these margins, the shot will be impossible [6].



(a) Ghost ball concept [17]



(b) Leftmost and rightmost concept [5]

Figure 4: Aiming concepts

Evaluation Function

A search algorithm evaluates shots to differentiate them and selects the best one for execution. There are several ways of measuring and differentiating shots from each other.

JPool [5] uses an algorithm based on Monte-Carlo search with a sample size of 400. The leaves are evaluated using a sum of several heuristics advised by a professional player [12] such as the quality of the current cue ball position, the number of balls in game and the difficulty of pocketing the other balls.

PickPocket [17, 18] and CueCard [2, 4] also use a Monte-Carlo search base algorithm, with 15 and 25 to 100 samples (depending on the time available), respectively. The leaves are evaluated using the sum of the probability of success of the best 3 shots retrieved from the precomputed table.

PoolMaster [10, 11, 13, 14, 15] calculates the value of a node using a function that takes into account the quality of the cue ball position, the probability of being in that position zone and the range of successful parameters with a sample of 15.

Shing Chua *et al.* explained in [8, 9] the calculation of the shot difficulty using fuzzy logic. The fuzzy sets were defined for the distance traveled by the cue ball before the collision with the target ball, the distance from the target ball to the pocket and the cut angle. They infer the rule for the specific shot situation during runtime. Only direct, bank and combination shots are considered and they are prioritized in this order.

IMPLEMENTATION

In this section, we will explain how we explored and implemented the main architectural components of our artificial player: search algorithm, shot generator and evaluation function.

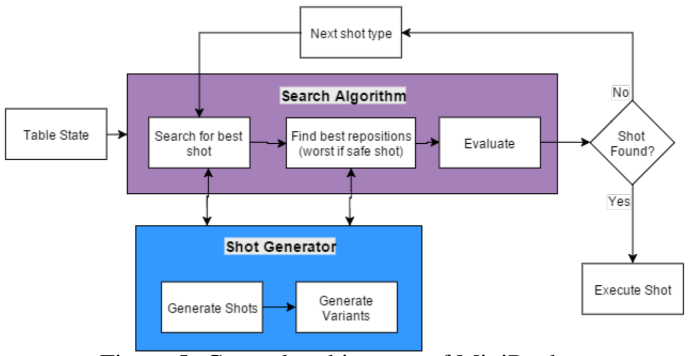


Figure 5: General architecture of MiniPool

Search Algorithm

The search algorithm is responsible for selecting the best sequence of shots. In this layer of the artificial player program, it is crucial to only explore the most relevant nodes of the tree to reach a solution, specially when the time constraints are tight.

Monte Carlo was selected as base for the search algorithm since it works very well in the 8-Ball domain, as it can be seen in the artificial players studied in Section 3.

Using the original *Monte Carlo* algorithm and evaluating each node as *success* or *failure*, being *success* pocketing the target ball in the target pocket and *failure* otherwise, we end with a sequence of shots rated with their probability of success. This approach highly depends on sampling and has a huge explosion of states generated since *Monte Carlo* performs a new search for every sample. For that reason, some pruning techniques were used. The easiest and more obvious technique is to stop the search when the probability of success of a given shot is under a certain threshold. Based on the results in [17, 18], shots besides the direct ones are only used 6% of the time. By generating shots for a given state of the table, one type at a time, in order of increasing difficulty, we can reduce even more the number of states generated when a direct shot is found.

Before the search starts, shots are rated and sorted based on their difficulty using a function from [8, 9]. This function is based on the distance between the balls and the pocket, and the cut angle, according to Equation 1.

$$\triangle = \frac{d_{co}d_{op}}{cos^2\alpha} \quad (1)$$

$\triangle$ is the shot difficulty, d_{co} is the distance between the cue ball and the target ball, d_{op} is the distance between the target ball and the pocket and α is the cut angle between the two balls. Since this formula penalizes longs shots and high cut angles, the shots that will be explored first will be the ones closer to the cue ball and less vulnerable to noise. This formula does not need to be very precise since it will only serve to differentiate the easiest shots from the others. By rating all the shots with this formula and sorting the list in crescent order, we are guaranteeing that the shots with the highest probability of success (less difficult) will be at the beginning. Since we are evaluating one type of shot at a time, we stop the search when the difficulty of a shot is too high and give an opportunity to the next shot type, trying to find an easier solution.

Although we have significantly reduced the state space with these modifications, there is still the case where all the shots are too difficult and the cut condition is never reached. PoolMaster optimizes the shot parameters for every next ball with a local optimization algorithm, which is the approach that achieves better results. Therefore, if we use the search only to find the next ball and the sampling only to evaluate the probability of pocketing and repositioning, we will not need to perform a search in every sample. We already know which ball we want to reach and the probability of succeeding the plan will already give us the difficulty of succeeding. The question is: which table state should we use to search for the next ball? In this case, we used the noiseless state, since it will be the average resulting position of the cue ball.

By using the reposition for the next ball as part of the evaluation, we are forcing a shot to be at a good reposition. This allows us, in an implicit way, to benefit shots that not only break clusters, but also to guarantee that the cue ball will be at the best position possible for the next ball.

Shot Generator

The shot generator is responsible for generating all the possible shots for a given table state. In this work a backtracking search algorithm with ray tracing was used, this kind of search for shots guarantees from the root that the shots will be executable, and will also give us for free a complete description of what will happen on the table, such as the balls and rails involved and distance traveled. This information is very important to control the complexity of the shots being generated and also gives us a tool to control the skill on the player in terms of tactic behavior for a single shot.

The general algorithm is the following:

1. For every pocket, set the objective point as its center:
2. Cast a ray from every ball to the objective point:
   - (a) If the ray reaches the objective point, set the objective point as the center of the ghost ball position for this ball.
     - i. If the cut angle is greater than a certain limit, stop iteration on this path.
     - ii. Else, if this ball is the cue ball, calculate the shot parameters and add it to the list of shots.
     - iii. Else, go to step 2.
   - (b) Else, stop iteration on this path.

Ball collisions with high cut angles are not explored to remove shots that barely touch the balls and that do not produce relevant results.

For the case of the bank and kick shots, we used the PoolMaster table mirroring method adapted for ray tracing approach. For every rail collision allowed, a level of mirrors is added to the raytracer object list. For example, with 1 rail allowed, we add 4 mirrored tables (one on the left, right, top and bottom of the original table); with 2 rails allowed, we add 12 tables (4 from the level 1 and 8 around level 1 for the level 2). With this information on the raytracer, we can treat bank and kick shots like direct and combinations shots. However, the higher the number of rails allowed, the slower the ray tracing will be due to the number of objects in the list; that is why the bank and kick shots are only explored after combination shots. When using this method, we are assuming that the angle of incidence is equal to the angle of reflection, which is not true in the *FastFiz* engine.

Using the proposed approach, by only controlling the depth of the search and the number of rail collisions allowed, we have a general algorithm for almost every shot type without having to explicitly look for it.

The calculation of the initial shot parameters is done as follows:

- θ is set to the minimum possible value.
- ϕ is calculated aiming the cue ball center to the ghost ball center (the objective point of the ball before the cue ball).
- a and b are set to zero.
- V is retrieved from a precomputed table of minimum velocities.

To quickly find the minimum velocity required to pocket a ball, for a given shot situation, a minimum velocities table is precomputed. PickPocket [17, 18] and CueCard [2, 1, 4] generated direct shot situations by discretizing the cut angle (the angle that the cue ball makes with the target ball), the distance between the cue ball and the target ball, and the distance of the target ball to the pocket and simulating shots incrementing the velocity by $0.1 m/s$ until the ball is pocketed. We generalized this to every shot type by discretizing the distance traveled by the sum of all the balls involved and the number of balls and rail collisions involved in the shot also.

At this point, we have a list of shots that put the target ball in the target pocket with the minimum velocity. There is an infinite number of variants of these shots that could still pocket the target ball. For position play, it is important to generate a set of shots that captures the range of possible follow-up states. The solution used to find the most significant variants was to pick n values equally spaced starting from the minimum parameter value up to the maximum for each shot parameter. The shots that accomplish the goal of pocketing the target ball are added to the shot list. Since the number of variants has a huge impact on the branch factor of the algorithm, a study of the most relevant parameters was made in Section 5.

Evaluation Function

The evaluation function is responsible for differentiating shots from each other with a specific metric. In MiniPool, the evaluation of a shot is made by counting the number of times a shot is successful while sampling it a number of times with noise. A shot is considered successful if it pockets the target ball in the target pocket and it has a clear way to the target reposition point. If, while calculating this probability of success, a shot cannot reach a minimum threshold of probability, the evaluation stops. This is done to reduce the computation time on poorly reliable shots.

With this evaluation function, we have a metric of how difficult it will be to execute a shot that leads to a good reposition for the next one. There is no need for anything else since, if a shot is more successful than another, it is because it will be less vulnerable to noise. However, this approach might make a ball closer to a pocket better than another. According to Jack Koehler [12] these balls should only be pocketed in special situations. Foreseeing these situations requires a better plan for a sequence of shots which, due to shot execution time constraints, could not be done in this work.

RESULTS

To demonstrate the quality and potential of the approach developed, the results of various tests are presented in this section as well as the environment in which the tests where made. In all the tests only one component is modified in order to better demonstrate the impact of it. The graphics in every test show the accumulated average of the clean success percentage of the table (to demonstrate that it stabilizes before the end of the test), the time per shot and the reason why the algorithm stopped the iteration (to understand what is causing it to stop). The clean success metric is the number of times the player was able to pocket all the balls without losing the turn divided by the number of games played. The time per shot metric is the time passed since the beginning of the turn until execute the shot (computation time).

The tests were made using the *FastFiz* engine. For each test, 500 table configurations were randomly generated and the algorithm was executed until it loses the turn. The average time until it executes a shot and the reason why the iteration stopped are stored for each table. The computer used for the tests has a Windows 10 Pro Operating System, Intel Core i5 CPU at 2.30 GHz and 4 GB of RAM.

In *FastFiz*, shots are affected by a perturbation model, noise. The standard deviation values for each parameter are: $\phi = 0.125°$, $\theta = 0.1°$, $V = 0.075$ m/s, $a = 0.5$ mm and $b = 0.5$ mm. In these tests, the simulations were made at 0x and 0.5x of these deviations.

The maximum cut angle is set to 70°, the maximum number of balls and rails involved in the shot generation is set to 3 and 1 respectively, the maximum shot difficulty is set to 0.7, the minimum success probability is set to 60% and the acceptable probability of success is set to 80%.

Analysis

Results with noise

In general, the clean success probability in a noisy environment is very low comparing with the other players. PickPocket is able to reach 67% within 60 seconds per shot, JPool reaches 74% with 44 seconds per shot and PoolMasters reaches 97% with 19 seconds per shot. Looking at Figure 6, we can see that the main problem causing the low results are shots that failed to pocket the target ball. This problem can occur for two reasons; either the shot parameters were wrong or the shot was risky. This test was made with a sample size of 25, a depth of 2 and 125 variants of V, ϕ and b (5 per parameter). Looking at the results in Figure 8, by using a bigger sample size, we can increase by 10% the clean success probability, taking 7 more seconds per shot. In Figure 7 we used 2 more variants per parameter, resulting in 343 variants per shot. As expected, this did not change significantly the success of the player. For the change in the number of variants to be relevant, the gap between each parameter variant needed to be as small as possible, however for this gap to be small enough we would need much more than 7 variants per parameter. A possible solution in order to not depend on shot variants would be an optimization algorithm like PoolMaster did. For each ball-pocket combination it makes an optimization search to find the parameters that pocket the ball and reach a good position. This kind of approach however, relies on an objective function that needs to have a finer discretization for the algorithm to find a result faster. Finding such function in 8-Ball domain it is not an easy problem.

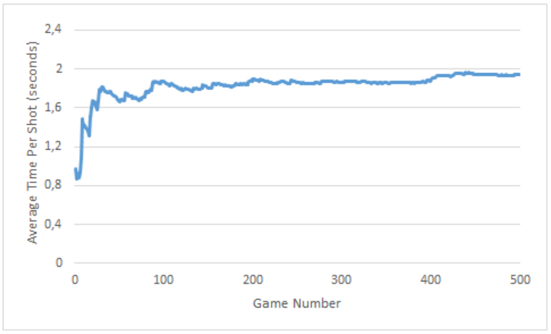
Results without noise

The results without noise were good comparing to others: JPool and PoolMaster are able to achieve 100% in approximately 44 and 19 seconds respectively. With a depth of 2 and 125 variants, MiniPool can reach clean table success of 86% in less than half a second. Using all shot types, we can have an improvement of 5% with a cost of 4 more seconds.

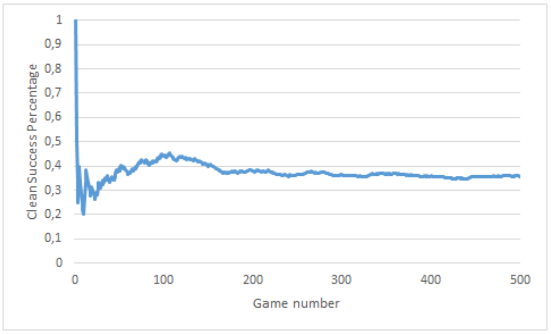
Since there is no noise in these tests, the reason for the algorithm to not be able to clean the table is probably a planning problem. To check this, a test was made using a depth of 3 (see Figure 11), however the results did not improve. Given that the results in Figure 10 were better, this drives us to conclude that there are situations when the algorithm is not able to continue because the target ball or pocket is obstructed. This situation prevents a direct shot from being executed for that ball. Since the algorithm searches for one type at a time, another direct shot will be chosen, and the next sequence of shots might put the cue ball in a situation where the algorithm cannot place it near the problematic ball again or reach that pocket. When we generate all shots at once, the algorithm will probably find a situation where the ball can be pocketed using another type of shot and solve the problematic ball.

CONCLUSION

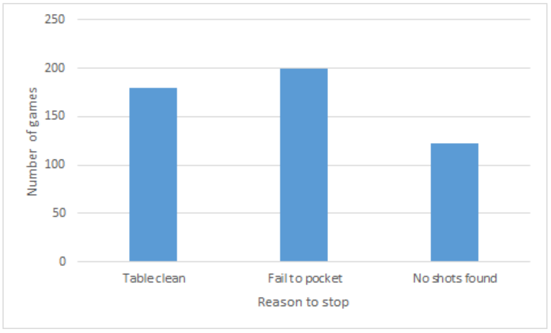
The main purpose of this work was to develop an artificial player for 8-Ball video game with a real-time response. Looking at the results of the tests without noise, we consider that this goal was achieved since videogames normally do not have noise perturbations. By ordering the shots by difficulty, taking



(a) Average time per shot stabilizes at 1,94 seconds



(b) Table clean success percentage, Stabilizes at 35.8%



(c) Motive for the algorithm to stop the iteration

Figure 6: Test results for 25 samples with 0.5x noise

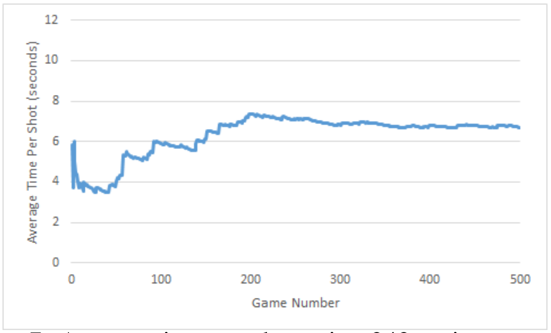
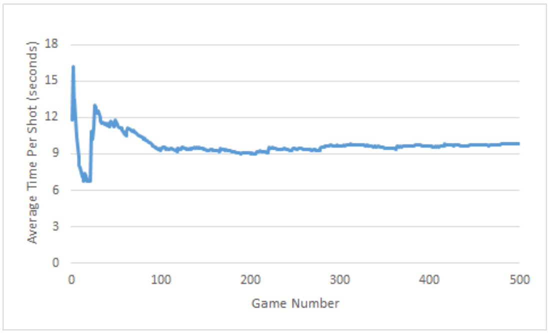
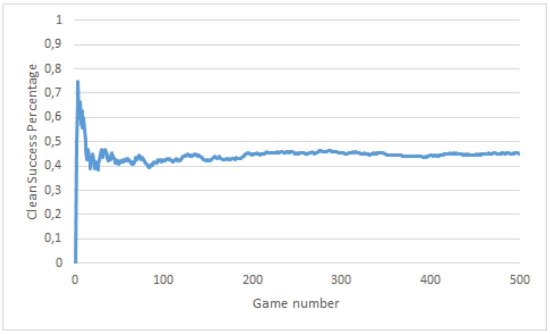


Figure 7: Average time per shot using 343 variants per shot. Stabilizes at 6.69 seconds with 0.5x noise

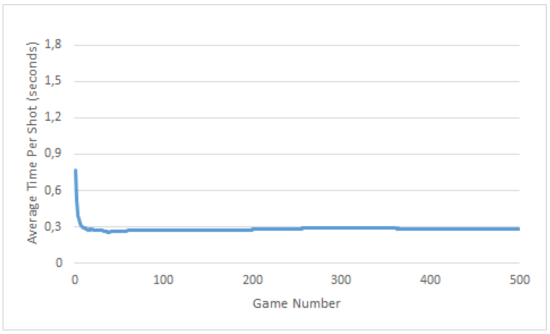


(a) Average time per shot stabilizes at 9,74 seconds

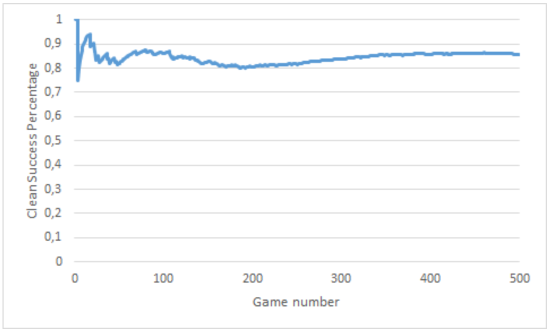


(b) Table clean success percentage, Stabilizes at 45.2%

Figure 8: Test results for 100 samples with 0.5x noise



(a) Average time per shot stabilizes at 0.29 seconds

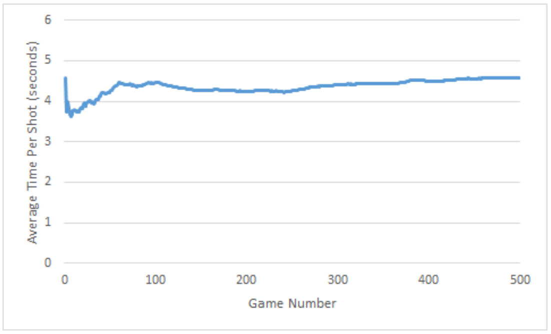


(b) Table clean success percentage, Stabilizes at 86%

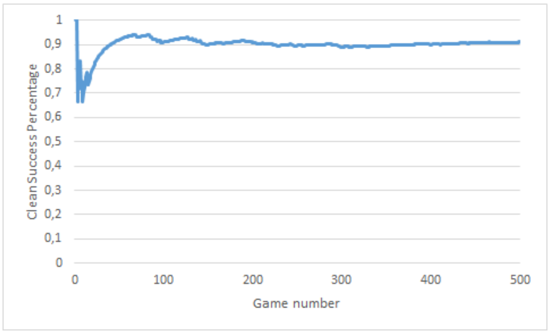
Figure 9: Test generating shot types one by one without noise

into account the distance of the balls, we are able to clear the table by zones. Combining this with the evaluation function, which benefits shots that are in a good position for the next ball, MiniPool can clear almost every table in less than half a second without having to search deeper in the tree.

On the other hand, a goal that was also in mind when developing MiniPool was to develop a player that could play in a environment with noise. The results for this case were not as good as expected and there are still some improvements that

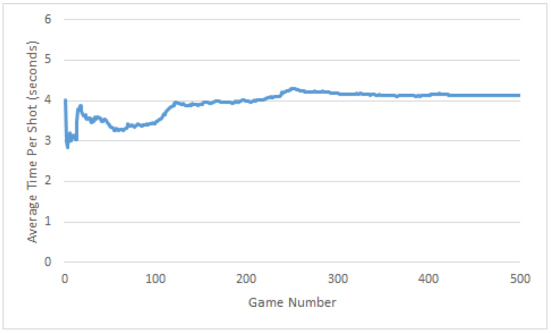


(a) Average time per shot stabilizes at 4.57 seconds

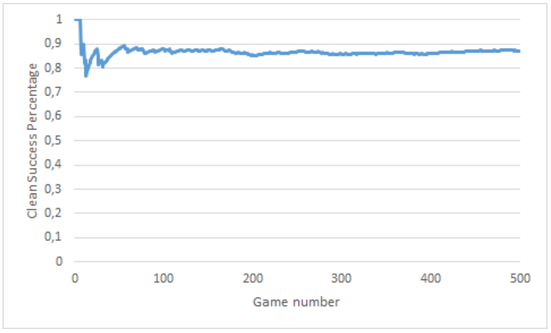


(b) Table clean success percentage, Stabilizes at 91%

Figure 10: Test generating all shot types without noise



(a) Average time per shot stabilizes at 4.12 seconds



(b) Table clean success percentage, Stabilizes at 87%

Figure 11: Test with a depth of 3 without noise

can be done as future work. One of the main problems of the algorithm was relying too much on the number of shot variants for the look-ahead. PoolMaster, by using the optimization approach, removed this dependence and could expend more time generating a more robust shot. Using a similar approach in MiniPool might be the solution to improve the performance in a environment with noise.

Other improvements that can be done to reduce the time needed to generate the shots for a given state are the raytracing acceleration techniques, such as kd-trees. By using these techniques we can reduce the number of objects that need to be tested for collision, and optimize the performance of the raytracer. This optimization will probably allow us to generate all shot types at once with a lower cost in time.

MiniPool was developed to be highly configurable and give a complete control of its skill. Since MiniPool was developed to be used as an artificial opponent in an 8-Ball video game, it would also be interesting to study how to simulate several types of skill or even automatically adapt the skill to its opponent.

ACKNOWLEDGMENTS

Thank you Vânia Mendonça, Soraia Meneses Alarcão, Fabio Alves, Élvio Abreu, Ruben Rebelo, João Moreira, Luis Sampaio, Liliana Santos, Joaquim Silva, Simone Silva, Beatriz Branco, Conceição and Carla Branco for your feedback and support.

This work was supported by national funds provided through Fundação para a Ciência e a Tecnologia (FCT) (UID/CEC /50021/2013).

REFERENCES

1. Christopher Archibald, Alon Altman, Michael Greenspan, and Yoav Shoham. 2010. Computational pool: A new challenge for game theory pragmatics. *AI Magazine* 31, 4 (2010), 33–41.

2. Christopher Archibald, Alon Altman, and Yoav Shoham. 2009. Analysis of a Winning Computational Billiards Player.. In *IJCAI*, Vol. 9. Citeseer, 1377–1382.

3. Christopher Archibald and Yoav Shoham. 2009. Modeling billiards games. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 193–199.

4. Christopher James Archibald, Yoav Shoham, Jurij Leskovec, and Robert Wilson. 2011. *Skill and Billiards*. Stanford University.

5. Jens-Uwe Bahr. 2012. A computer player for billiards based on artificial intelligence techniques. (2012).

6. Nicolas Bureau. 2012. Sensibilité des coups au billard. *CaMUS, Université de Sherbrooke, Sherbrooke, QC, Canada* (2012), 9–26.

7. Subir K Chakrabarti. 2003. Pure strategy Markov equilibrium in stochastic games with a continuum of players. *Journal of Mathematical Economics* 39, 7 (2003), 693–724.

8. Shing Chyi Chua, Eng Kiong Wong, and Voon Chet Koo. 2005. Intelligent pool decision system using zero-order

sugeno fuzzy system. *Journal of Intelligent and Robotic Systems* 44, 2 (2005), 161–186.

9. Shing Chyi Chua, Eng Kiong Wong, and Voon Chet Koo. 2007. Performance evaluation of fuzzy-based decision system for pool. *Applied Soft Computing* 7, 1 (2007), 411–424.
10. Jean-Pierre Dussault and Jean-François Landry. 2005. Optimization of a billiard player–position play. In *Advances in Computer Games*. Springer, 263–272.
11. Jean-Pierre Dussault and Jean-François Landry. 2006. Optimization of a billiard player–tactical play. In *International Conference on Computers and Games*. Springer, 256–270.
12. Jack H Koehler. 1995. *The Science of Pocket Billiards*. Sportology publications.
13. Jean-François Landry and Jean-Pierre Dussault. 2007. AI optimization of a billiard player. *Journal of Intelligent and Robotic Systems* 50, 4 (2007), 399–417.
14. Jean-François Landry, Jean-Pierre Dussault, and Philippe Mahey. 2012. A robust controller for a two-layered approach applied to the game of billiards. *Entertainment Computing* 3, 3 (2012), 59–70.
15. Jean-François Landry, Jean-Pierre Dussault, and Philippe Mahey. 2013. A heuristic-based planner and improved controller for a two-layered approach for the game of billiards. *IEEE Transactions on Computational Intelligence and AI in Games* 5, 4 (2013), 325–336.
16. James MacQueen and others. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA., 281–297.
17. Michael Smith. 2006. Pickpocket: an artificial intelligence for computer billiards. In *Masters Abstracts International*, Vol. 45.
18. Michael Smith. 2007. PickPocket: A computer billiards shark. *Artificial Intelligence* 171, 16 (2007), 1069–1091.