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

David Silva
Computer Science and Engineering
IST/UL, Lisbon, Portugal
E-mail: david.d.silva@tecnico.ulisboa.pt

***Abstract*—The importance of artificial intelligence in games has been growing over the years due to their continuous increasing realism and the need to keep their immersiveness while playing. Games like 8-Ball offer many interesting challenges to both communities of AI and optimization duo 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 good artificial players developed, however they tend to take more than 30 seconds to select and execute a shot. Under normal circumstances a player would give up playing the game if he had to wait that long to play.

In this document I propose a real-time solution for an 8-Ball artificial player using a Monte-Carlo Expectimax hybrid search algorithm with raytracing techniques.

***Index Terms*—Artificial Player, Billiards, 8-Ball, Stochastic Game, Video Game, Real-time**

I. Introduction

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

In the last years took place the *Pool Computer Olympiads* [1] where participants had to develop an artificial billiards player and compete with each other. All these artificial players, that will be mentioned later in section III, focus in different aspects of the game and explore different points of view to overcome the design difficulties found in 8-Ball.

One way of applying the *state of the art* in computer billiards and further develop it in a more specific case, is to use it on video games. The games industry has been growing over the years. According to the Entertainment Software Association [2] statistics in 2014, 155 million Americans play video games; 62% on PC, 56% on a dedicated game console and 35% on a smartphone.

Developing artificial intelligence for games brings other interesting challenges such as the limited resources, the real-time response and the skill balance. 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 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 make every player give up playing the game. In the context of games players are expecting a real time response from the opponent.

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

The rest of this document is structured as follows. First, in section II, I present a short background of 8-Ball and an overview of the characteristics of the game were the proposed solution will be tested. In section III I introduce and discuss some of the related work already done. In section IV I give an overview of the proposed solution as well as some explanation of the key elements of MiniPool. The section V contains all the experimental results and analysis of the individual contributions of each component of Minipool. Finally in section VI there is some overall discussion and future work.

II. Background

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

A. 8-Ball Rules

8-Ball belongs to the Pool-Billiards games family. It's a turn-based game played by two players on a rectangular pool table with 6 pockets and 16 balls (7 solids, 7 stripes, 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 it needs to be done with the cue stick. To pocket a ball, the player can try to hit the object ball directly with the cue ball, hoping that

[1] http://web.stanford.edu/group/billiards/index.html
[2] http://www.theesa.com/

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 stripe) 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 needs to be of the type that the current player is seeking, otherwise it is called 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 type of the current player enters the pocket, he needs to seek for 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 automatically.

The complete game rules of 8-Ball and other variations of Billiards can be found in [6].

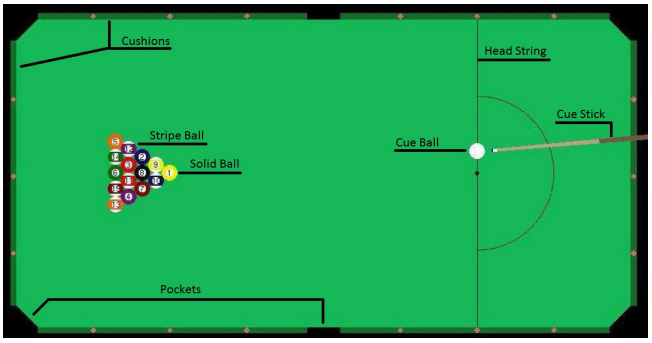


Fig. 1: Initial pool table layout

B. 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.

C. 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 *pocking*; when a moving ball collides with a stationary ball with the purpose of moving the stationary ball is called *strike*; when the purpose is just to adjust the trajectory of the moving ball is called a *kiss*. A shot can also consist in combinations 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):

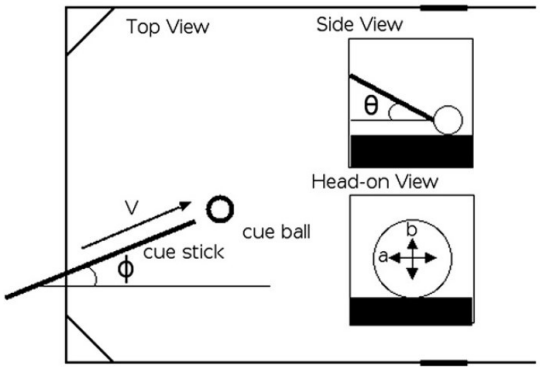


Fig. 2: Parameters that define a shot in Billiards (image from [19])

- **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. The purpose of it is to reposition the cue ball such that it will be difficult to the opponent to continue.
- **Break shot:** Is the name of the initial shot which have the purpose of disperse the initial cluster of balls.

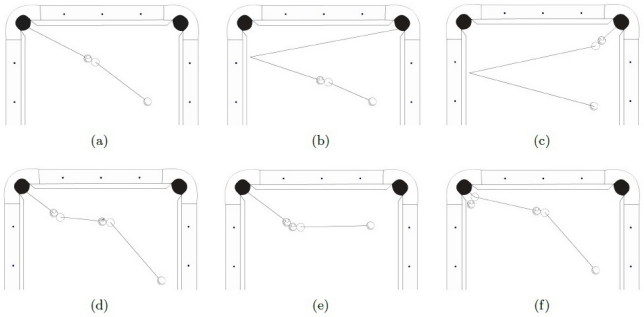


Fig. 3: Shot types: (a) direct shot, (b) bank shot, (c) kick shot, (d) combination shot, (e) pulk shot, (f) kiss shot (image from [5])

D. 8-Ball video game

I developed a simple 8-Ball 2D video game to test the artificial player. The game uses the *FastFiz* [3] physics engine (the same engine used in [1]–[3], [5], [11], [12], [14]–[16], [19], [20]), which is based on the simulator used in the past

[3]http://web.stanford.edu/group/billiards/FastFiz/

Pool Computer Olympiads [4], PoolFiz [17]. This engine has the particularity of having an event based physics system implemented which allows to simulate shots faster than a time based one.

In this video game the shot needs to be executed in less than 20 seconds, otherwise the player will lose the turn. The game can be set to use or not the noise models defined by FastFiz.

III. Related Work

Currently, there are already several artificial 8-Ball players developed; PickPocket [19], [20], CueCard [1]–[3] and PoolMaster [11], [12], [14]–[16], who participated in the *Pool Computer Olympiads*, and JPool [5]. To better understand the differences between them I will explain the most important topics one by one, starting with a possible game model for 8-Ball found by Christopher Archibald *et al.*

A. Billiards Game Model

Christopher Archibald *et al.* proved in [1], [4] that billiards has a *pure stationary Markov perfect equilibrium*. 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 reach, if the game is at the beginning or at the end or if 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 and that 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 using the proposed equation, 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.

An explanation of this model for stochastic games in general can be found in [8].

B. Search Algorithms

For the particular case of 8-ball and considering stochastic environments, PickPocket [19], [20] and JPool [5] suggested *Expectimax* and *Monte-Carlo* as possible search algorithms.

Expectimax generates chance nodes for ever action with a stochastic outcome. This chance nodes will be evaluated with its probability of occurrence, which 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.

[4]http://web.stanford.edu/group/billiards/index.html

Monte-Carlo was used in all the artificial players that will be explained in section III-C with the exception of PoolMaster, since that *Monte-Carlo* by sampling shots avoid 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. They cluster balls with the *K-Means* algorithm [18] to improve the search and explore less riskier shots first.

C. Shot Generation

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

Since the shot generator algorithm differs from each one of the artificial players, I will explain them individually.

1) PickPocket [19], [20]: Generates shots one type at a time in order of increase 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 better 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 the ball was there and then an *Hill-Climbing search* is performed in several random cells to find the local maximum.

2) CueCard [1]–[3]: Similar to PickPocket but does not prioritize the shot types. Cluster similar resulting states with *K-Means* to reduce state space. The Break Shot used was precomputed. The Ball-in-hand is similar PickPocket too but before discretizing the table it first tries to place the cue ball where the ghost-ball would be (see section III-D).

3) PoolMaster [11], [12], [14]–[16]: The focus of PoolMaster lies on position play. First they generate all pairs ball-pocket possible given the table state, then they analyze the table for the possible next shots. Once this information is gathered they call an optimization algorithm to minimize an objective function which have into account the distance to the next shot as well as pocketing the target ball.

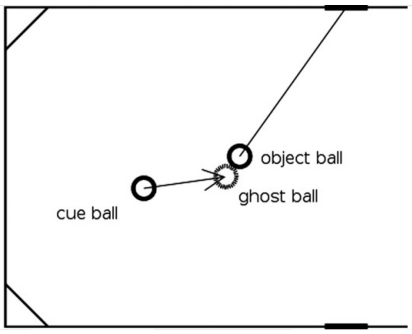
4) JPool [5]: Takes a different approach, given that it models a shot as a series of steps like a tree, being not limited to predefined shot types. The break shot parameters were precomputed. For position play JPool creates polygons around every ball where it would be a good place to pocket and then does a line-polygon overlap detection using the trajectory of the cue ball at maximum speed. The crossing zones are rated as the cue ball was there and the cue ball is aimed to reach these areas. The rest of the parameters as discretized.

D. Aiming

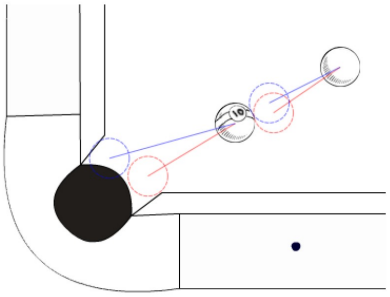
PickPocket [19], [20] and CueCard [1], [3] use the traditional concept called *ghost ball* (illustrated in figure 4a). If the cue ball is aimed in such a way that hits the object ball

in the position of the ghost ball, the object ball will travel in the direction of the target position.

PoolMaster [11], [12], [14]–[16] 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 the limits of the pocket, which gives them two ghost ball positions (illustrated in figure 4b). These leftmost and rightmost are adjusted to have into account the possible obstacles in the way, so if the ball does not fit between these margins the shot will be impossible. A better explanations of this concept can be found in [7].



(a) Ghost ball concept (image from [19])



(b) Leftmost and rightmost concept (image from [5])

Fig. 4: Aiming concepts

E. Evaluation Function

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

JPool [5] uses a Monte-Carlo search base algorithm with a sample size of 400. The leaves are evaluated using a sum of several heuristics advised in [13] such as the quality of current cue ball position, the number of balls in game and the difficulty of pocketing the other balls.

PickPocket [19], [20] and CueCard [1], [3] use a Monte-Carlo search base algorithm too, 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 [11], [12], [14]–[16] 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.* shown in [9], [10] 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. On runtime they infer the rule for the specific shot situation. Only direct, bank and combination shots are considered and they are prioritized in this order, respectively.

IV. Implementation

In this section I will explain the implementation of the artificial player developed for the propose of this document. As mentioned in section III there are three main components in every search-based architecture: search algorithm, shot generator and evaluation function. I will explain in the following topics how each one was explored and implemented.

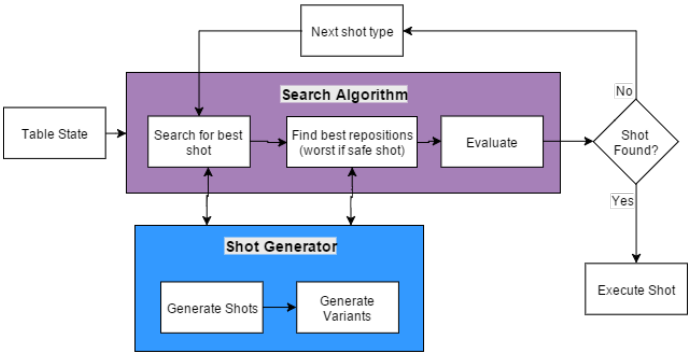


Fig. 5: General architecture of MiniPool

A. Search Algorithm

The search algorithm will be responsible for selecting the best sequence of shots. At this level of the artificial player program it is crucial to only explore the most relevant nodes of the tree to reach a solution, especial 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 can be seen in the artificial players studied in section III.

As first iteration lets start by evaluating each node as success or failure, being success pocketing the target ball in the target pocket and failure otherwise. With this evaluation we are selecting for execution the sequence of shots that has the highest probability of success. The main problem with this approach is that it highly depends on sampling and has a huge explosion of states generated since *Monte Carlo* performs a new search at every sample. The only option to reduce the explosion of states is to use some pruning. The easier and more obvious pruning is to stop the search when the probability of success of a given shot is under a certain threshold, however this is not enough. Based on the results in [19], [20], shots besides the direct ones are only used 6% of the time. By generating shots for a given table state one type at a time in order of increasing difficulty we can reduce enormously 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 [9], [10]. This function to evaluates shots based on the distance between the balls and the pocket and the cut angle.

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

Where $\triangle$ 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 doesn't need to be very precise since it will only serve to differentiate the easiest shots from the others. Rating every shots with this formula and sorting the list in crescent order we are guaranteeing that the shots with the highest probability of success will be at the beginning. Since we are evaluating shots one type at a time it is wise to stop the search when the difficulty of a shot is too high and start searching another shot type because a easier solution will probably be found.

Although we have reduce the state space a lot with these modifications, there is still the case where the shots are all too difficult and the cut condition is never reached. PoolMaster optimize the shot parameters for every next ball with a local optimization algorithm, which is the approach that have better results. So, if we use the search only to find the next ball, and the sampling only to evaluate the probability of succeed the plan of pocketing the target ball and repositioning for the next ball we won't need to perform a search in every sample, because 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, what table state should we use to search for next ball? The answer is, the noiseless state, because 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 have to be at a good reposition. This allow us in implicit way to benefit shots that not only break clusters but also guarantee that the cue ball will be at the best position possible, given the situation, for the next ball.

B. Shot Generator

The shot generator will be responsible for generating all the possible shots for a given table state. For the purpose of having a more feasible shot oriented generation it was used a backtracking search algorithm with ray-tracing. With this kind of search for shots we are guaranteeing 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 if very important to control the complexity of the shots being generated and also give 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 the 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 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 barley touch the balls and do not produce relevant results.

For the case of the bank and kick shots it was used the PoolMaster [11], [12], [14]–[16] table mirroring method adapted for raytracing approach. For every rail collision allowed it is added to the raytracer object list a level of mirrors, 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 raytracing 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. 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. The solution to this problem will be explain later on this section.

As you can see, 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 explicit look for it.

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

- $theta$ is set to the minimum possible value.
- phi 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 for a given shot situation to pocket a ball it was precomputed a minimum velocities table. PickPocket [19], [20] and CueCard [1]–[3] done this generating direct shot situations by discretizing the cut angle (the angle that the cue ball does 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. I 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 too.

At this point we have a list of shots that puts the target ball in the target pocket with the minimum velocity. There are 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.

C. 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 the evaluation stops. This is done reduce computation time on low reliable shots.

With this evaluation function we have a metric of how difficult it will be to execute a shot and be at 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. This approach however might make a ball closer to a pocket better than another. According to Jack Koehler in [13] these balls should only be pocket in special situations. But foreseeing these situations requires a better plan for a sequence of shots which, due to shot execution time constraints, cannot be done in this work.

V. Results

To demonstrate the quality and potential of the approach develop for the purpose of this document, the results of various tests are presented in this section as well as the environment in which the tests where made. In every tests only one component is modified in order to better demonstrate the impact of it. The graphics in every tests show the accumulated average of the clean success percentage of the table and the time per shot to demonstrate that the value being shown stabilizes before the end of the test, and reason why the algorithm stop the iteration to better understand whats causing it to stop and how to improve it.

The tests were made using the *FastFiz* engine. For each test, 500 random tables 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 operative 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 deviations for each parameter are: ϕ = 0.125°, θ = 0.1°, V = 0.075 m/s, a = 0.5 mm and b = 0.5 mm. In these tests the simulations were made with at 0x and 0.5x of these deviations.

The maximum cut angle is set to 70°, the maximum number of balls involved is set to 3, the maximum number of rails involved is set to 1, 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%.

A. Analysis

1) Results with noise: In general the clean success probability in a noise 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 35 seconds per shot. But given the time constraints that this project was developed for it is difficult reach those values too. However, looking at 6 we can see that the main problem for the low results are shots that failed to pocket the target ball. This problem can occur for 2 reasons; or the shot parameters were wrong or the shot was risky. But since the search algorithm only selects shots that accomplish the objective of pocketing the ball and that have a probability of success greater than 60% we can exclude the first reason. Looking at the results in 8 and 9, by taking 7 more seconds per shot we can increase by 10% the clean success probability using a bigger sample size.

In 7 we used 2 more variants per parameter, resulting in 343 variants per shot. As expected this did not change much 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 meed much more than 7 variants per parameters. A possible solution to not be dependent on shot variants is using 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, relays on an objective function that needs to be as much continuous as possible for the algorithm to find a result faster. Finding such function in 8-Ball domain it is not an easy problem.

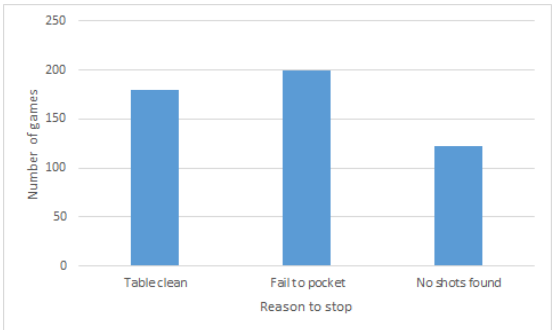


Fig. 6: Motive for the algorithm to stop the iteration with 0.5x noise

2) Results without noise: The results without noise are very good comparing with the others, JPool and PoolMaster are able to achieve 100% in 44 and 18 seconds respectively. With

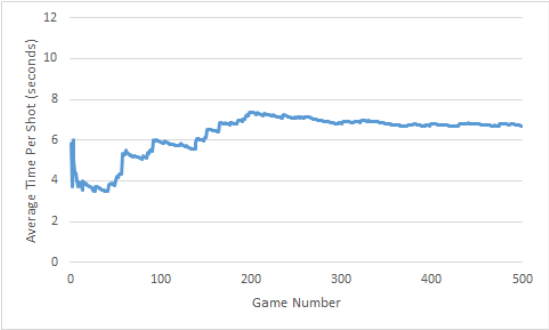


Fig. 7: Average time per shot using 343 variants per shot. Stabilizes at 6.69 seconds with 0.5x noise

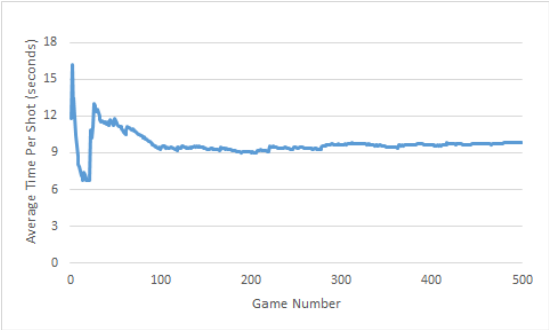


Fig. 8: Average time per shot with a sample size of 100. Stabilizes at 9.79 seconds with 0.5x noise

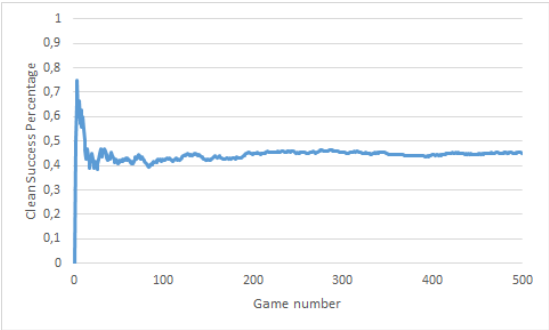


Fig. 9: Table clean success percentage with a sample size of 100. Stabilizes at 45.2% with 0.5x noise

a depth of 2 and 125 variants MiniPool can reach a 86% clean table success in less than half a second. Using all shot types we can have an improvement of 5%, which, given the time cost of four more seconds is not worth it in my opinion.

Since there is no noise in these tests the reason for the algorithm to not be able to clean the plan is probably a planning problem. To check this it was made a test using a depth of 3, 15, however the results did not improved at all. My guess for this, and looking at the results in 13 where the results were better, is that the situations where the algorithm is not able to continue are when the pocket its obstructed by the opponent balls. This situation prevents a direct shot from being executed for that ball, and since the algorithm search 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. When we generate all shots at once, the algorithm will probability find a situation where the ball can be pocket to another pocket earlier.

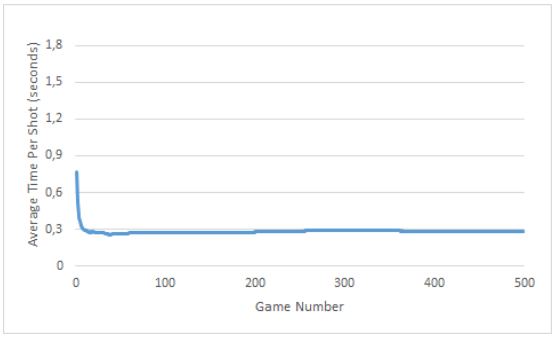


Fig. 10: Average time per shot generating shot types one by one. Stabilizes at 0.29 seconds without noise

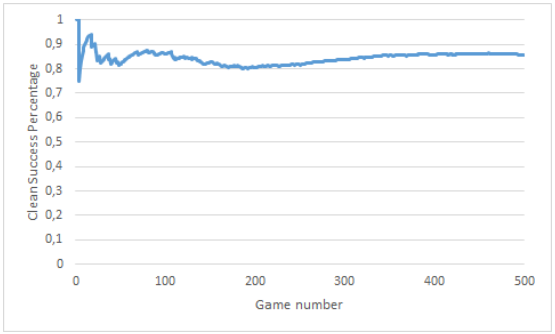


Fig. 11: Table clean success percentage generating shot types one by one. Stabilizes at 86% without noise

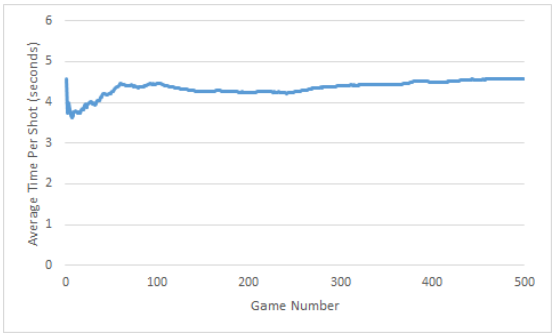


Fig. 12: Average time per shot generating all shot types. Stabilizes at 4.57 seconds without noises

VI. 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 good results of the tests without noise we consider that this goal was a success since games normally do not have noise perturbations. By ordering the shots by difficulty, which have into account the distance of the balls, we

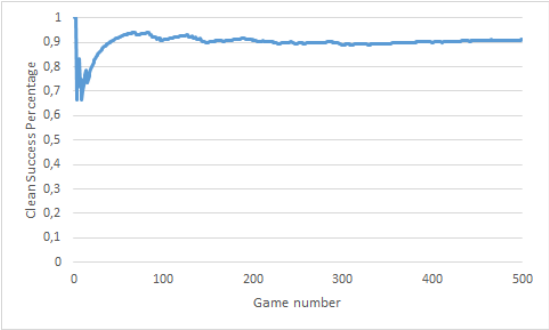


Fig. 13: Table clean success percentage generating all shot types. Stabilizes at 91% without noise

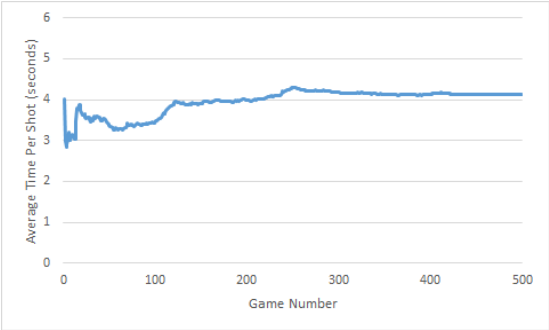


Fig. 14: Average time per shot with a depth of 3. Stabilizes at 4.12 seconds without noise

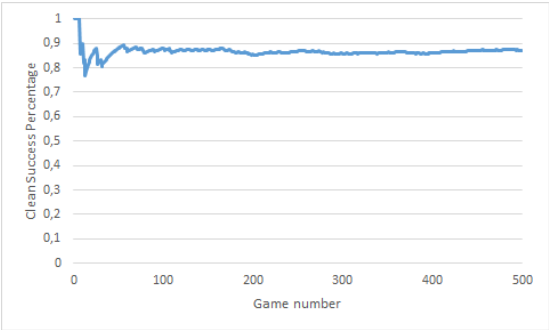


Fig. 15: Table clean success percentage with a depth of 3. Stabilizes at 87% without noise

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 can be done as future work. One of the main problems of the algorithm was relaying 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.

Another 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 up to 4 times. 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. It would also be interesting to study how to simulate several types of skill or even automatically adapt the skill to its opponent, since MiniPool was developed to be used as an artificial opponent in a 8-Ball video game.

References

[1] Christopher Archibald. Skill And Billiards. (August), 2011.

[2] Christopher Archibald, Alon Altman, Michael Greenspan, and Yoav Shoham. Computational Pool : Game Theory Pragmatics. pages 33–41, 2010.

[3] Christopher Archibald, Alon Altman, and Yoav Shoham. Analysis of a Winning Computational Billiards Player. pages 1377–1382, 2009.

[4] Christopher Archibald and Yoav Shoham. Modeling Billiards Games. 2009.

[5] Jens-uwe Bahr. A computer player for billiards based on artificial intelligence techniques. (September), 2012.

[6] Billiards Congress of America. *Billiards: The Official Rules and Records Book*. 2014.

[7] Nicolas Bureau. Sensibilité des coups au billard. 2:9–26, 2012.

[8] K Chakrabarti. Pure Strategy Markov Equilibrium in Stochastic Games. (June 2011), 2013.

[9] S.C. Chua, E.K. Wong, and V.C. Koo. Performance evaluation of fuzzy-based decision system for pool. *Applied Soft Computing*, 7(1):411–424, January 2007.

[10] Shing Chyi Chua, Eng Kiong Wong, and Voon Chet Koo. Intelligent Pool Decision System Using Zero-Order Sugeno Fuzzy System. *Journal of Intelligent and Robotic Systems*, 44(2):161–186, January 2006.

[11] Jean-pierre Dussault. Optimization of a Billiard Player – Position Play. pages 263–272, 2006.

[12] Jean-pierre Dussault. Optimization of a Billiard Player – Tactical Play. pages 256–270, 2007.

[13] J. Koehler. *The Science of Pocket Billiards*. Sportology Publications, 1995.

[14] Jean-François Landry and Jean-Pierre Dussault. AI Optimization of a Billiard Player. *Journal of Intelligent and Robotic Systems*, 50(4):399–417, October 2007.

[15] Jean-François Landry, Jean-Pierre Dussault, and Philippe Mahey. A robust controller for a two-layered approach applied to the game of billiards. *Entertainment Computing*, 3(3):59–70, August 2012.

[16] Jean-Francois Landry, Jean-Pierre Dussault, and Philippe Mahey. 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):325–336, December 2013.

[17] Will Leckie and Michael Greenspan. An event-based pool physics simulator. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4250 LNCS(1995):247–262, 2006.

[18] J B MacQueen. Some Methods for classification and Analysis of Multivariate Observations. *5th Berkeley Symposium on Mathematical Statistics and Probability 1967*, 1(233):281–297, 1967.

[19] Michael Smith. PickPocket: An Artificial Intelligence For Computer Billiards. 2006.

[20] Michael Smith. PickPocket: A computer billiards shark. *Artificial Intelligence*, 171(16-17):1069–1091, November 2007.