TÉCNICO LISBOA

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

David Gonçalo Branco da Silva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Rui Filipe Fernandes Prada

Examination Committee

Chairperson: Prof. Paolo Romano

Supervisor: Prof. Rui Filipe Fernandes Prada

Member of the Committee: João Miguel de Sousa de Assis Dias

November 2015

Acknowledgments

I would like to thank my supervisor, Prof. Rui Prada for all the help, valuable feedback and motivation that he gave to me during all the development of this work and for being always pointing me to the right direction when I was starting to feel lost.

To my family and friends in general, thank you for understanding me in all the times that I had to stay at home working, instead of hanging out with you. A special thank you Vânia Mendonça, Soraia Meneses Alarcão, Fabio Alves, Élvio Abreu, Ruben Rebelo and João Moreira for being my friends during all these years and for being always available for reading the dissertation and suggest improvements. To Luis Sampaio, a great mathematician that helped me with some formulas and calculations. I am really glad for finding all these true amazing friends during my academic life. I hope we can still see each other after this. And a big thank you to my girlfriend Liliana Santos who helped me a lot emotionally and supported me in everything. I really love you, thank you very very much.

To my father Joaquim Silva and my sister Simone Silva thank you for always supporting me in all the moments of more anxiety and panic during all these years. To my grandmothers Beatriz Branco and Conceição that are always asking me when will I finish my dissertation, TODAY IS THE DAY! (I hope so). And a very special thank you to my mother Carla Branco who unfortunately died while I was developing this work and could not see me graduate. Thank you for being the most wonderful person, mother and friend I could ever had.

Without the help of all of you I do not think I could finish this. Thank you all!

Resumo

A importância da inteligência artificial em jogos tem vindo a crescer ao longo dos anos devido ao contínua aumento do seu realismo e a necessidade de manter a sua imersividade enquanto se joga. Jogos como o Bola 8 oferecem muitos desafios interessantes para ambas as comunidades de IA e otimização devido ao seu características contínuas e estocásticas do domínio. Para ter sucesso, um jogador deve ser capaz de planear a melhor sequência de tacadas e executar a tacada escolhida com exatidão e precisão, para não perder o turno.

Já existem vários bons jogadores artificiais desenvolvidos, porém tendem a demorar mais do que 30 segundos para selecionar e executar uma tacada. Sob circunstâncias normais, um jogador iria desistir de jogar o jogo se tivesse que esperar tanto tempo para jogar.

Neste documento proponho uma solução para um jogador artificial 8-Ball que responde em tempo real usando um algoritmo de procura baseado em Monte-Carlo e Expectimax com técnicas de raytracing.

Keywords: Jogador Artificial, Brilhar, Bola 8, Jogo Estocástico, Jogo de Vídeo, Tempo-Real

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 o plan the best sequence of shots and execute a shot with accuracy and precision, so he doesn't lose the turn.

There are already several good artificial players developed, however they tend to take more than 30 seconds to select and executed 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 a 8-Ball artificial player using a Monte-Carlo Expectimax hybrid search algorithm with raytracing techniques.

Keywords: Artificial Player, Billiards, 8-Ball, Stochastic Game, Video Game, Real-time

Contents

List of Figures

1

Introduction

1.1 Motivation

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 [2]. 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 3, 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

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

[2] `http://www.theesa.com/`

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.

Having all this into account makes the development of an artificial player for an 8-Ball game, in my point of view, a problem worthy of attention and investigation.

1.2 Goals

The main focus of this work will be to develop artificial player capable of compete with the best 8-Ball players while having a good balance between skill and resources used. The artificial player developed here will be able to plan a sequence of shots that clears the table under noise perturbations on the shot parameters and executed a shot in less than 6 seconds.

To achieve this main goal, we need to focus on three different sub-goals:

- The search algorithm needs to explore the minimum number of states as possible without penalizing the look-ahead.
- The shot generator needs to generate only the most relevant shots for a given table state.
- The evaluation function needs to penalize shots that are more vulnerable to noise perturbations.

1.3 Document Structure

The rest of the document is organized as follows:

- The *Background* section describes the basic rules of 8-ball, the shot parameters and the most common shot types used by players. Additionally, there is also a brief description of the 8-ball video game where the artificial player will be tested.
- The *Related Work* section describes some of the work already done in billiards artificial players, with special emphasis on PickPocket [1, 4], CueCard [5–7], PoolMaster [3, 8–11] and JPool [2]. The topics under discussion are search algorithms, shot generation and aiming techniques, the evaluation functions and shot optimization methods. Every topic has a brief discussion and, when it is possible, I also compare the work done on the artificial players mentioned above.
- The *Implementation* section has a figure with the general architecture of the MiniPool player and an explanation of how the main features were implemented.

- The *Results* section has the experimental results and analysis of MiniPool.
- Finally, the *Conclusion and Future Work* section sums up the important aspects and decisions taken and some future work.

2

Background

2.1 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 2.1) toward 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 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 [12].

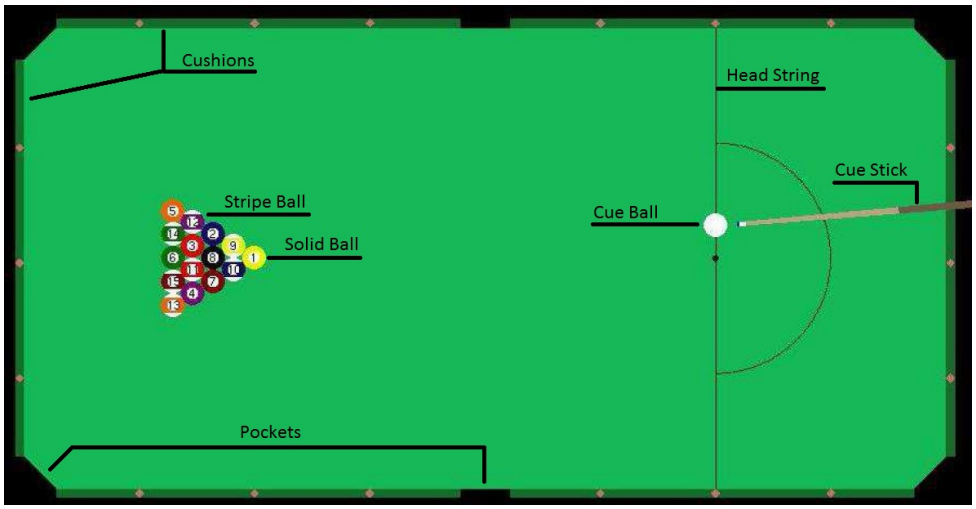


Figure 2.1: Initial pool table layout

2.2 Shot Parameters

In every Billiards variant a shot is defined by five continuous parameters (illustrated in figure 2.2):

- 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.

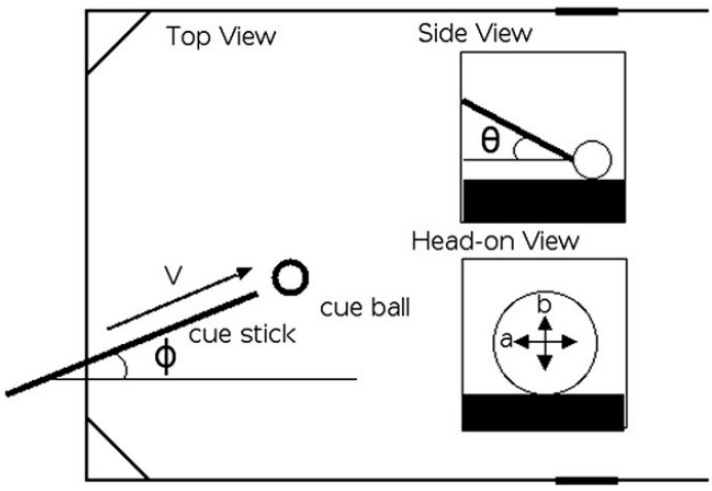


Figure 2.2: Parameters that define a 8-ball shot (image from [1])

2.3 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 [2] (illustrated in figure 2.3):

- **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.

2.4 8-ball Videogame

I developed a simple 8-Ball 2D video game to test the artificial player. The game uses the *FastFiz* [1] physics engine (the same engine used in [1–11]), which is based on the simulator used in the past *Pool Computer Olympiads* [2], PoolFiz [13]. 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.

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

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

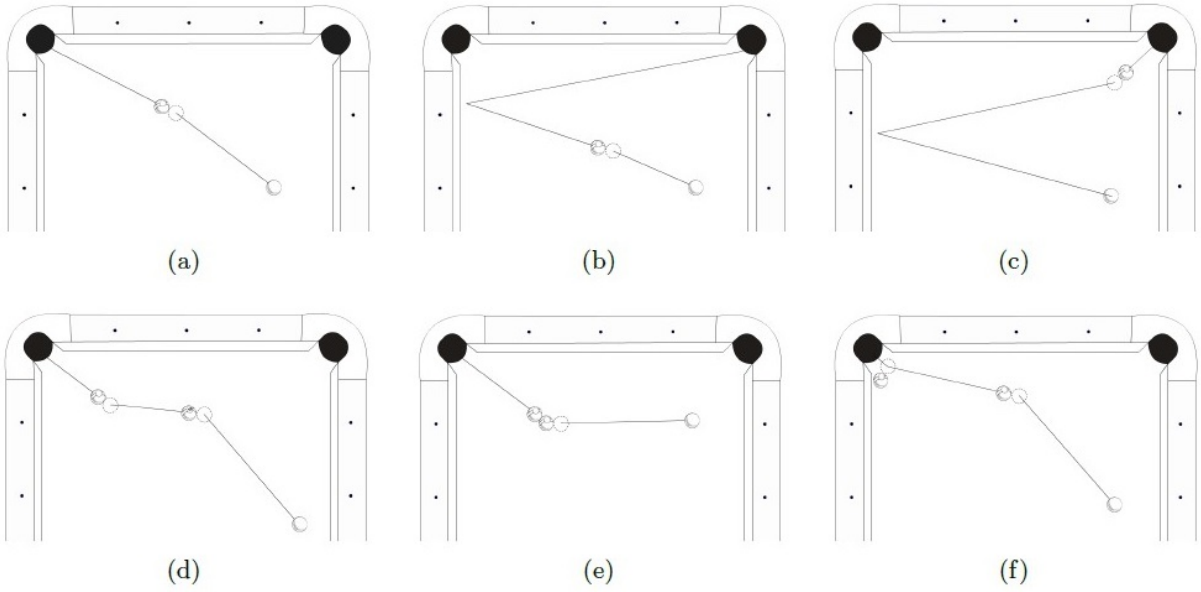


Figure 2.3: Shot types: (a) direct shot, (b) bank shot, (c) kick shot, (d) combination shot, (e) pulk shot, (f) kiss shot (image from [2])

3

Related Work

Currently, there are already several artificial 8-Ball players developed; PickPocket [1, 4], CueCard [5–7] and PoolMaster [3, 8–11], who participated in the *Pool Computer Olympiads*, and JPool [2].

To better understand the differences between them and the most important modules of any search based program I organized all the information in several topics. First, the game model for 8-ball, which might help in some design decisions; the search algorithms that can be used in this domain to find the best sequence of shots for a given table state; the complete algorithms used by each one of artificial players mentioned above to select the shot for execution; aiming techniques to help discover the angle ϕ of a shot during the shot generation process; methods to discover the best place to reposition the cue ball, so that the player is less likely to lose the turn and functions to evaluate each state of the search algorithm;

3.1 Billiards Game Model

Christopher Archibald *et al.* said in [7, 14] that billiards belongs to a class of games called *turn-taking stochastic games* which are games with a continuous action and state space, a turn-taking feature and a reward accumulation upon game termination. They proved that billiards have a *pure stationary Markov perfect equilibrium*. A strategy is called *pure stationary* if a player will take the same action for each state, independent of how the state was reached and regardless the game's current stage. In the case of billiards this same strategy has also a *Markov perfect equilibrium*, which means that in every turn

there is no incentive to a player to change his own strategy even if they know the opponent's strategy.

The value function *v* for a state *s* in a *Markov perfect equilibrium* for the current player is:

$$v(s) = \max_a \left[\int_S v(s')p(s'|s,a) \right], s \in S \tag{3.1}$$

Where p is the probability of reaching state s' when executing action a on the state s. If we find the sequence of shots that leads to the maximum value of the equation 3.1 until the last ball, we achieve, at the end, the optimal solution.

With this knowledge they shown too that the best response to a pure stationary strategy will also be a pure stationary strategy. Which means that only by exploring pure stationary strategies we are not eliminating the optimal response to the opponent.

An explanation of this model for stochastic games in general can be found in [15].

3.1.1 Discussion

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 turn 2 our 20 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 the opponent, which is minimizing the function 3.1.

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, in fact is not necessary at all to achieve the optimal solution. If a strategy is optimal it will remain optimal regardless the opponent actions and strategy.

Having a model for a game can help in some design decision, however, the proposed model remains intractable due to the action space being continuous. To compute the value of a state, $v(s)$, using the equation 3.1, we would need to try all the possible actions a, which are infinite (the same goes to the safe shots, where we want to minimize the equation 3.1 in the opponents perspective). Thus, any approach still needs to perform an intelligent search space partitioning to overcome this problem.

3.2 Search Algorithms

When playing a generic game, players have to find a sequence of actions that puts them in a better situation. One way of resolving this problem is using search algorithms. An action generator provides to the search algorithm possible actions for a given game state. The search algorithm will then select the sequence of actions, with a given evaluation criteria (see section 3.5), that puts the player in a better position to win the game.

For the particular case of 8-ball and considering stochastic environments, PickPocket [1,4] and JPool [2] suggested *Expectimax* and *Monte Carlo* as possible search algorithms.

- **Expectimax:** An adaptation of *Minimax* for stochastic domains. *Min* nodes for the opponents actions, *max* nodes for our deterministic actions and *chance* nodes for the actions with a non-deterministic outcome. Every *chance* node have a child for each possible outcome with their probability of occurrence. The value of a *chance* node is the sum of these probabilities. For the case of 8-ball a *chance* node value would be a sum over an infinite number of outcomes with a minuscule probability of occurring. Thus, in [4] was discussed an approach estimating this value during shot generation with other criteria. The algorithm is shown in listing 3.1, where $ShotGenerator()$ returns shots with an estimated probability of success and $Simulate()$ tries the shot without external perturbations.

```
1  float Expectimax(TableState state, int depth){
2          if(depth == 0) return Fitness(state);
3          List<Shot> shots = ShotGenerator(state);
4          int bestScore = -1;
5          TableState nextState;
6          foreach(Shot shot in shots){
7                  nextState = Simulate(shot, state);
8                  if(!Success(shot)) continue;
9                  score = shots.probabilityEstimate * Expectimax(nextState, depth - 1);
10                 if(score > bestScore) bestScore = score;
11         }
12         return bestScore;
13 }
```

Listing 3.1: Expectimax search algorithm. (code from [4])

- **Monte Carlo:** Search tree with nodes representing an action. Each node is sampled *num_samples* times to obtain its value. This value will be estimated by the average value of the samples. In the case of 8-ball, a node would be a shot configuration that would be simulated *num_samples* times to estimate the result of that shot. The algorithm is shown in listing 3.2, where $ShotGenerator()$ returns possible shots, $PerturbShot()$ perturbs the shot parameters and $Simulate()$ tries the shot.

```
1  float MonteCarlo(TableState state, int depth, int num_samples){
2          if(depth == 0) return Fitness(state);
3          List<Shot> shots = ShotGenerator(state);
4          int bestScore = -1;
5          TableState nextState;
6          Shot tempShot;
7          foreach(Shot shot in shots){
8                  int sum = 0;
9                  for(j = 1 to num_samples){
10                         tempShot = PerturbShot(shots);
11                         nextState = Simulate(tempShot, state);
12                         if(!ShotSuccess(tempShot)) continue;
13                         sum += MonteCarlo(nextState, depth - 1);
14                 }
15                 score = sum / num_samples;
16                 if(score > bestScore) bestScore = score;
17         }
18         return bestScore;
19 }
```

Listing 3.2: Monte-Carlo search algorithm (code from [4])

PoolMaster [3, 8–11] uses a different search algorithm. They first cluster balls using the *K-Means* algorithm [16] and then, using a heuristic based search, they select the sequence of clusters which

minimizes the distance travel between them. The sequence of shots is them selected based on this sequence of clusters.

3.2.1 Discussion

The Expectimax approach, as already said, cannot be applied directly to 8-ball since the chance node value would be a sum over an infinite number of outcomes, each with a minuscule probability of occurring. However, clustering states like CueCard does (see section 3.3.2) to reduce the state space might give us more significant values to make this approach reliable. The alternative version in the listing 3.2 have some drawbacks too, since differentiating shots by their probability of success and without any other type of information will not distinguish shots that reposition the cue ball better than the others.

Monte-Carlo was used by all the artificial players that will be explained in section 3.3 with the exception of PoolMaster. Using a direct approach might be a better option for 8-ball, since the average of the value of the samples can give us a relative good estimation of the value of a shot under a stochastic environment. Shots with more good resulting states will score higher then the others and *vice versa*. However, this is not a perfect solution because it relies on the number of simulations of shots (samples) for good estimations, which has a huge performance cost.

PoolMaster by choosing the sequence of shots based on the distance traveled will only select a riskier shot if needed. In their results they shown that this kind of look-ahead may give better results than a tree based search approach, however they don't clearly explain how their search is made.

3.3 Algorithms Overview

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 every artificial player have different approaches to overcome the challenges of 8-Ball, I will explain them individually.

3.3.1 PickPocket

As explained in [1, 4], PickPocket generates shots one type at a time in order of increase difficulty, direct, bank, kick and combination shots, respectively. When the generation and evaluation (see section 3.5) of every shot and resulting states for a specific type is complete, if at least one shot was found, the shot generator returns the shot with higher value. If not, PickPocket falls back and explores the next shot type until a shot is found. The maximum search depth is set to 2-ply.

The generation of each shot type and the ball-in-hand situation are done as follows:

- **Direct shots:** For every legal ball for every pocket a direct shot is generated. ϕ is chosen such that the object ball is aimed to the pocket (see section 3.4). V is retrieved from a precomputed

table of minimum velocities to get the object ball into the pocket. θ is set to a minimum physically possible value, while a and b are set to zero. Additional shots are generated perturbing V, a and b to find better resting spots for the cue ball. The minimum velocities table returns a velocity for the closet combination of cut angle, distance between the cue ball and the target ball and the distance from the target ball and the pocket.

- **Bank shots:** They are generated in a similar way but for every cushion too. Since it is physically impossible to bank off a rail into a pocket along that same rail, one rail is skipped for side pockets, and two rails are skipped for corner pockets.
- **Kick shots:** They are generated using a precomputed rail rebound table. If at least one candidate entry in the table is found, a kick shot is generated; otherwise, it is not.
- **Combination shots:** For every pair of object balls between the cue ball and the target pocket, for every pocket, a combination shot is generated. ϕ is calculated by backtracking from the object ball to be pocketed and positioning ghosts balls where the balls must be hit.
- **Break shot:** The break shot parameters are selected by sampling shot variations 200 times and selecting the one which returns better results.
- **Safe shots:** Shots perturbing V and ϕ are generated and evaluated in the opponent's perspective. The shot with the lowest value is selected for execution. Safe shots are generated only when no direct shot was found.
- **Ball-in-hand:** The table is discretized in a grid and every cell is assigned with the value of the best shot as the ball was there. Then an hill-climbing search is performed in several random selected cells to find the local maximum. The best cells are further searched to find the best spot.

3.3.2 CueCard

Similar to PickPocket shot generation but does not prioritize the shot types. The steps to select the best shot are the following [5–7]:

1. Shots are generated with minimal parameters to pocket every legal ball to every pocket. Direct shots with other legal balls on the way are not discard to eventually disperse clusters of balls.
2. Variants perturbing V, a, b and θ of each feasible shot from step 1 are generated.
3. Each resulting state of the feasible shots from steps 1 and 2 are evaluated (see section 3.5).
4. The samples (generated due to the evaluation) of the top 2 shots from step 3 for each (ϕ, V) pair variant are clustered into groups of similar table states (position of the balls and the value of the resulting states) and a representative state is chosen for each cluster of each shot.
5. The top 20 shots from steps 1 and 2 are selected for a second level of search starting from their representative resulting states calculated in step 4.

- In the early game the second level of search is essentially repeating steps 1 to 4 with smaller perturbations.
- In the late game, CueCard calculates the probability of winning the game by conducting a search until the end of the game for every possible shot without having into account external perturbations on them. When the end of the game is reached, the probability of winning for that state is set to 1 and this value is then backed up the search tree to give value to the states where it is still CueCard's turn.

6. A reevaluation is done and the best shot from step 5 is selected. If the value of the resulting state is under a certain threshold or no feasible shot was found, shots with maximum V and random ϕ are generated and the best one (value of the resulting state for CueCard minus the value of the resulting state for the opponent) is selected.

The break shot parameters were precomputed (via an extensive offline search over possible cue ball locations and shot parameters) so that it keeps the turn 92% of the time.

For the ball-in-hand situation, CueCard generates shots placing the cue ball where it would be the ghost ball to pocket each legal ball and perturbing ϕ. If this is not possible, it uses the same method as PickPocket.

3.3.3 PoolMaster

The focus of PoolMaster [3,8–11] lies on position play. Instead of randomizing the shot parameters to get the best results, they analyze the table for good repositions zones for the cue ball and use this information for the minimization of the objective function to determine the shot parameters. They introduced the idea of a two layered approach, where a planner selects the best sequence of shots in a (target ball, target pocket) combination and the controller optimize the parameters for the wanted result.

The steps to select the best shot are the following:

1. Direct, bank, kick and combination shots are generated in a (target ball, target pocket) combination. Bank and kick shots are generated using the aid of a mirror table (illustrated in figure 3.1), being able to treat them like a direct shot.

2. The shots from step 1 are rated based on their difficulty using the method in [17] which uses the distance between the leftmost and the rightmost vectors as a metric (see section 3.4 for more details). The higher the distance, the easier the shot.

3. The next valid balls are clustered and the clusters are sorted such that the distance traveled between them is minimized. In late game instead of clustering balls they make a search up to the end of the game to calculate the probability of success of a shot as CueCard does.

4. The parameters for best shots from 2 are then selected minimizing the objective function using the the BOBYQA [18] local optimization from the NLOPT [1] library with the balls sorted from 3.

[1] `http://ab-initio.mit.edu/nlopt/`

The objective function used is:

$$\min_{u} f(u) = \max(Posval(b_{cue}(u)) - Val_{min}, 0) + \rho_1 \parallel b_{cue}(u) - t \parallel + \rho_2(\triangle\theta_A * \overline{d}) + \rho_3 * \eta \quad (3.2)$$

Where u is the shot parameters, b_{cue} is the cue ball, $Posval$ returns the reposition value (lower value meaning a better position), Val_{min} is the minimum position value for a zone, ρ is a penalization factor to ensure that the object ball enters a pocket and the cue ball reaches a good reposition zone, t is the target position, $\triangle\theta_A$ is the shot difficulty, $\overline{d}$ is the norm of the difference between the cue ball velocity after impact and the center of the zone of the target position and finally η is the norm of the gradient of $f(u)$.

5. The shots from 4 are evaluated (see section 3.5) and the best one is selected for execution. If the value of the resulting state is under a certain threshold or no feasible shot was found, safe shots are generated (random shots aiming to legal balls and evaluated in the opponent's perspective).

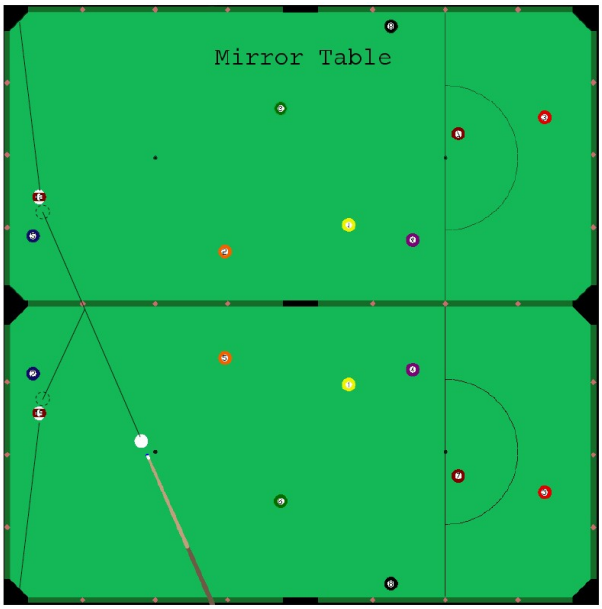


Figure 3.1: PoolMasters' mirror table for a kick shot (image from [3])

3.3.4 JPool

Takes a different approach, given that it models a shot as a series of steps like a tree (illustrated in figure 3.2), being not limited to predefined shot types like the other agents. The types of steps are cue strike, pocketing a ball, rail collision (or rail collision), ball striking, ball kissing and force post-collision trajectories calculations for both balls (*ball both*).

The steps to select the best shot are the following [2]:

1. For each pocket the step *Pocket* is added to the *undoneShots* list.

2. Target points to every leaf of every entry in the *undoneShots* list are calculated (for example center of a pocket, ghost ball position).

3. Checks for every ball not involved if the target points from step 2 are reachable directly, per rail collision or by kiss, and generates new entries in the *undoneShots* list with the respective working events added.

4. If the cue ball was used in step 3 the event *CueStrike* is added to the respective shot and saved in the *doneShots* list with maximum V. Returns to step 2 until there are 3 balls involved in every shot or no more steps are found.

5. New shot entries are generated perturbing a, b and θ of every shot from the *doneShots* list.

6. Position zones are analyzed for every resulting state of every shot in the *doneShots* list and their parameters are modified so that the cue ball can reach them. These position zones are generated using polygon meshes with 60 cm ray and 90 degrees opening angle on the side of the ball opposed to the pocket (where it would be a good place to pocket that ball) and triangles obstructed are removed from the polygon (illustrated in figure 3.3). To compute the shot parameters to reach these zones, JPool 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.

7. The result working shots from step 6 are evaluated and the shot with the resulting state with higher value is selected for execution. If there is only one ball on the table, the method will be the probability of success for pocketing that ball. If no feasible shot is found, JPool generates random shots aiming to the legal balls and selects the one with the lowest value in the opponent's perspective.

The break shot parameters were precomputed.

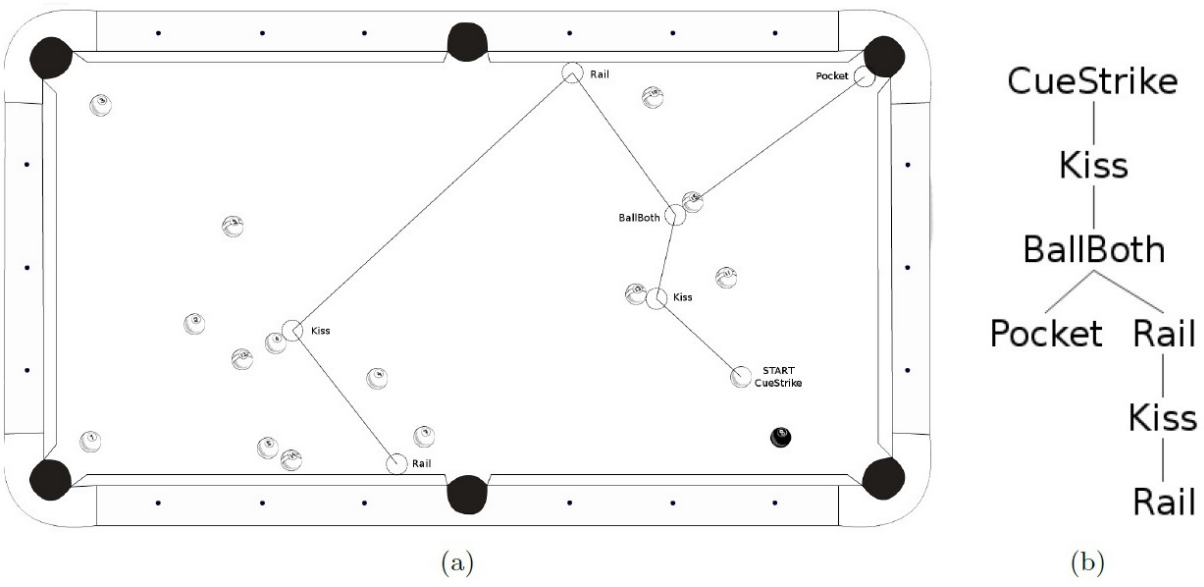


Figure 3.2: (a) steps to execute a kiss shot (b) step tree used by JPool (image from [2])

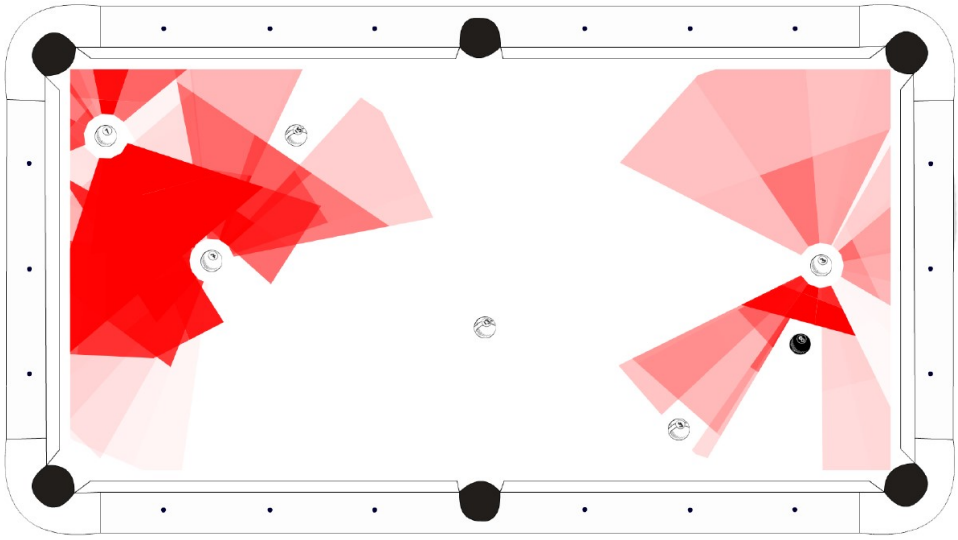


Figure 3.3: JPools' best regions for the cue ball positioning aiming for the solids (image from [2])

3.3.5 Discussion

CueCard is an upgrade to PickPocket and what made it win was those modifications. PickPocket assumes that a direct shot is always better and this is not true every time. CueCard by clustering similar states can explore a lot more shot variations. In the *ball in hand* situation, CueCard starts by placing the cue ball near the legal balls and this gives him more reliable shots than the discretization grid made by PickPocket.

CueCard and PickPocket made several tests to understand the importance of depth on the decision making. These results shown that under a low stochastic nature a 2-ply depth gives better resu lts than using 1-ply. This is easy to understand since the predictions made at 2-ply where based in a more feasible 1-ply depth output. However, under a higher stochastic nature the results are inconclusive. CueCard made other tests trying to understand if using the time that would have been spent on the 2-ply depth exploring more shots at 1-ply depth would be better, but the results were also inconclusive.

After this, PoolMaster came up with a planner for 8-ball that gave him better results than PickPocket and CueCard. This happened because PoolMaster have a much more robust shot optimization which gives him more ”stochastic proof” predictions. PickPocket and CueCard use precomputed tables and some discretizations to improved their performance and rely on sampling to optimize the shot. These simplifications might have been the problem for their results on the tests.

The mirror table used by PoolMaster simplifies a lot the shot generation, however since it assumes that the angle of incidence equals the angle of reflection, requires a more robust shot parameters optimizer like the one they have, because the shot parameters will need to be fixed in order to reach the target reposition after the collision with the rail. An approach relying on sampling may not fix this.

The tree approach of JPool to generate shots is a very interesting algorithm since it does not search for a specific shot type. This makes the code more flexible. However the polygon method to find reposition zones is too restrictive. It will force the shot generator to find parameters that reach these zones

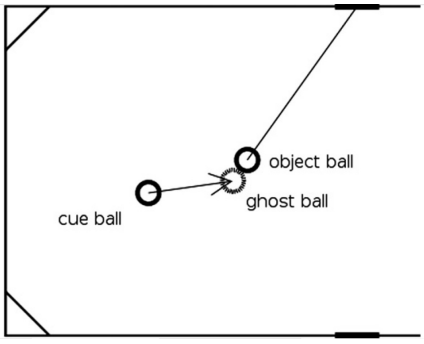
which might generate shots too complex or not explore shots that do not reach them. Shots that do not reach these zones are not necessarily impossible. The sampling approach used by the others is not the best one too because it relays on the number of samples to get the best results. PoolMaster by having a more restrictive sampling with the optimization function generates only a shot at the end but this shot will be less riskier for the wanted result.

PickPocket algorithm takes an average of 60 seconds to selected and execute a shot, JPool takes around 44 and PoolMaster takes 35 seconds. CueCard does not mention this information, but since it was made to support parallel executions it might be faster. The 8-ball game were I will work on will have a 30 seconds time limit to execute a shot, thus I will have to use and find ways to optimize as much as possible the decision process without losing to much precision and skill.

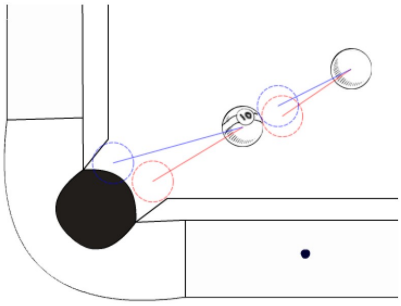
3.4 Aiming

As already stated, when generating a shot we need to find five parameters. One of these parameters is ϕ, the aiming direction. PickPocket [1, 4] and CueCard [5, 7] use a conceptual aid called *ghost ball* (illustrated in figure 3.4a). If we extend a line from a target position (for example, the center of a pocket) to the center of the object ball, the position adjacent to this ball will be the ghost ball position. And if the cue ball is aiming 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. The angle used to aim for this ghost ball will be our ϕ.

PoolMaster [3, 8–11] and JPool [2] 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 3.4b). 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. JPool took this further by having in account also rail reflections when calculating these margins. The returned ϕ will be the angle in the middle of the two ghost balls generated, since is the one with equal margins of error.



(a) Ghost ball concept (image from [1])



(b) Leftmost and rightmost concept (image from [2])

Figure 3.4: Aiming concepts

3.4.1 Discussion

Pool players frequently use this concept called *ghost ball* because it is a simple and easy way to aim for a ball and get the expected results. However, the improvement made by PoolMaster and JPool to this concept is very interesting because it can predict impossible or very difficult shots sooner when updating the margins, which in the continuous stochastic domain of 8-ball is a great help to reduce the action space.

3.5 Evaluation Function

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

JPool [2] uses a Monte-Carlo search base algorithm with a sample size of 400. The leaves are evaluated using a sum of several heuristics based on the advises given by Jack Koehler in [19]. These heuristics are:

- How good is the current position of the cue ball;
- How good is the current position of the cue ball for the opponent (the same but in the opponent's perspective);
- The number of balls already pocketed;
- Average ball value of the current player's balls (for example the difficulty of each ball);
- Average ball value of the enemy player's balls (the same but for the opponent's type of balls);
- Unpocketed moved balls values (if the moved balls are in a better position now or not)

PickPocket [1, 4] and CueCard [5, 7] 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 a sum of the best 3 shots. The value of a shot is its probability of success. The probability of success of a shot is retrieved from the nearest entry on a precomputed table having into account the cut angle and the distance between the balls and the pocket.

PoolMaster [3, 8–11] uses a sample of 15 to calculate the difficulty of a shot using the following function:

$$f(u) = \max[\max(Posval(b_{cue}(u)) - Val_{min}, 0)\forall\zeta] * \alpha_0 * dist(\zeta, U) \tag{3.3}$$

Where u is the shot parameters, b_{cue} is the cue ball, $Posval$ returns the reposition value (lower value meaning a better position), Val_{min} is the minimum position value for a zone, α_0 is the repositioning success probability, U is the range of successful shot parameters and finally ζ corresponds to the possible noise perturbations.

Shing Chua *et al.* shown in [20, 21] 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 between the two balls. They have a 24 rules table combining these variables with the respective overall difficulty assign. 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.

3.5.1 Discussion

What probably made PoolMaster lose the two Olympiads was the heuristic for shot selection they used back then. The selection of the best shot was made based on the quality of the shot repositioning, resulting in shots very risky in some situations. PickPocket accounted for this using a probability of success as heuristic, which in a implicit way penalizes shots that may have unintended results. With this new evaluation function presented by PoolMaster, this problem was solved and, with the help of the two layered approach for the shot generation and selection, made PoolMaster the state of the art in 8-Ball artificial players.

Fuzzy logic is a very interesting method, because it simplifies the rating process in problems with vagueness, uncertainty, or where there is no exact model for the problem. However, the method proposed by Shing Chua *et al.* lack the capacity of comparing the difficulty of different shot types, because the prioritization assumes that a direct shot is always the best option for a specific situation, which might not be true.

The best number of samples used to estimate the value of a node depends on how sensible the rest of the program is with these values, however the higher the number of samples the better the estimation. The tests made by PickPocket shown that having more than 15 samples did not change significantly the results. For CueCard this number was between 30 and 150 samples, something bellow or above this was worse.

4

Implementation

In this section I will explain the implementation of MiniPool, the artificial player developed for the propose of this document. As mentioned in section 3 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. At the end of the section there is a figure with the general architecture of MiniPool.

4.1 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 3.

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 [1, 4], 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. In practice real players only attempt more complex shots when they lack easier options. The order of shots types used, Direct, Combination, Bank/Kick and Safe Shots, is different from the one used by PickPocket due to the complexity of the algorithms used in the shot generator (see section 4.2).

At this point we have reduce a lot the state space, but we still have a solution highly dependent on sampling. PoolMaster reduced this dependency by rating shots based on their difficulty and only exploring the less difficult. The rating function they use only have into account the size of the angle at which the target ball still enter the pocket. In [20, 21] was used a function to evaluates shots that have into account the distance between the balls and the pocket and the cut angle.

$$\triangle = \frac{d_{co} d_{op}}{cos^2 \alpha} \tag{4.1}$$

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. CueCard reduced the number of states explored after sampling by clustering similar states, but what if we could find a way to reduce even more the number of states without reducing to much the quality of the look ahead? PickPocket, CueCard and JPool use this search on the samples to find the next shots and calculate and average score of the quality of the next shots based on the cue ball final position. 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. To

check if the cue ball is in a good reposition for the next ball a ray with the thickness of the ball is cast in the direction of the next ball (similar do what is done in section 4.2 on the shot generator) and if the way is cleared of obstacles, since the target ball was already retrieved as being part of a shot, the shot is possible.

The general search algorithm is the following:

```
1  ShotPlan SearchAlgorithm(ShotPlan previousShot, int depth){
2          if(depth == 0)  return null;
3          List<ShotPlan> shots = shotGenerator.GenerateShots(previousShot.state);
4          SortShotsByDifficulty(shots);
5          ShotPlan bestShot = null;
6          float bestScore = MIN_SUCCESS_PROBABILITY;
7          foreach(ShotPlan shot in shots){
8                  if(shot.difficulty > MAX_DIFFICULTY) break;
9                  ShotPlan nextShot = SearchAlgorithm(shot, depth - 1);
10                 previousShot.nextShot = nextShot;
11                 Rate(previousShot, SAMPLES, bestScore);
12                 if(previousShot.score > bestScore ){
13                         bestShot = shot;
14                         bestScore  = previousShot.score;
15                         if(besScore >= MAX_SUCCESS_PROBABILITY) return bestShot;
16                 }
17         }
18         return bestShot;
19 }
```

Listing 4.1: MiniPool search algorithm

A $ShotPlan$ is a structure with information about the characteristics of the shot, such as: the balls involved, the number of rails involved, the state where the shot will be executed, the distance traveled, the cut angle of the cue ball, the shot parameters, the target ball, the target pocket, the noiseless resulting state and a pointer for the next shot plan reposition. The $SortShotsByDifficulty()$ method rates every shot with the 4.1 function and sorts them in crescent order, being the less difficult in the beginning. The $GenerateShots()$ method of the Shot Generator is explained in section 4.2 and the evaluation function, $Rate()$, in explained in section 4.3.

When the return value of this algorithm is null a new search is preformed with the Shot Generator set to search next shot type, otherwise the optimization function is called for the returned shot.

With all this modifications to the base algorithm we have a solution for a real time response problem that does not reduce to much the skill of the player.

4.2 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 backtrack algorithm is the following:

```
List<ShotPlan> Backtrack(GameState state, ShotPlan shot){
        if(shot.nBalls == MAX_BALLS) return null;
        List<ShotPlan> shots = new List<ShotPlan>();
        foreach(Ball ball in state.GetBalls()){
                if(!IsValidBall(state, shot, ball)) continue;
                float angle,
                if(rayTracer.CastRay(shot.ghostBall, ball.getPosition(), angle)){
                        if(angle > MAX_CUT_ANGLE) continue;
                        bool needRail = NeedRailColision(ball) ;
                        if(needRail && shot.nRails == MAX_RAILS ) continue;
                        if(needRail) shot.nRails++;
                        shot.nBalls++;
                        if(ball.name == CUE_BALL){
                                if(shot.nBallsInvolved == 0 ) continue;
                                calculateParameters(shot);
                                shots.push(shot);
                        }
                        else {
                                shot.nBallsInvolved++;
                                shot.ghostBall = ball.getPosition() + (ball.getPosition() - shot.ghostBall).normalize() * BALL_RADIUS * 2;
                                shots.push(BackTrack(state, shot));
                        }
                }
        }
        return shots;
}
```

Listing 4.2: MiniPool backtrack algorithm

The $IsValidBall()$ method checks if the ball is one that the current player needs to hit first or wants to pocket and checks is the ball was already used in the current shot plan. The $CastRay()$ method from the raytracer returns true if the the ray reaches the target position without obstacles and the angle of collision. The $NeedRailColision()$ checks if a rail collision is needed to reach the target ball based on the ball position (better explanation later is this section). The backtrack algorithm is called for every pocket with the ghost ball starting in the center of it.

One pruning that was used in the algorithm was the maximum cut angle. This cut was added to remove shots that barley touch the balls, because they will not produce relevant results.

For the case of the bank and kick shots it was used the PoolMaster [3, 8–11] 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 number of balls involved and the number of rail collisions allowed we have a general algorithm for almost every shot type without having to explicit look for it. Althouth it was not explored in this work, by using a raytracing approach we can use raytracing acceleration techniques such as kd-tree [22], which can improve up to four times the performance of the raytracer. Since a big part of the MiniPool algorithm depends on raytracing, using such techniques can improve even more the overall performance. The raytracer developed for the purpose of this work uses a ray with

thickness to have into account the ball dimension when checking for obstacles in the way.

At this point we have a schema for every possible shot with what will happen on the table. We now have to calculate the shot parameters that successfully do the expected schema. PickPocket, CueCard and JPool randomize the shots parameters and generate a new shot for each combination that successfully pocket the ball in the expected pocket. PickPocket and CueCard reduce the number of impossible combinations by initially giving to every shot a minimum velocity retrieved from a precomputed table of minimum velocities. PoolMaster calculates the parameters for every shot with the minimization of an objective function with a local optimization algorithm (see section 3.3.3 for more details). The approach of PoolMaster needs more time than the others to retrieved relevant results and for that reason to quickly find the minimum velocity required to pocket a ball it was precomputed a minimum velocities table like PickPocket and CueCard. However the table was generalized to every type of shot using the cut angle (the angle that the cue ball does with the target ball), the distance between the cue ball and the first ball involved , the distance of the target ball to the pocket, the number of rails and balls involved and the distance traveled by the rest of the balls. The velocities were calculated generating situations of every combinations of inputs and simulating shots incrementing the velocity by 0.1*m/s* until the ball is pocketed in a noiseless environment . The purpose of using this approach is only to reduce the number of velocities variants that do not pocket the target ball, so it does not need to be very precise.

The calculation of all 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 the precomputed table of minimum velocities.

There are an infinite number of variants to these parameters 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. The variations on ϕ will help fix the direction of the ball on bank and kick shots and will also allow shots to pocket balls other than aiming the target ball to the center of the pocket, since the target ball will be hit in a different position. However the quality of this approach highly depends on the number of variants and the domain at which the variations are made. For that purpose the maximum and minimum ϕ are calculated with basic trigonometry so that the target ball is always hit.

Since the number of variants have a huge impact on the branch factor of the algorithm a study of the must relevant parameters was made in section 5.

4.3 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.

```
1  List<ShotPlan> Rate(ShotPlan shot, int samples, float bestScore){
2          int score = 0;
3          for(int i = 0; i < samples; i++){
4                  ShotResult result = shot.Simulate();
5                  if(IsSuccess(result){
6                          float angle;
7                          if(rayTracer.CastRay(shot.state.getCueBallPosition(), shot.nextShot.ghostBall, angle)
8                                  if(angle <= MAX_CUT_ANGLE) score++;
9                  }
10                 if ((bestScore * samples) - score > samples - i) break;
11         }
12         return score / samples;
13 }
```

Listing 4.3: MiniPool evaluation function

The $IsSuccess()$ method checks if the players keeps the turn. A pruning was also added to the evaluation function to reduce the number of simulations done. If the number of samples remaining is not enough to reach the best score, the calculation stops.

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 [19] 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.

4.4 8-Ball Specific Situations

There are 3 situations in 8-Ball that cannot be handled by the regular shot detection algorithm: break shots, ball-in-hand and safe shots. This section presents the techniques used by MiniPool to handle each one of them.

4.4.1 Break Shot

The break shot is the name of the shot with the purpose of break the initial clusters of balls.discretizes

PickPocket generates a number of shots variants and selects the one which have the higher probability of pocketing a ball. CueCard and JPool use a precomputed break shot which keeps the turn 92% and 98% of the time. The motive for CueCard to not use a shot with a higher probability of keeping the turn is because the creators of CueCard think that 92% is a good balance between success and how

spread the balls are. If the balls are too disperse and the player loses the turn leaves the opponent in a good position.

The break shot has an unpredictable outcome even in a noiseless environment (in *FastFiz* the gap between each ball is randomize). Since the results shown by CueCard using a precomputed shot are very good and since this solution is the one that uses less resources, the break shot used by MiniPool is the same as CueCard.

4.4.2 Ball-in-hand

The ball-in-hand situation happens when a player makes a foul. This situation gives to the other player the possibility to put the cue ball anywhere on the table.

Every artificial player discussed in this document discretizes the table in a grid and performs hill-climbing in several cells. The differences are in some optimizations. CueCard before discretizing the table tries to positioning the cue ball in the position of the ghost ball for a specific ball and selects the direct shot with higher probability of success. JPool discretizes only the zones covered by the reposition polygons. Finally PoolMaster uses grid-based heuristic to find the local maxima.

MiniPool starts by positioning the cue ball at the ghost ball position for each ball, found using the raytracer techniques explain in section 4.2, and performs a typical search on each possible shot to select the shot with higher probability of success. If this is not possible there is no possible shot to be done on the table, so random positions on the table are generated and the first with a clear way to a ball is selected for the generation of a safe shot.

4.4.3 Safe Shot

Whenever no valid shot is found for a given table state the player is forced to make a last resort shot, the safe shot. The purpose of a safe shot is touch a ball we are sinking, so that we do not make a fault, and reposition the cue in a difficult place for the other player to keep the turn.

MiniPool handles this situation in a similar same way as JPool by generating random shots aimed in the direction of each legal ball. For each shot it is perform a search for direct shots to the opponents balls from the cue ball noiseless final position and the average score of each shot using the formula 4.1 is given to the original shot. The shot with the highest average difficulty is selected for execution. MiniPool only performs this search when no shot was found, since the model discussed in section 3.1 proves that there is no gain doing it in other situation.

4.5 Architecture

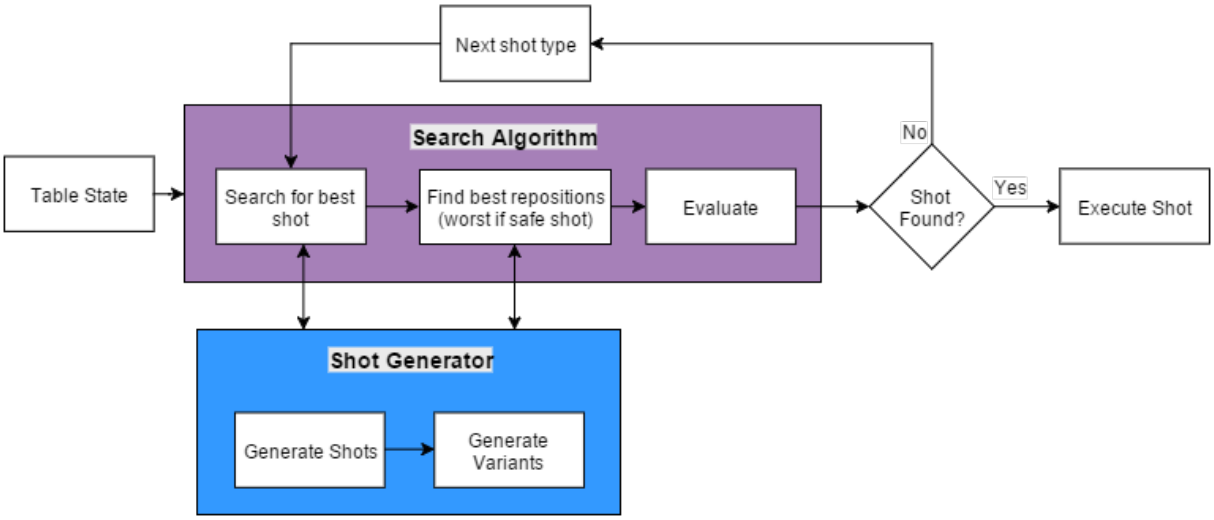


Figure 4.1: General architecture of MiniPool

5

Results and Analysis

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.

At the end of this section there will be a comparison with the other artificial players discussed in this document.

5.1 Testing Environment

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%.

5.2 Results with noise

The graphics 5.1 and 5.2 show the impact of using a bigger sample size. These test were made using a sample size of 25 and 100 respectively, a depth of 2 and 125 variants of V, ϕ and b.

The graphics 5.3, 5.4 and 5.5 show the impact that variations on each shot parameter have on the quality of the player. These tests where made using a sample size of 25, a depth of 2 and 125 variants. In 5.3 was generated variants of V and ϕ only, in 5.4 V, ϕ and a and in figure 5.5 variants of V, ϕ and θ.

The graphics in 5.6 show the impact of using a higher number of variants. This test was made using a sample size of 25, a depth of 2 and 343 variants of V, ϕ and b.

5.2.1 Analysis

In general the clean success probability 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 5.1c 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 5.2, by taking 7 more seconds per shot we can increase by 10% the clean success probability using a bigger sample size.

By generating shots types one by one it was possible to reduce by 6 seconds the average time per shot penalizing only 5% of the clean success probability. It is a very huge gain compering with the loss of skill.

The number of variants generated per shot have a huge impact on the branch factor of the search algorithm. For that reason it was made a study to see the parameters that give better results. ϕ and V were used in all the combinations because they are essential to fix the parameters for the mirroring approach. Varying 3 parameters with a limit of 125 variants per shot will only generates 5 variants of each parameter (cubic root of 125) . Having all the parameters varying it would generate 3125 variants per shot. Using less variants per parameters generates variants with huge gaps between, which is something not wanted. For these reasons it were only tested combinations up to 3 parameters. As can be seen in 5.1, 5.3, 5.4 and 5.5 b shown results 7% better than the others. The reason for this might be because θ needs a and b varying, and a needs small variants of ϕ. b by changing the results of the shot only along the direction of the movement does not rely as much as the others on other parameters.

In 5.6 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.

5.3 Results without noise

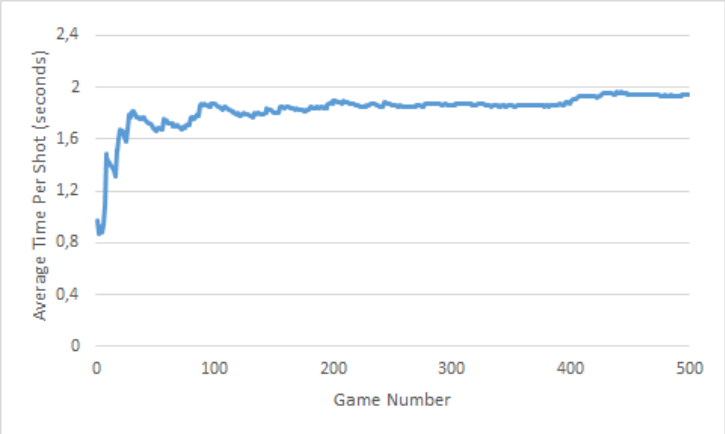
The graphics 5.7 and 5.8 show the impact of generating or not all the shots type at once. These test were made using a sample size of 1, a depth of 2 and 125 variants of V, ϕ and b.

The graphics in 5.9 show the impact of using a higher depth. This test was made using a sample size of 1, a depth of 3 and 125 variants of V, ϕ and b.

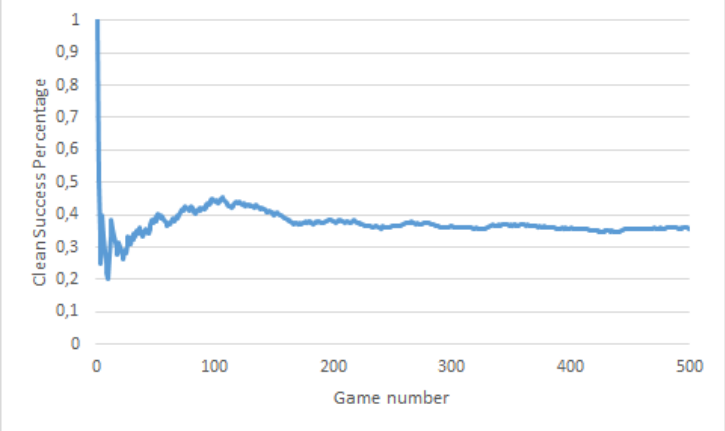
5.3.1 Analysis

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 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.

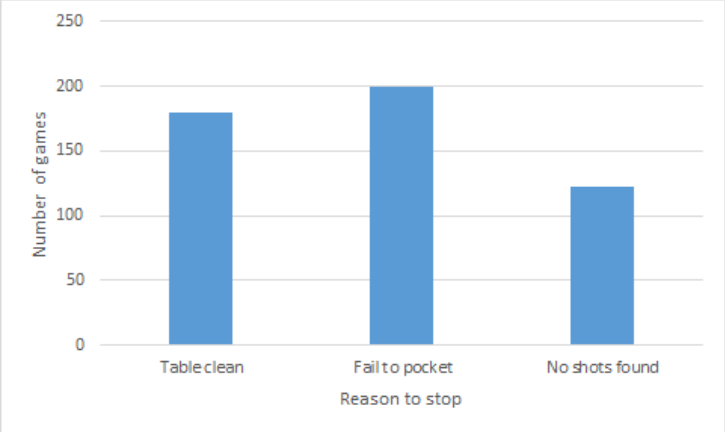
Since there is no noise in these tests the reason for the algorithm to not be able to clean the plan is probably a planning problem. To check this it was made a test using a depth of 3, 5.9, however the results did not improved at all. My guess for this, and looking at the results in 5.8 where the results 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.



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

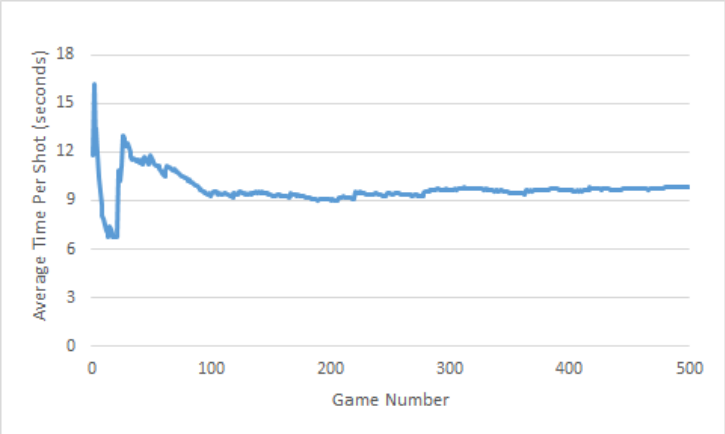


(b) Table clean success percentage, Stabilizes at 35.8%

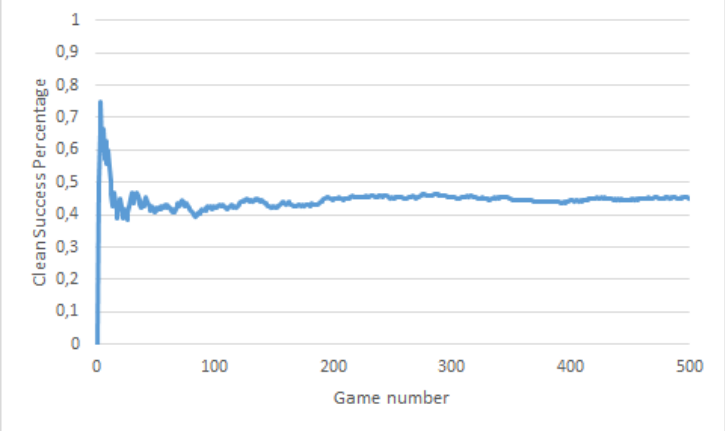


(c) Motive for the algorithm to stop the iteration

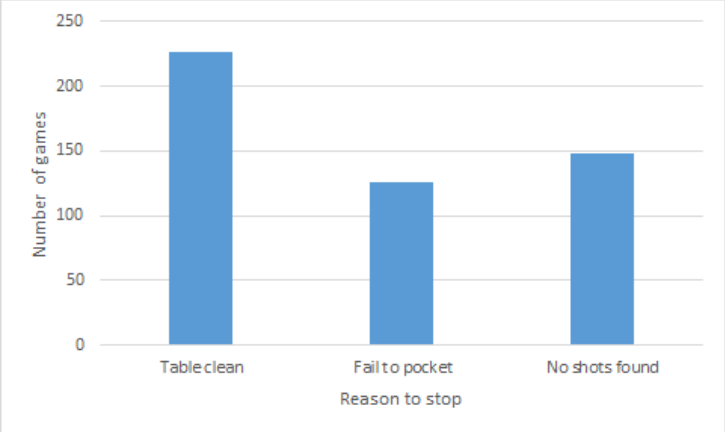
Figure 5.1: Sample size of 25 with noise at 0.5x



(a) Average time per shot. Stabilizes at 9.79 seconds



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



(c) Motive for the algorithm to stop the iteration

Figure 5.2: Sample size of 100 with noise at 0.5x

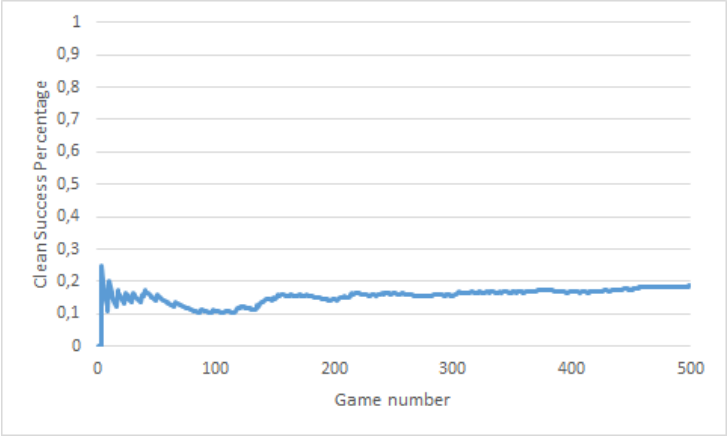


Figure 5.3: Table clean success percentage for variations on ϕ and V. Stabilizes at 18.6% with noise at 0.5x

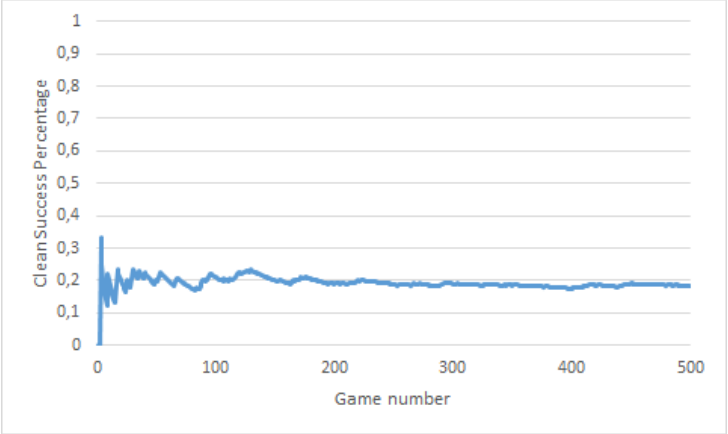


Figure 5.4: Table clean success percentage for variations on ϕ , V and a. Stabilizes at 18.4% with noise at 0.5x

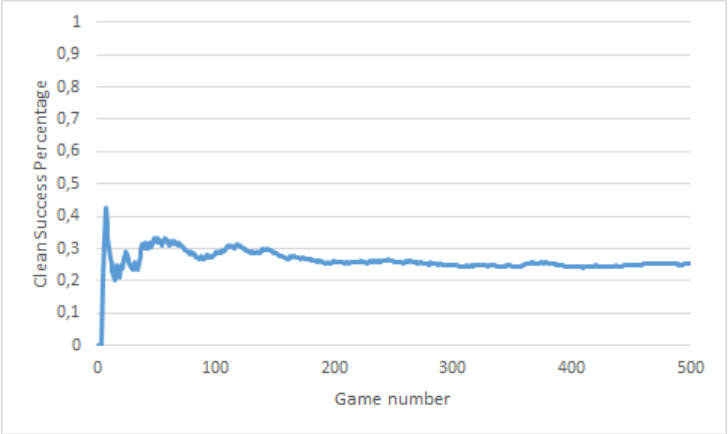
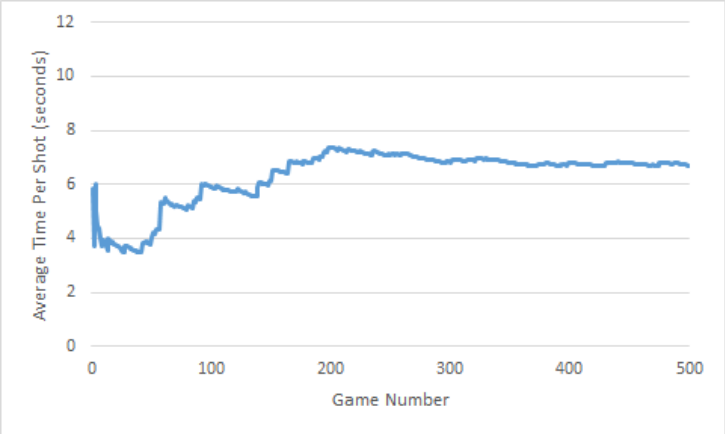
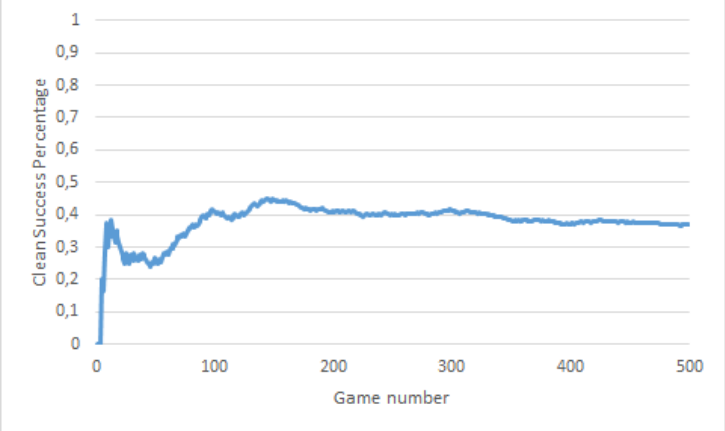


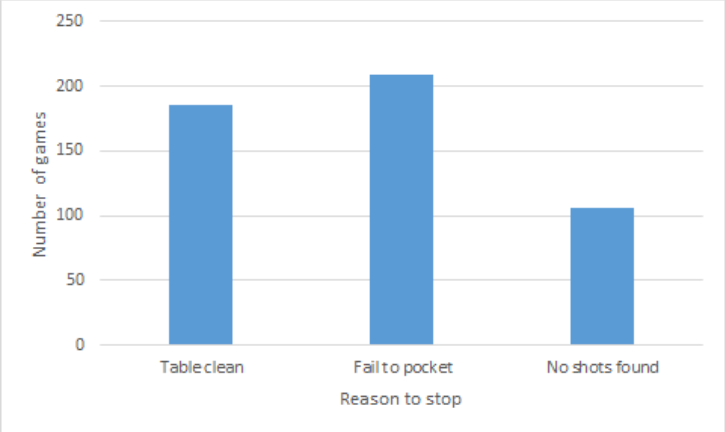
Figure 5.5: Table clean success percentage for variations on ϕ , V and θ. Stabilizes at 25.4% withnoise at 0.5x



(a) Average time per shot. Stabilizes at 6.69 seconds

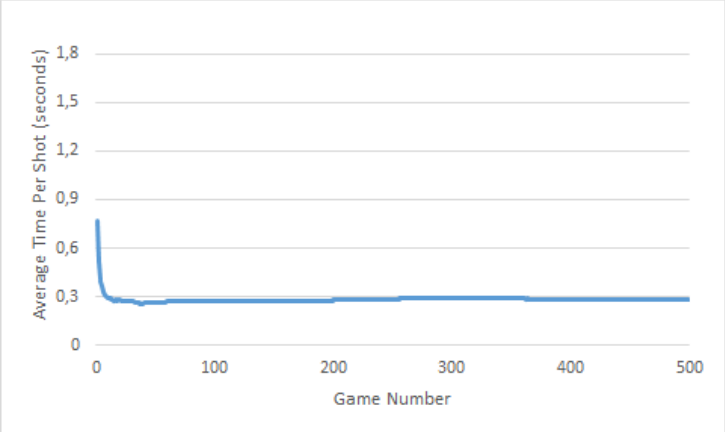


(b) Table clean success percentage. Stabilizes at 37%

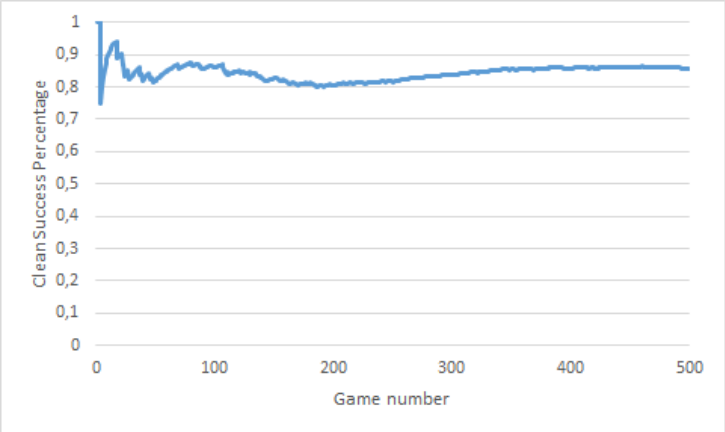


(c) Motive for the algorithm to stop the iteration

Figure 5.6: Using 343 variations per shot with noise at 0.5x

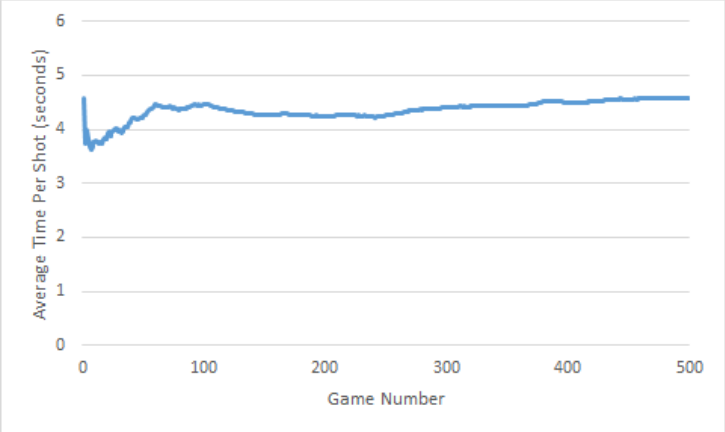


(a) Average time per shot. Stabilizes at 0.29 seconds

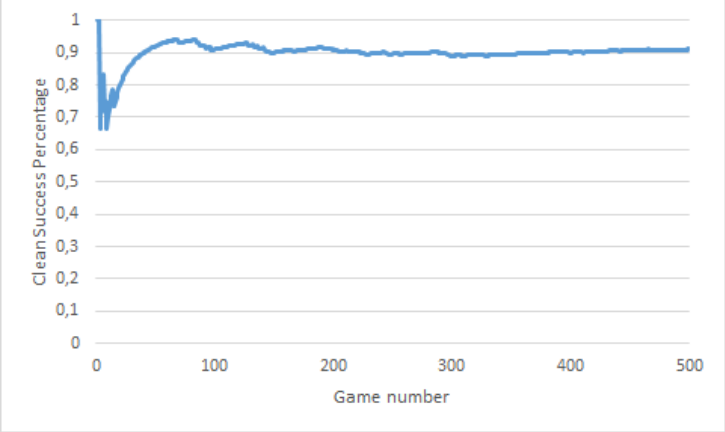


(b) Table clean success percentage. Stabilizes at 86%

Figure 5.7: Shot types one by one without noise

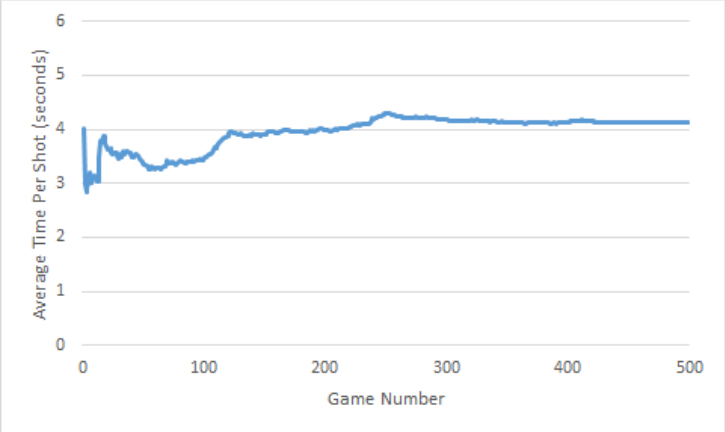


(a) Average time per shot. Stabilizes at 4.57 seconds

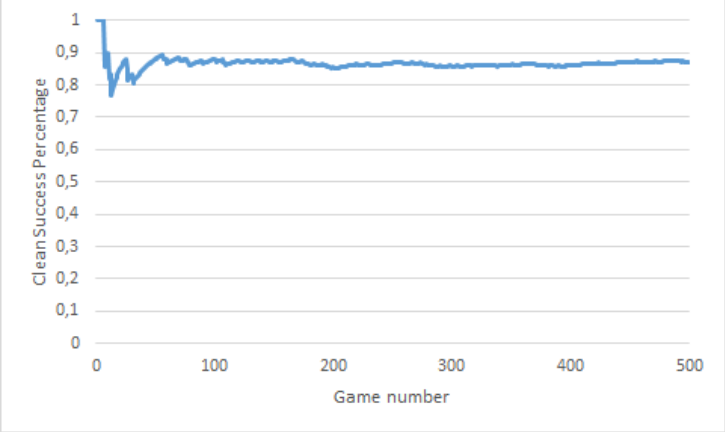


(b) Table clean success percentage. Stabilizes at 91%

Figure 5.8: All shot types at once without noise



(a) Average time per shot. Stabilizes at 4.12 seconds



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

Figure 5.9: Depth of 3 without noise

6

Conclusions and Future Work

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 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.

Bibliography

[1] M. Smith, “PickPocket: An Artificial Intelligence For Computer Billiards,” 2006.

[2] J.-u. Bahr, “A computer player for billiards based on artificial intelligence techniques,” no. September, 2012.

[3] J.-p. Dussault, “Optimization of a Billiard Player – Tactical Play,” pp. 256–270, 2007.

[4] M. Smith, “PickPocket: A computer billiards shark,” *Artificial Intelligence*, vol. 171, pp. 1069–1091, Nov. 2007.

[5] C. Archibald, A. Altman, and Y. Shoham, “Analysis of a Winning Computational Billiards Player,” pp. 1377–1382, 2009.

[6] C. Archibald, A. Altman, M. Greenspan, and Y. Shoham, “Computational Pool : Game Theory Pragmatics,” pp. 33–41, 2010.

[7] C. Archibald, “Skill And Billiards,” no. August, 2011.

[8] J.-p. Dussault, “Optimization of a Billiard Player – Position Play,” pp. 263–272, 2006.

[9] J.-F. Landry and J.-P. Dussault, “AI Optimization of a Billiard Player,” *Journal of Intelligent and Robotic Systems*, vol. 50, pp. 399–417, Oct. 2007.

[10] J.-F. Landry, J.-P. Dussault, and P. Mahey, “A robust controller for a two-layered approach applied to the game of billiards,” *Entertainment Computing*, vol. 3, pp. 59–70, Aug. 2012.

[11] J.-F. Landry, J.-P. Dussault, and P. 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*, vol. 5, pp. 325–336, Dec. 2013.

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

[13] W. Leckie and M. Greenspan, “An event-based pool physics simulator,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4250 LNCS, no. 1995, pp. 247–262, 2006.

[14] C. Archibald and Y. Shoham, “Modeling Billiards Games,” 2009.

[15] K. Chakrabarti, “Pure Strategy Markov Equilibrium in Stochastic Games,” no. June 2011, 2013.

[16] J. B. MacQueen, “Some Methods for classification and Analysis of Multivariate Observations,” *5th Berkeley Symposium on Mathematical Statistics and Probability 1967*, vol. 1, no. 233, pp. 281–297, 1967.

[17] N. Bureau, “Sensibilité des coups au billard,” vol. 2, pp. 9–26, 2012.

[18] M. J. D. Powell, “The BOBYQA algorithm for bound constrained optimization without derivatives,” 2009.

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

[20] S. C. Chua, E. K. Wong, and V. C. Koo, “Intelligent Pool Decision System Using Zero-Order Sugeno Fuzzy System,” *Journal of Intelligent and Robotic Systems*, vol. 44, pp. 161–186, Jan. 2006.

[21] S. Chua, E. Wong, and V. Koo, “Performance evaluation of fuzzy-based decision system for pool,” *Applied Soft Computing*, vol. 7, pp. 411–424, Jan. 2007.

[22] T. Foley and J. Sugerman, “KD-tree Acceleration Structures for a GPU Raytracer,” *ACM*, pp. 15–22, 2005.