A heuristic-based planner and improved controller for a two-layered approach for the game of billiards

Jean-François Landry, Jean-Pierre Dussault, Philippe Mahey

A heuristic-based planner and improved controller for a two-layered approach for the game of billiards

Jean-François Landry, Jean-Pierre Dussault & Philippe Mahey

August 4, 2013

Abstract

In [16] a two-layered approach was proposed to compute a winning strategy for the game of billiards. AI tools as well as robust optimization routines for noisy environments were combined to plan the sequence of shots. We complete here the modelling by introducing significant developments for the high-level planner which guides the precise optimal controller to generate a plan given any random initial state. We will first resume the general model for this particular class of problems and then propose several domain-specific heuristics to guide our search and render the problem tractable. Several improvements to the optimal robust controller including refinements in the objective function will also be presented in order to improve single-shot optimization. Results are presented demonstrating the full potential of the methods proposed making it the state of the art in regards to the game of billiards.

Keywords: Pool, Billiards, Snooker, Game, Simulation, Planning, Strategy, Optimization, Artificial Intelligence

1 Introduction

The game of billiards is a game of skill, tactics and ingenuity. It has been played all over the world for countless years and is even officially recognized as an Olympic sport by the International Olympic Committee. It has also been the focus of many recent projects in the research community. The engineering challenge to create a robot precise enough to compete and even win against the best professional players is a very strong motivation for the field of robotics. Many projects have been developed from the mid 1990s to today, (see [12], [19], [9], [8], [3], [25] [2], [22]) with approaches ranging from a robotic arm set-up on a gantry over a table to a standalone robot moving freely around. Another interesting aspect which was investigated was the usage of augmented reality tools to help guide and teach new players how to play the game or better select shots on the table. Efforts have been made in [17] and [13] with interesting results. Some work has even been done in video analysis in [14] to extract information from tournaments played on television to get game summaries.

Another aspect, the focus of this paper, concerns the intelligence required to plan and select the best shot sequences on the table, given a complete physical simulator of the game. A lot of work has also been done in this regard. The usage of fuzzy logic has been documented in [10] as a way to evaluate shot difficulty on a given table state. Bayesian network models have also been proposed in [23] in the same context. Monte-Carlo sampling approaches were used in [27] to not only evaluate shot difficulty but generate possible plans. Finally, least-square optimization approaches were described in [15] as an efficient approach for position-play.

When playing the game of billiards as amateurs, we are usually confronted by one key problem: cue-ball control. Although we know which shot we are aiming for, executing it is a completely different matter. It is a matter of skill, technical precision, and mechanical physics. We can usually estimate the trajectory of the cue-ball given an average cue-strike but it takes years of practice to perfectly master the shot execution, and to be able to gauge which shot we can successfully complete. Thus for an amateur, the best shot is almost always the easiest one. This makes the game very interesting since as we progress, we are able to execute harder shots leaving us with a new array of possible shots which opens up a new level to the game and adds in the complexity of planning. This is also the area where we see the pro's come in. Having better cue-control and position play they can envision more than one shot ahead, to be able to select the shot which may be harder to execute but yields a better chance of finishing the table.

The approach discussed in this paper was inspired from watching real-world players select and execute their shots. Up to now, state of the art approaches ([27], [4]) usually planned for 2-3 shots in advance by discretizing the search space such as to try various angle and speeds at which to hit the cue-ball, simulating the shot, and then continuing at the next ply from the position reached by the cue-ball. What was proposed in [16], however, was to divide the problem in two layers, such that a robust controller may be guided by a high-level planner, taking advantage of possible knowledge which can be extracted from the domain. Although the controller described in that work proved to be very robust, the downside was a necessity for many simulator calls for every single evaluation of the objective function. Also, no specific planner was proposed, thus leaving the full potential of the two-layer approach to be yet demonstrated.

With this paper we bring forward several improvements to the robust controller by modeling the objective function differently using several penalties to guide the minimization process more effectively, generating better solutions with less calls to the simulator. We also bring forth the elements necessary to complete the two-layered approach which was suggested, by expanding on the description of high-level planner that will establish a preferred shot sequence to maximize chances of finishing the game with success. Results are presented, showing an improvement of around 13% success-rate (at highest noise-level) with modifications to the controller alone in the case of randomly dispersed tables, while a 7% improvement is observed for the planning heuristics, yielding a total of over 20% of improvements over pass results and positioning this approach as state of the art in regards to the game of 8-ball.

The rest of this paper is structured as follows. We first present an overview with a short background on the game of billiards and the two-layered scheme proposed previously and then introduce the heuristic strategy for the high-level planner. In section 4, we refine the objective function of the robust controller to generate better solutions with less calls to the simulator. Finally we will discuss the results of several tests aimed at demonstrating the benefits of the various elements presented in this paper.

2 Background

Although there exists a significant number of billiards games, we focus in this paper on the game of 8-ball. This game variant has been the focus of a lot of research, and allows us to establish some comparisons with state of the art approaches. To provide a general idea of the difficulties present in the game, we describe below the general rules used in the game of 8-ball. We then provide a short description of the physics simulator engine used to generate the tests in the final section.

2.1 The game of 8-ball

The game of 8-ball is generally played table of 1:2 dimensions, with four pockets in the corners, and two on the sides. There are fifteen numbered balls and one cue-ball, which will be hit with the cue-stick to impact the other balls and send them in the pockets. At the beginning of the game, the balls are placed in a triangle at one end of the table, and the player which has been selected to start will break the triangle using the cue-ball from the other end of the table. If he manages to pocket one of the balls, he can continue playing, and decide to aim for the low balls (1-7) or the high balls (9-15). Once a player starts playing for the low or high balls, he must finish them all and then keep the 8^{th} ball last. If he pockets it by accident (scratch), he loses the game. For each shots after the break, he must call the target ball and the pocket he aims for, if he misses, he loses his turn. He can also call a safety shot, which means he wilfully gives the next turn to the opponent after executing his shot. This type of shot is usually done when a player has no shot possible and prefers to leave the opponent in a bad position. The first player to finish all of his balls and pocket the 8^{th} ball wins the game.

2.2 Physics-based simulator

The attributes on which we focus in this paper are related the computational aspects of billiards. As such, we do not address the difficulties related to the creation of a realistic and accurate physics-based simulator, which is a great challenge in its own. Thus, we present here the framework we use to test the approaches proposed in this work, the FastFiz simulator, which is based on the Poolfiz ([18]) simulator used in past computational pool tournaments. Fastfiz is an event-based physical simulator which simulates the physics of a billiards table

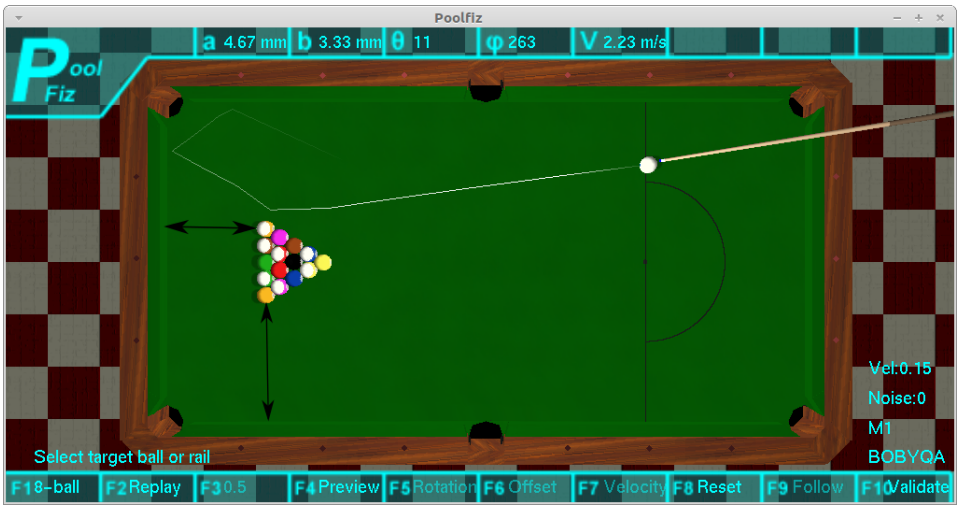

Figure 1: Break shot at the beginning of the game. At least two balls need to hit the rails.

by analytically solving a set of equations describing the movement of balls on the table after an initial velocity and spin is imparted to the cue-ball by the cue-stick.

For our work, we suppose that it acts as a black-box. We provide the simulator with 5 parameters corresponding to a shot, and in return it gives us the outcome of this shot after simulation. The simulator can also provide a detailed list of the events that occurred during the shot simulation.

The five control variables corresponding to a shot defined as a vector $u = \{a\ ,\ b\ ,\ \theta\ ,\ \phi\ ,\ v\}$ are the following:

- a: Horizontal offset from the ball's center.
- b: Vertical offset from the ball's center.
- θ: Angle of the cue-stick in relation to the plan of the table.
- ϕ: Orientation of the cue-stick.
- v: Initial speed given to the cue-ball.

The Fastfiz simulator is deterministic, which means the same shot will always result in the same table state. In reality however, imperfection in the ball collisions, the rail's elasticity and other factors may influence the outcome of a shot. To simulate such effects, in past computational tournaments, different noise-levels were applied to the shot parameters before the shots were simulated. A secondary goal was also to try and create tournaments in which players' planning abilities would tested as much as their technical abilities. The shot parameters we perturbed by the addition of randomly generated noise using a Gaussian distribution with the following standard deviations:

ϕ=0.125 deg, θ=0.1 deg, v=0.075 m/s, a=0.5mm, b= 0.5mm.

Thus in the results section, a noise-level of 1x corresponds to this noise-level, and $\frac{1}{2}$ x corresponds to half this level.

2.3 A two-layered approach

A general model to represent the game of billiards was first proposed in [6], demonstrating the presence of a Nash Equilibrium in the game. This is very interesting and serves as a motivation for pure stationary strategies (i.e. where an optimal strategy will remain optimal regardless of the opponent's actions). The model proposed, however, is clearly intractable due to the state and action space being continuous. Thus, to be successful, any approach needs to perform an intelligent partitioning of the search space to render the problem tractable.

It is also clear that this model ([6]) suggests we consider the possibility of executing safe shots by accounting for whichever shot our opponent would do, to then regain control of the game in a favourable position. This scenario, where we go as far as predicting the result of our opponent's shot, is highly unlikely and almost impossible to predict. A safety shot is usually done only if no shot sequence leads to victory with an acceptable success rate. In such a case, it will be better to leave the turn to our opponent, but leave him in the worst possible position on the table, to then regain control of the game later on. Very good players will often have several exchanges of safe shots when the table state has many clusters. Taking these factors into consideration, we will for this work only consider safety shots as a last resort when it is clear no shot will be possible for our next move.

It is important to remember that the mix of continuous and discrete aspects, in a domain subject to stochastic perturbations brings forward many complex elements to the model, and demands a very careful analysis. We proposed a novel approach in [16] under the form of a two-layered model, using a planner and controller to establish the optimal shot sequence. It is the first time such an approach has been proposed for the game of billiards, other approaches studied until now usually relied on a discretization of the action space to generate solutions, and while these are effective in general, we believe that a more specific approach to the problem as we propose in this work will give better results in the long run.

3 High-level planner

Planning with continuous states and actions under stochastic environment is a challenge; we propose in this section a set of domain-specific heuristic rules to help narrow the search space and find highly efficient solutions. These heuristics are part of the planner depicted in Figure 2, which fits into the two-layered approach previously proposed (see [16] for more details).

The planner receives a given table state and performs a preliminary analysis to determine which shots are possible from the cue ball's position. This shot list will be composed of direct, indirect and combination shots, and is basically a list of pocket-ball combinations, with the queue angle to execute the shot as a starting point for the optimization. The shot list will then be analysed and a value representing the shot difficulty will then be associated to each.

Next we will proceed to the planning of the sequence of shots to execute, depending on the number of balls remaining on the table. If only 3 balls are remaining, we will use a backward computation approach to compute the shots, otherwise we will use one of the

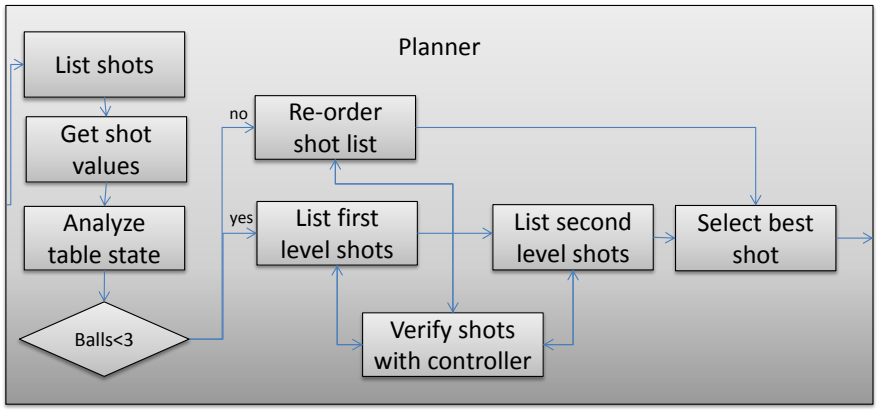

Figure 2: Detailed illustration of the planner.

heuristics described below to specify a preferred shot-order. The planner will then access the controller and optimize the selected shots. Based on the results, further evaluations will be done if necessary, otherwise the best shot will be selected and executed.

Since we are playing the game of billiards, we can extract several key components from the given table state to help decide which shot sequence should lead to a better outcome. We describe several heuristics in the following subsections which help the planner establish a good shot sequence, but first, we do a short recall on the optimization model describing a shot sequence.

Given a cue ball position on the table, we can first elaborate the list of possible shots from that point in order of difficulty. A typical list would include a combination of the following shots:

- Straight shots: The cue-ball first collides with the target-ball and it goes directly in the pocket.
- Combination shots: The cue-ball first collides with a given ball, which sends it colliding on the target-ball and sends it in the pocket.
- Kick shots: The cue-ball first hits the rail, and then collides with the target-ball to send it in the pocket.
- Bank shots: The cue-ball first hits the target-ball and has it rebound on the rail to then end in the pocket.
- Cluster shots: The cue-ball sinks the target ball directly or indirectly, and then collides with a given cluster to disperse or better reposition the balls.

Many of these shots are not necessarily viable candidates as they can be quite complex to execute (i.e. long shots) and have a minimal success rate. For this reason, only the first few (best ones) are considered. The idea we propose to use is in continuity with what

was discussed in the Future Work section of [16] with advanced methods for selecting shot sequences.

Instead of partitioning the search space by discretizing the continuous shot parameters, we will analyse the table state to define a sequence of shots and repositioning targets, and each single shot will be solved as a sub-problem by a non-linear optimization library ([24] was used in this case).

If we define $b(u)$ as the target ball position after fixing the shot parameters u, p as the pocket for the given target ball, u_n as the shot parameters of our n^{th} shot and c_t as the position reached by the cue-ball, we can minimize function f as:

$$\begin{aligned} \min f(u_0, u_1, ..., u_n) = \text{Posval}(c_0, c_1, ...c_n) \ \forall t : 0...n \\ \text{s.t. } \|b_t(u_t) - p_t\| = 0, t = 1, ..., n. \end{aligned} \tag{1}$$

The function $Posval()$ correspond to an estimate of the value of the position in regards to a subsequent shot. For a given cue-ball c_i position on the table, $Posval(c_i)$ will return a very small value if the next shot can be easily done from this point. Details on the computation of this function will be presented in subsection 3.3.

Looking again at the function (1), we wish to search for the repositioning targets $c_0, c_1, ...c_n$ that will be reachable with the shot parameters u, and minimize the position value at the same time. What was previously proposed to solve this model was to do an evaluation of the table state by scanning the table to find the best positions for each ball-pocket combination, such as to establish the most desirable repositioning targets $c_0, c_1, ...c_n$. This proved problematic in our preliminary tests as restricting the optimization to get the cue-ball as close as possible to a target may is not even possible in some cases, and in these cases the value of the position reached is often worthless.

What we propose to do instead is to minimize the $Posval()$ function to get a good position value. Thus, if we combine these two aspects together in a single function, we will have (for one ball):

$$\begin{aligned} \min f(u) = \text{Posval}(c) + \rho_1 ||b_{\text{cue}}(u) - c|| \\ \text{s.t. } \|b(u) - p\| = 0. \end{aligned} \tag{2}$$

We now try to find the shot parameters that minimize our position value, but at the same time, we also try to reposition the cue ball at a specific place on the table (corresponding to our previous evaluation of good positions to reach). Adding this new parameter will also help us in situations where we don't have an initial guess for our optimization method that gives us a valid position value. By guiding optimization search this way, we will have better chances of success at finding a good position value for our cue-ball if the initial guess left us in a position where $Posval()$ was null (if no shot is possible $Posval()$ is set to return an extreme value).

3.1 Partially ordered search

In this section, we present two related heuristics which are used to specify preferred shot sequences. The motivation for presenting these two approaches is that the first heuristic specified may be too restrictive in regards to the value given to some shot sequences. We suspect however that it may behave better when less noise is added to the shot parameters. As such, both are presented and later discussed in the results section.

3.1.1 Preferred shot sequence using ball distances

The first heuristic we propose to select a shot sequence is a simple shortest-path approach. Indeed, if we step back and look at a given table state, the most successful sequence of shots will usually be the one where the cue-ball travels the least distance. The less distance it has to cover, the less chances it has to deviate from its aimed trajectory or to collide with an obstacle. It is also an obvious behaviour observed when watching professional players. They will rarely cross over the whole table to do a shot in the distance if a closer shot is possible. It is also usually easier to control the cue-ball position when less speed is given to the cue ball.

Given an ordered sequence of balls $\{ball_1, ball_2, ..., ball_n\}$, and their position in (x, y) the table defined by $pos(ball_i)$, the value (H_1) associated with this shot sequence will be computed as:

$$H_1 = \sum_{i=1}^{n-1} ||pos(ball_i) - pos(ball_{i+1})||. \tag{3}$$

If we have a look at the same example that was used in [16] to demonstrate the importance of planning in some specific situations, we can see that by using the shortest-path search we just described to determine the ball sequence, the correct ball would have been selected and the table state would be finished with a much higher success percentage.

If the first shot is selected as can be seen in figure 3, the total distance covering the shortest path between the balls corresponds to 3.77 meters, while if the shot 2 is selected (figure 4), only 2.63 meters separate the balls, and less distance will be covered by our cue ball.

3.1.2 Ball-order clustering

A different heuristic we propose in this section is based on the same idea as the one in 3.1 but not as restrictive. Since different games may require different tactics and heuristics, we present the two approaches even though our final tests suggest that the ball-order clustering heuristic introduced next gives better results.

Specifying shot-sequences using the previous heuristic can be too constraining in some situations. For example, in figure 5, we can see that if we choose to pocket the ball in the bottom left corner, as long as we choose to reposition for one of the three closest balls, the overall distance covered by the cue ball will most likely be very similar. In some other situations however, if we put too much importance on respecting the specified order of

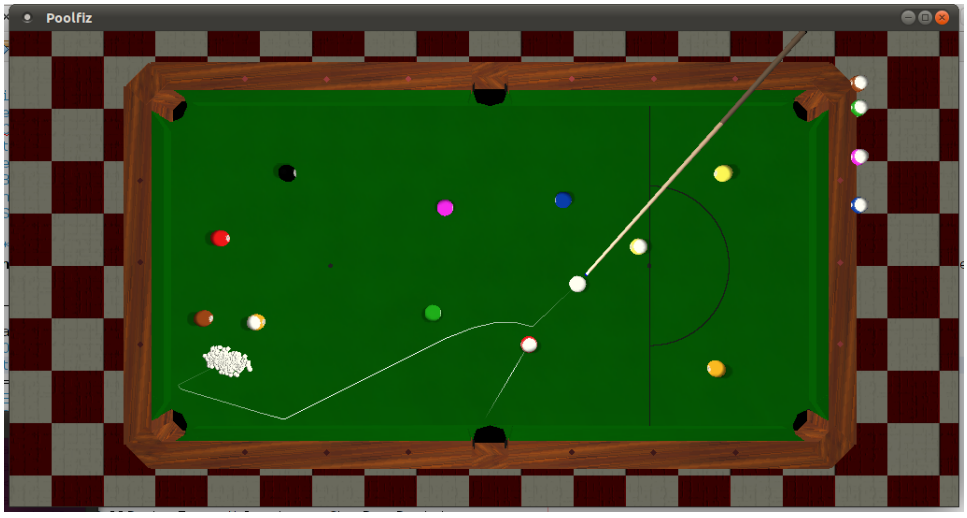

Figure 3: In this figure, if we choose to reposition for the ball in the bottom left of the table, the shortest path separating all of the remaining balls will be of 3.77 meters. The success-rate to finish the table was of 54% in this case, based on 500 trials.

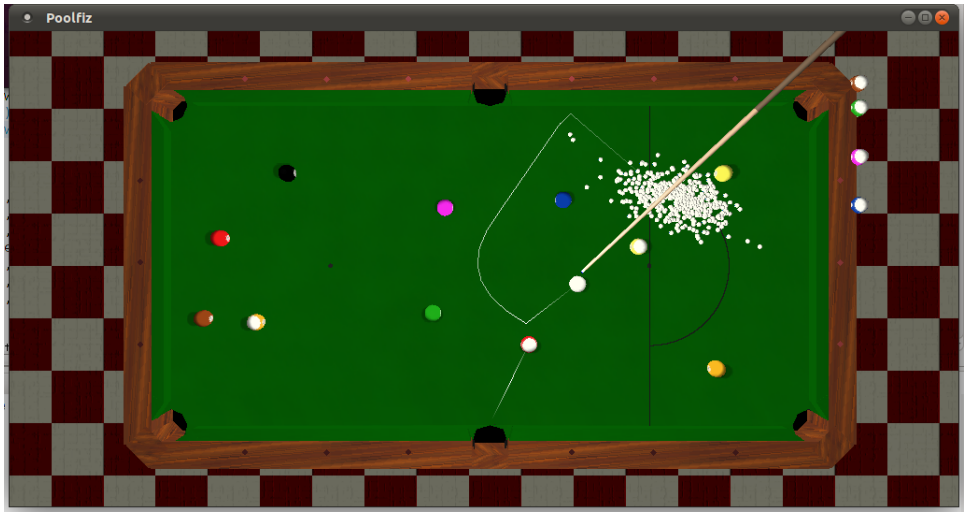

Figure 4: In this figure, if we choose to reposition for the ball in the top right of the table, the shortest path separating all of the remaining balls will be of 2.63 meters. The success-rate to finish the table was of 96% in this case, based on 500 trials.

shots, we may end-up selecting one which will is slightly riskier without having gained any significant reward for the risk taken. Thus, by evaluating the table state and clustering balls prior to evaluating the shortest path between them, we can insure that a riskier shot is only selected when necessary, and that we consider the table as a global state where we want to pocket all the balls in a given area of the table before proceeding to the next one.

Determining the clusters to which each ball belongs is actually a problem in itself. However, in this case, since we only want a good approximate and are not too concerned with precision, we decided to use a simple k-means clustering approach [20]. We start with 2 clusters, which are initialized at the position of two of our balls. We then iteratively attach each ball to its closest cluster, and recompute the centroid of each of these clusters. We continue until we have converged and our balls are set and no longer switch between clusters.

Finally, we check all distances between balls and their respective clusters, and if this distance is larger than a pre-determined cut-off value (a distance of 30cm was used), the cluster size is incremented (K) and the clusters are re-generated. This step is necessary as the k-means approach requires we specify the number of clusters beforehand. By recomputing our clusters when needed, we are guaranteed to have sufficient distances between each of our clusters, and since we only have a maximum of 8 balls to group in clusters, the computation cost is negligible. The result of this process is illustrated in figure 5.

The choice for not using a fixed cluster size in our evaluation can be explained by the fact that if two balls are slightly far away from each other and still clustered together, then when selecting the preferred shot order they will have the same priority. However we wish to give a higher priority to the closest one, given that a solution is generally found faster by exploring closest shots first.

The heuristic we use will be the same as before. Given an ordered sequence of clusters $cl_1, cl_2, ..., cl_n$, and their position (x, y) on the table defined by $pos(cl_i)$, the value (H_2) associated with this cluster sequence will be computed as:

$$H_2 = \sum_{i=1}^{n-1} ||pos(cl_i) - pos(cl_{i+1})|| \tag{4}$$

3.2 Dynamic programming

Our final addition to the planner is the use of dynamic programming to do a backwards search on the remaining ball-pocket combinations on the table. We described the general idea in section 2.3 but also mentioned how it was intractable to directly implement. However, if we implement a slight variation of it when only a few balls remain on the table, we may actually find good results.

The reason for not using using dynamic programming unless there are only a few shots left on the table is simple; as we advance in the game, less balls on the table means we have less chances of disturbing the table state when executing our shots. If an optimal shot sequence exists to finish the game, we are more likely to find it this way and execute it properly. The hypothesis that no ball-ball collisions will happen in our search is a strong

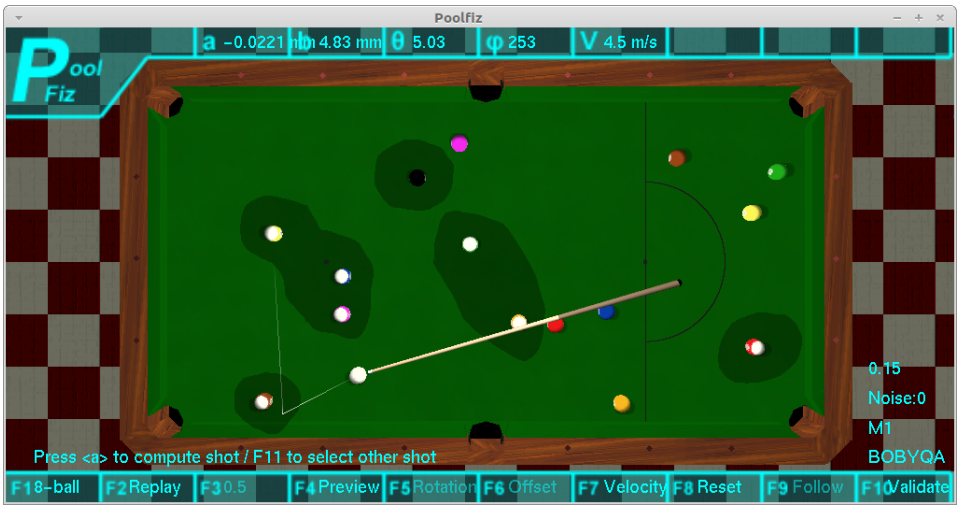

Figure 5: In this figure, we are able to see five different zones which were selected as a result of a k-means clustering of the balls on the table.

one, but we verify it by when performing our backwards search. If a ball in our sequence is blocked after computing previous shots, this specific shot sequence will not be selected, unless no other alternatives exist.

The first step is to establish the desired positions to execute our n^{th} shot. For example, if we desire to pocket balls 1, 2 and 3, we will start by removing our balls 1 and 2 from the table, generate the best position to pocket ball 3 in the easiest pocket, and then compute a shot from this position. If such a shot is successful, we'll return its value and propagate it through our search tree, such that we will then compute the parameters to pocket ball 2 in a pocket, to reposition for ball 3.

Since we have discretized our state space into states we wish to reach, we are no longer guaranteed optimality. However we can still get a good approximation this way and find out if a specific shot sequence is better than another. Our controller being robust, we will also be informed with a good estimate of the difficulty of a shot sequence compared to another, thus will be able to make the right decision.

One important factor to take into account about using this approach is that it will most likely be a very powerful tool for other variants of the game of billiards. The game of Straight for example requires a player to be very careful when selecting his last shot since he will need to position the cue-ball in such a way as to be able to rebound and re-break the balls which are put back in a triangle to continue the game. By planning in advance with DP we will be able to foresee such a situation and select the best shot to continue the game.

3.3 Position evaluation

Up to this point, the efficiency of the $Posval()$ function to define the quality of a position wasn't discussed. The method which was used in [15] relied on a rough estimate of a shot's difficulty, using distances and cut angles to compute shot difficulty measures on the fly. Based

on the 90° rule (see [1]), one can have a good estimate of the best position for the cue-ball, and thus subsequently have a good estimate of the shot's difficulty. The methods described in [26] and [4] also seemingly use similar approaches, however using precomputed coefficients in a tabulated database for a given cue-ball, target-ball position and pocket. Since defining a list of desirable target positions relies heavily on a good evaluation of shot difficulty, the planner will greatly benefit from any improvement in this area.

One of the main issues we have discovered with these methods is that quite often, the direct path between a target-ball and pocket will be clear of obstacles, but a slight deviation of this path which may still pocket the target-ball may be partially blocked by another obstacle ball.

Another problem also arises when we evaluate the value of a corner shot. If the target-ball is sufficiently close to the rail, its chances of success will be better than if it is aligned directly at 45° angle with the pocket, as it can use the rail to reach its final point.

The geometry corresponding to these aspects has been studied in parallel to the writing of this paper and the details can be found in [7]. We will still describe here a few of the key elements which we use for our evaluation. We can see in figure 6 that a direct shot, with the cue-ball and pocket-ball directly aligned with the pocket is actually harder than a shot in which the target-ball is close to the rail (figure 7). The shot closer to the rail will have the possibility of a higher deviation from its path while still ending up in the same pocket, while the direct shot might rebound off the side of the pocket. We can actually divide these areas in two zones, one for direct shots (as shown in 6), and one for rebound shots (as shown in 7).

To correctly compute the value of a corner shot, in the case where no ball is partially blocking the path, we first determine to which zone it belongs. Using some basic geometry, we can use the following developments.

If the ball is in the first zone, we can use the following formula to compute our target ball - pocket difficulty coefficient:

$$Posval() = \Delta\theta_A = \arccos\left(\frac{a^2 + b^2 - c^2}{2ab}\right)$$

where a is defined as the vector between the target ball and one side of the pocket, b as the vector between the target ball and the other side of the pocket, and finally c as the vector between the two sides of the pocket.

If the ball is in the second zone, we use the same formula but use a mirror table to get the position of our ghost ball on the pocket and compute b (see figure 7).

We then use basic geometry to determine any ball is completely or partially blocking its path. To do so requires to do two verifications. The first is to check whether or not the ball is in the region delimited by the angle of the zone for which the ball is pocketable. The second is to check whether or not the ball intersects with one of the sides of the triangle representing the angle. If a ball is partially or completely obstructing the path, we will adjust the angle by computing the tangent between our target-ball and obstacle ball. An example of partially blocked zone is shown in figure 8.

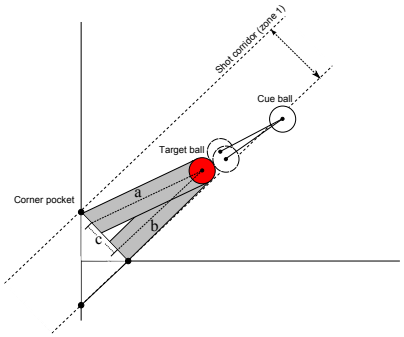

Figure 6: Corner shot illustrating a straight-in shot as well as the corresponding corridor. The ball can only be pocketed directly, if a collision with the rail occurs, it will deviate and rebound off the desired trajectory.

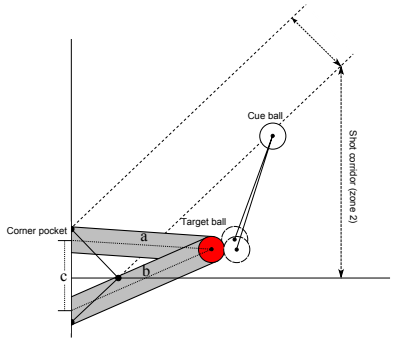

Figure 7: Cornershot illustrating a shot with a greater angle, with a shot corridor including possible rebounds on the rail.

Finally, two last conditions also need to be checked when computing the value of the shot. The first can be seen in figure 9, and corresponds to a case where the cut angle is too small, and we have to actually adjust the angle to correctly represent the zone. The second is if the cue-ball is too close to the target ball. In such a case it may not see the complete angle corresponding to the target-ball – pocket trajectory.

It should be noted that the function we propose here is to evaluate one single position on the table for one single shot. It is just as easily possible to evaluate a position on the table based on all possible shots from that position (e.g. by adding all their values or taking the max). However in the approach we have proposed so far, we assume we are doing a specific sequence of shots, as such, we only verify the position value for the next shot in the sequence.

4 Improvements and additions to the robust controller

The controller presented in [16], while very effective in low-noise situations, was not as impressive when higher noise-levels were used. Faced with the necessity to do more calls to the controller to allow the planner to work effectively, we studied the possibility of changing the model used for the single-shot optimization. The result was a much-improved controller providing better solutions than before with actually less calls to the simulator.

As a reminder, to generate robust shots in the presence of noise, the past controller was based on the use of non-linear optimization methods to minimize the objective function:

$$\min_u f(u) = \max(\text{Posval}(u,\zeta) - \text{Val}_{\min}, 0) + \rho||b_n(u,\zeta) - p||_\infty \ \forall\zeta \tag{5}$$

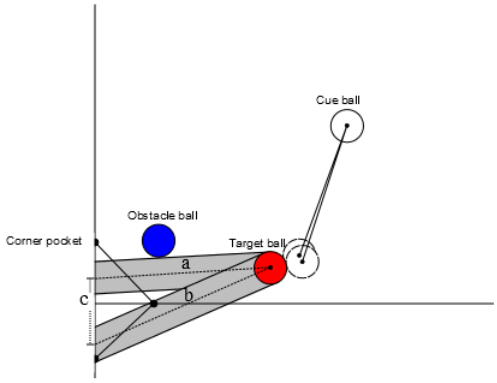

Figure 8: Same corner shot, but this time blocked by an obstacle ball. The zone is then rectified using the tangent point of the line between the obstacle ball and cue ball.

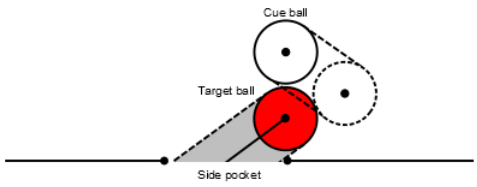

Figure 9: A side shot where the cue ball is positioned very close to the target ball. We can see that a correction will need to be made on the angle corresponding to the shot difficulty since the cue-ball cannot directly reach the position which would normally be desirable for greatest chance of success.

where ζ represents a vector of noise perturbation applied to the parameters in u. Thus, to actually minimize this function, it was required, for every function evaluation, to sample the function several times with different noise parameters to minimize the worst-case scenario (min-max). The problem with this model is that it is highly dependent on the sample-size, and even though it was possible to get better solutions using higher sample-sizes, the solutions found were often only marginally better.

Since the problem we are facing is actually based on a deterministic simulator (using Fastfiz), with a stochastic aspect simulated by the addition of noise to the shot parameters, a smarter search or modeling of the function should be possible in such a way as to provide solutions in a better neighborhood. Thus, most of the improvement we propose in this section are in regards to slight modifications to the optimization models used in [16] in order to minimize the need for more function evaluations. Instead of using a min-max approach, we will use a few domain-based tricks to guide the optimization in finding robust solutions without actually needing to call the black-box simulator as often.

As a reminder, the noise-less model which was used previously to compute the parameters necessary to pocket a target-ball in a specific pocket while aiming to reposition the cue-ball

at a specific position corresponds to:

$$\begin{aligned} &\min f(u) = ||b_{\text{cue}}(u) - t|| \\ &\text{s. t.} \\ &\quad ||b_i(u) - p|| = 0 \end{aligned} \tag{6}$$

where the distance between the cue-ball's final position and a given target t on the table will be minimized, under the constraint of pocketing the target ball. To help find a solution satisfying the constraint of pocketing the target-ball, it can be reformulated as a penalty minimizing its distance with the pocket.

4.1 Directional vector

The success of a shot in a stochastic environment, which we will define here as sinking a target ball into a desired pocket, is closely related to one important element; the parameters used to execute the shot. These parameters must be selected very carefully, in a manner which will not only pocket the target ball, but send it with a velocity that will guarantee its success even under noise perturbations. For an easy shot, however, this doesn't always mean we have to hit the center of the pocket dead center. As a matter of fact, in certain cases where the ball is almost already in the pocket, the direction almost doesn't matter as long as we gently push it in the pocket with a minimal velocity. For other shots, harder ones, the success rate will however be directly related to this direction.

Thus to improve our shot generation technique, one approach we propose here is to compute the center of the zone in which the target ball should be sent. We then trace the directional vector from the target ball to the center of this zone. Finally after executing a shot, we check the cue ball and target ball collision to get the post-impact velocity of the target ball. We then penalize our objective function with the distance between this vector and the directional vector for the center of the pocket. We add this parameter to our objective function with a penalty based on the shot difficulty coefficient. We choose to penalize based on the shot difficulty coefficient since as we said before, for an easy shot it is not as important to hit the pocket dead-center, and this way we may have more room to achieve a better position for the next shot. Supposing our target-ball's velocity after impact is represented by $\tilde{v}_{bi}$, and the vector representing the center of the zone $\tilde{z}_p$, we can define variable $\bar{d} = ||v_{bi} - z_p||$ as the norm of the difference between our two vectors, and $\Delta\theta_A$ our shot difficulty.

The function will now be reformulated as:

$$\begin{aligned} \min_u f(u) = \max(\text{Posval}(b_{\text{cue}}(u)) - \text{Val}_{\min}, 0) + \\ +\rho_1 ||b_{\text{cue}}(u) - t|| \ + \rho_2(\Delta\theta_A * \bar{d}) \end{aligned} \tag{7}$$

We removed the constraint $\rho_0||b_i(u) - p||_\infty$ (to pocket the object ball) from the function as it becomes redundant if the penalty on the directional vector is satisfied.

Figure 10: A side shot in which we see a slight offset from the pocket center (black dotted line), allowing for a better reposition while still guaranteeing a high shot success rate.

4.2 Robust positioning using derivatives

As we just discussed, by using a directional vector to specify the direction in which to send the object ball, we are almost guaranteed that if the penalized parameter is satisfied, our shot will be robust. However one key element we have ignored until now is the repositioning ability of our controller. When we specify a target to reposition for the next ball, we can either analyse the table and find the best possible target, which will usually be right behind a target-ball, or we can try to find the center of the zone corresponding to the area in which the shot remains feasible. In some cases, if we aim for the best possible position for the cue-ball, it may be very isolated, such that the slightest deviation will result in the cue-ball being in a position where it cannot complete the shot. In other cases, if we aim for the center of the largest zone, it might put us in a position that is rather mediocre, leaving us in a hard spot for the next shot. In the past, we have noted problems using both approaches. Thus, to try and find a better way to generate robust shots, we decided to proceed differently.

One of the key aspects making a shot's repositioning value robust or not is the rate of dispersion from the noiseless shot's position in relation to the variation of the shot parameters. Or to simply put, how sensitive will the reposition be to the slight variation in the shot parameters. An example is shown in figure 11 where the computed shot is highly sensitive to the smallest variation in shot parameters, while a second example in figure 12 shows a more robust shot. This aspect can actually be quantified with a very common tool; derivatives. By computing the derivatives of the function used to simulate a shot in relation to the position reached by the ball, we can obtain a very good estimate of its robustness, and we can then add this parameter to the objective function we are trying to minimize to generate safer shots.

The objective function will now have an added parameter $\rho_3 * \eta$ accounting for the

magnitude of the derivative of the function multiplied by a penalty value:

$$\begin{aligned}\min_u f(u) = \max(\text{Posval}(b_{cue}(u)) - \text{Val}_{\min}, 0) \\ +\rho_1||b_{\text{cue}}(u) - t|| \ + \rho_2(\Delta\theta_A * \bar{d}) + \rho_3 * \eta,\end{aligned} \tag{8}$$

where η is computed:

$$\eta = ||\nabla f(u)|| \tag{9}$$

with

$$f(u) = b_i(u) \tag{10}$$

where $f(u)$ is the final resting position of the ball for the shot parameters u and $\nabla f(u)$ is computed using forward finite differences by perturbing the value of each component of u separately while holding the other components at the nominal value. It should be noted that several parameters will have an impact on the outcome of the minimization process, to either generate shots which are more robust, or find solutions satisfying constraints such as pocketing the target ball. In this last function (8), we penalize using three distinct weight parameters. The first parameter, ρ_1, weights on a distance between the cue-ball's final position after the shot, and a target position t. The second penalty value, ρ_2, is applied on the computation of the difference between the desired directional vector and the velocity vector of the target-ball after impact. Since this value has an impact on the shot's robustness, it is indexed to the noise-level used for the current game. Finally the third scalar value, ρ_3, is applied to the value of the derivative of the resulting shot, η. This penalty is also indexed to the noise-level used for the game since it will have an impact on the robustness for the positioning of the cue-ball for the next shot. Optimizing the meta-parameters is indeed a challenge for the application of black box optimization. For the moment these parameters have been set manually after doing many tests and trying various settings. However since the three parameters interact closely together, further study should be done to determine their optimal settings, keeping in mind however that running only 500 games with a time-limit of 5 minutes per game may take up to 40 hours to complete.

This added parameter, which allows for our computed shots to be partially robust concerning reposition, is for the moment computed with finite differentiation. We hope in the future to be able to exploit automatic differentiation tools in order to get the derivatives directly from the simulator, however as was studied in [21] it is not yet clear if the gains will be noticeable.

4.3 Final shot evaluation

The modifications we made to the robust controller here all correspond to a nominal (noiseless) version of the problem. Although the shots computed this way should be fairly robust, they still might behave slightly different than predicted when noise is added. The main reason for this is that several discontinuities can still happen when we execute the shot with noise-induced parameters, such as pocketing the 8-ball by accident, or having the cue-ball pocketed. Although these aspects are part of the nominal model as constraints, they are

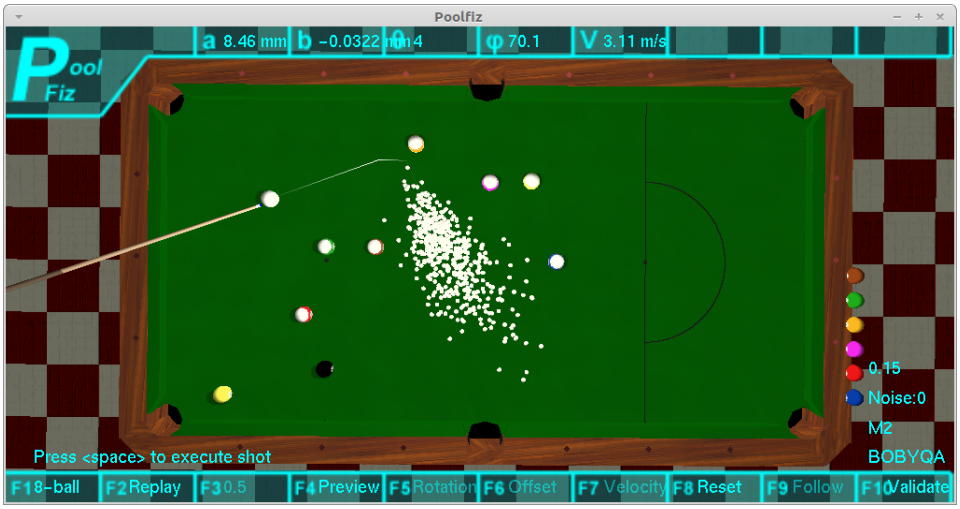

Figure 11: In this shot to the upper side pocket, we are able to see a number of possible positions at which the cue ball will end after being simulated 500 times with random gaussian noise added to the parameters.

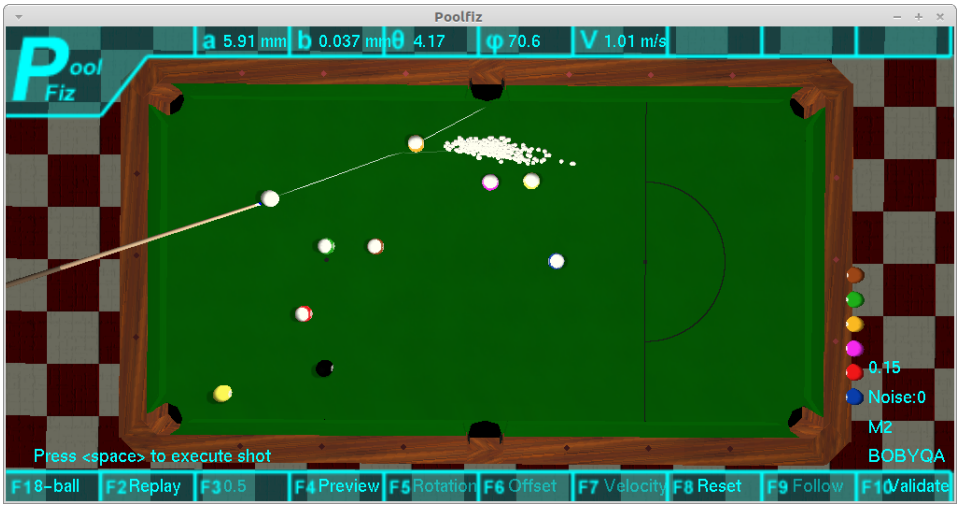

Figure 12: In this shot to the upper side pocket, we are able to see that the resulting positions of the cue ball after the shot are not as dispersed as widely as in the last figure [cite figure]. Thus the norm of the derivative of the position relative to the shot parameters is not as big.

not for the moment treated in a robust fashion. Thus a shot which works perfectly when evaluated without noise-induced parameters may work only 10% of the time when noise is added. What we can do however is use the same robust model as in [16] to evaluate the value of a shot after its nominal counterpart is computed.

As previously explained, the robust controller in [16] made several calls to the simulator to sample the objective function several times before taking the max of these function evaluations to perform a min-max search. The improvements we proposed up until now should actually present us with good solutions even without doing such a sampling. Since calls to the simulator are the most expensive in terms of CPU resources, for every evaluation done by our optimizer to minimize our model, the value of the result is stored given the specific parameters used. Thus as we go along and try to compute shots for various repositioning targets, we gather and keep all shot information in a vector. It is also important to note that the optimization library used for our computations is actually a local optimizer ([24]), which will have a tendency to make more evaluations as it progressively gets closer to the solution. Thus by recording those calls to the simulator, we can gather important information about the robustness of the shot.

When we want to get an idea of the robustness of a solution which was provided by the nominal model, we will simply go through the vector of previous shots, and use the values of all the shots in a given neighbourhood corresponding to the noise level used for the game. If not enough calls were made in the neighbourhood of our solution we can then generate some more to get a better estimate of the shot's value.

To give a specific value to a shot which was computed using the nominal model in (8), we will re-evaluate it, with one call to the following function:

$$f(u) = \max[\max(\text{Posval}(b_{cue}(u)) - \text{Val}_{\min}, 0) \forall \zeta] * \alpha_0 * dist(\zeta, U), \tag{11}$$

where U as the set of the normal range of the parameters under which our shot still remains successful $U = \{\zeta : b_i(u, \zeta) = p\}$. Thus, the parameter ζ will correspond to all possible noise pertubations which could be applied to the shot computed. If not enough parameters are present in the previously-saved vector of shots, more shots will be generated. The parameter α_0 in this equation also carries great importance, since it will correspond to the penalty we apply for the distance with the closest-shot parameters of a missed shot. If all shots succeed with noise parameter ζ then only the $Posval()$ function will carry weight.

5 Results

To demonstrate the potential of the presented approach as well as what we believe to be the state-of-the-art automated billiards player, the results of various tests are presented in this section.

The game on which we are performing tests at this time is one where the aspects of the planner and controller are often very closely interleaved. Some improvements done at the controller-level may generate better shots but leave us in positions from which it will be difficult to finish the game. Thus it is sometimes very difficult to gauge the gain of a

particular modification to the model in the overall problem. The same was noticed from [4] when they tried to dissect the key elements of their winning player.

In our case, we have nonetheless been able to discern some stable and very nice improvements to the results by using the methods described in this paper.

5.1 Testing environment

The test parameters which we used were the same as in [16]. We feel that by generating random tables rather than playing a full-game with breakshot we are able to better illustrate the planning ability of the player. In reality, professional players actually spend a lot of time perfecting their breakshots. Depending on the style of game played, it can have a crucial importance to break defensively or offensively and sometimes requires a great deal of skill. The study of the perfect breakshot is however very dependent on the simulator, and possible advantages can be exploited from analysing it to always leave the player in a favourable position to clean the table (mentioned in [4] as an important element). For the sake of curiosity we will nonetheless provide a few clean-off-the-break results, with a breakshot which was selected after doing some testing on the Fastfiz simulator.

Overall, 500 table states were generated, with balls randomly positioned. The same 500 tables used in this article were used in [16], thus allowing for a direct comparison to be done. The graphs presented show a cumulated average over the number of successes such as to illustrate the fact that the averages tend to stabilize before reaching 500 samples. Success is defined as the player sinking all the balls on the table successfully without missing a single shot. Averages are cumulated with a value of 1 for a success and 0 for a failure. Random noise is added to each of the computed shots parameters. This noise was added in the previous computational pool olympiads (see [11]) to simulate the imperfections which would normally be present on a real pool table, and to simulate different skill levels of real players. We choose to simulate with a noise level of 1x and 0.5x of the parameters used in the last competition (Standard deviations for each parameters: ϕ=0.125 deg, θ=0.1 deg, v=0.075 m/s, a=0.5mm, b= 0.5mm). The same maximal computing time of 10 minutes per table state was allowed. For the standard noise level, the average computing time (Intel Core Duo 2.53ghz) was of approximatively 280 seconds per game, while for $\frac{1}{2}$ noise level, only about 150 seconds per game were required.

5.2 Analysis

In figures 15 and 13, we can observe that one of the major improvements to the success rate was brought by the new shot evaluation function, which is now more precise than before, and thus allows for better repositioning on the table. The gain is more noticeable in higher noise-level tables, which suggests that positioning in these states is probably more of an important factor since a better player (less noise) will still manage to complete his shots even in tougher positions. We can also observe a clear gain of 8% (in figure 15) when planning using the cluster-based shortest path heuristic. This demonstrates that planning can actually be done at a higher level, without the necessity to compute every shot in a

tree-like search. It is hard to comment on the efficiency of the cluster-based heuristic vs shortest path, since the difference between the two approaches is only of 2%, which could be to the statistical significance of the sample.

Finally, figures 14 and 16 show the best results obtained using all of the improvements described in this paper, compared to the results which were originally obtained in [16]. We see a gain of over 20% success rate for the 1x noise level and of 9% for the $\frac{1}{2}$ noise level. Obviously the gain were not as important in lower noise-level table states, since the margin for improvement is smaller, but we nonetheless get very close to a perfect success rate.

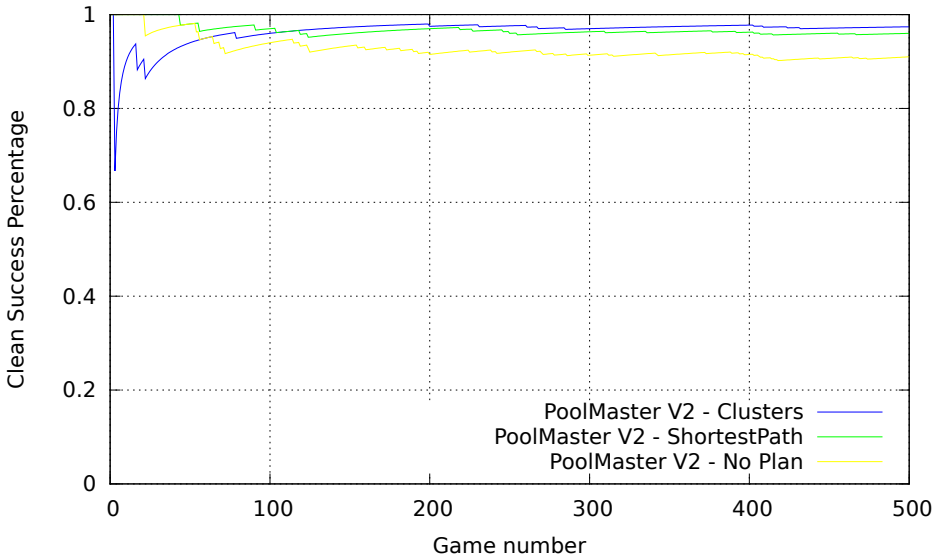

Figure 13: Cumulated averages for the results of 500 games played on randomly dispersed table states with $\frac{1}{2}$x noise level. An average of 91% success rate is achieved when no planning is used, 96% for shortest-path and 97.4% when using the cluster heuristic.

If we now have a look at figures 17 and 18, we can see the outcome of 500 games, where a player would break, and then try to finish all the balls on the table. As in [5] the player was allowed to re-break until successful, and allowed a maximum of 6 minutes to compute all of his shots, with a binary indicator of 1 for victory and 0 for failure. An average success rate of around 80% can be observed for games played on a 1x noise level, while a higher 93% was attained for $\frac{1}{2}$x noise level. The last published results in [5] seemed to be hovering around $60-65\%$ for 1x noise level, and $80-85\%$ for a $\frac{1}{2}$x.

As a disclaimer, it should be noted that it is very hard to do a direct comparison between different players/approaches without actually confronting them in a tournament. Since the tournaments for the Computational Pool Olympiads are not being held anymore we have to find other means for benchmarking. As such, these tests should not be taken as a definite proof of the value of a method over another, but rather as an indication of the potential of the described approach. Many other factors have influence on the outcome when playing with full-games.

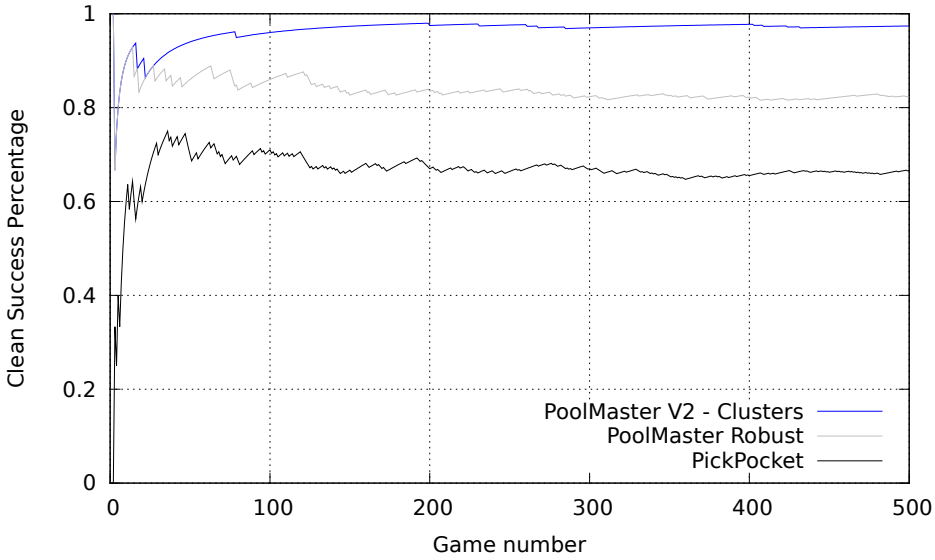

Figure 14: Cumulated averages for the results of 500 games played on randomly dispersed table states with $\frac{1}{2}$x noise level. These results are the results which were first presented in [16]. A noticeable improvement was made over PickPocket ([27]) and the old model, which had a success percentage of 84%

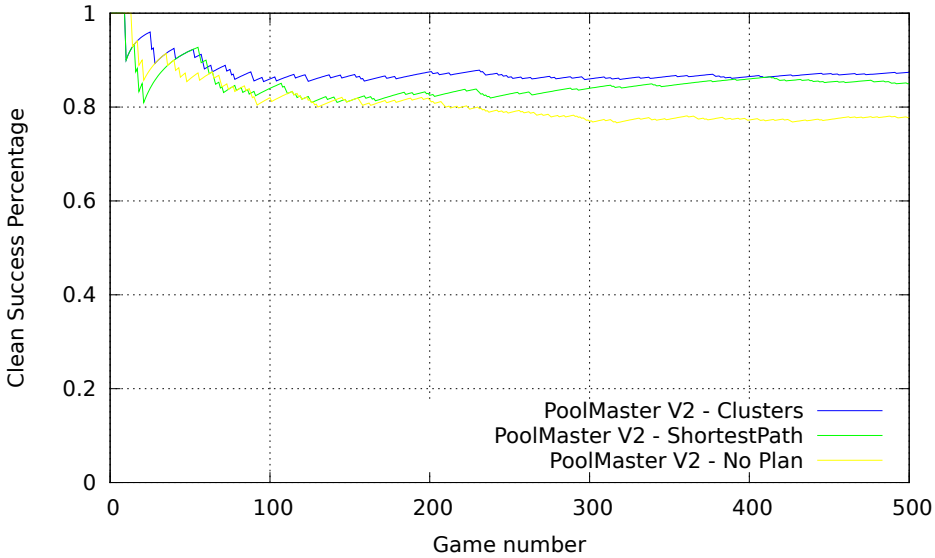

Figure 15: Cumulated averages for the results of 500 games played on randomly dispersed table states with 1x noise level. We can see the results when using the new improvements to the controller (77%), with the shortest-path heuristic (85%), and finally the cluster-based heuristic (87.4%).

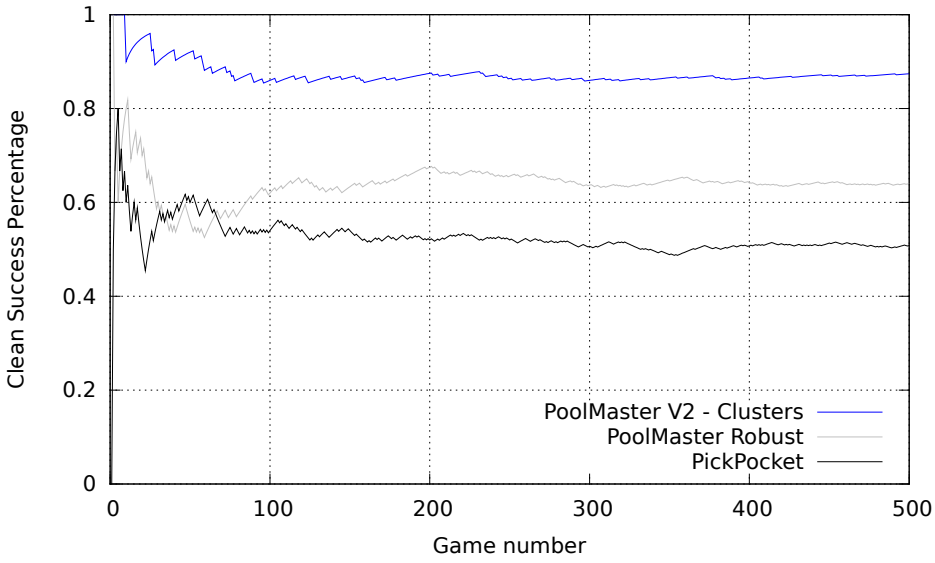

Figure 16: Cumulated averages for the results of 500 games played on randomly dispersed table states with 1x noise level. These results illustrate a gain of over 20% with a success rate of 87.4% while the old version was at 64%.

6 Perspective - Controller or Planner?

Upon reviewing the results presented, it is possible to see that noticeable improvements have been made over the model using the controller alone. It would seem that the additions and modifications made to the controller are responsible for to 13% of the improvements (77% vs 64% in the 1x noise example), which is quite important. However, the overall usage of a two-layered system with planning capabilities also provided a noticeable advantage over a single-layered approach. Although a robust controller by itself accounts for most of the successful games, it still needs to rely on a planner of some sort, even in the simplest form, to get scores closer to perfection.

Thus the question remains; controller or planner? Both, we believe. Future work on various versions of the games of billiards should provide even more insight concerning the necessity for a better planner or controller, and show that even though one may prove more important than the other in some games, both are still needed to achieve the best results.

Future work will most likely be focused on variations of the planner to improve its planning abilities, possibly by using learning algorithms to identify trouble-some states. Such a tool might also prove helpful to predict success rate given specific table state, and thus provide an efficient way to plan safety shots ahead of time.

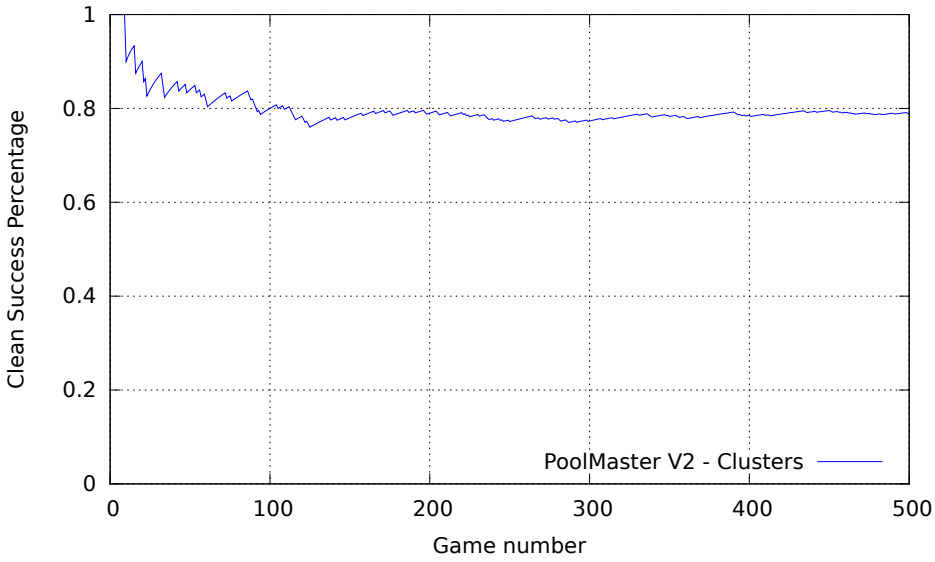

Figure 17: 500 games with breakshot, at a noise level of 1x

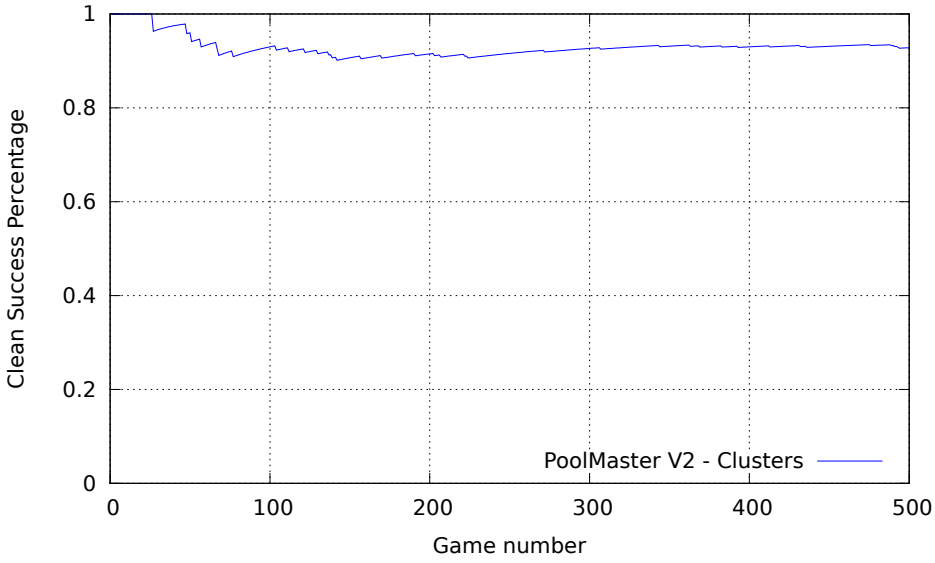

Figure 18: 500 games with breakshot, at a noise level of $\frac{1}{2}$x

References

[1] David Alciatore. *The Illustrated Principles of Pool and Billiards*. Sterling, 2004.

[2] Mohammad Ebne Alian and Saeed Bagheri Shouraki. A fuzzy pool player robot with learning ability. In *WSEAS Trans. on Electronic*, volume 1, pages pp422–425, 2004.

[3] Mohammad Ebne Alian, Saeed Bagheri Shouraki, M.T. Manzuri Shalmani, Pooya Karimian, and Payam Sabzmeydani. Robotshark: a gantry pool player robot. In *ISR 2004:* 35^{th} *Intl. Sym. Rob.*, March 2004.

[4] Christopher Archibald, Alon Altman, and Yoav Shoham. Analysis of a winning computational billiards player. In *IJCAI'09: Proceedings of the 21st international joint conference on Artifical intelligence*, pages 1377–1382, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[5] Christopher Archibald, Alon Altman, and Yoav Shoham. Success, strategy and skill: an experimental study. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 1089–1096, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[6] Christopher Archibald and Yoav Shoham. Modeling billiards games. In Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman, editors, *AAMAS (1)*, pages 193–199. IFAAMAS, 2009.

[7] Nicolas Bureau. Sensibilité des coups au billard. In *CaMUS*, pages 9–26. Université de Sherbrooke, 2012. http://camus.math.usherbrooke.ca/revue/volume2.html.

[8] B.R. Cheng, J.T. Li, and J.S. Yang. Design of the neural-fuzzy compensator for a billiard robot. In *IEEE Intl. Conf. Networking, Sensing & Control*, pages 909–913, March 2004.

[9] Shing Chyi Chua, Eng Kiong Wong, and Voon Chet Koo. Pool balls identification and calibration for a pool robot. In *ROVISP 2003: Proc. Intl. Conf. Robotics, Vision, Information and Signal Processing*, pages 312–315, January 2003.

[10] Shing Chyi Chua, Eng Kiong Wong, Alan W.C. Tan, and Voon Chet Koo. Decision algorithm for pool using fuzzy system. In *iCAiET 2002: Intl. Conf. AI in Eng. & Tech.*, pages 370–375, June 2002.

[11] Michael Greenspan. Pool at the 10^{th} computer olympiad. http://www.ece.queensu.ca/hpages/faculty/greenspan/papers/8ball.pdf.

[12] Michael Greenspan, Joseph Lam, Marc Godard, Imran Zaidi, Sam Jordan, Will Leckie, Ken Anderson, and Donna Dupuis. Toward a competitive pool-playing robot. *Computer*, 41(1):46–53, 2008.

[13] Brian Hammond. A computer vision tangible user interface for mixed reality billiards. Master's thesis, Pace University, 2007.

[14] Niall Rea Hugh Denman and Anil C. Kokaram. Content-based analysis for video from snooker broadcasts. *Computer Vision and Image Understanding*, 92(2/3):176–195, Nov./Dec/ 2003.

[15] Jean-François Landry and Jean-Pierre Dussault. AI Optimization of a Billiard Player. *Journal of Intelligent & Robotic Systems*, 50:399–417, 2007. 10.1007/s10846-007-9172-7.

[16] 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*, (0):–, 2012.

[17] Lars Bo Larsen, Morten Damm Jensen, and Wisdom Kobby Vodzi. Multi modal user interaction in an automatic pool trainer. In *ICMI 2002:* 4^{th} *IEEE Intl. Conf. Multimodal Interfaces*, pages 361–366, Oct. 2002.

[18] Will Leckie and Michael A. Greenspan. An event-based pool physics simulator. In H. Jaap van den Herik, Shun chin Hsu, Tsan sheng Hsu, and H. H. L. M. Donkers, editors, *ACG*, volume 4250 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2006.

[19] Fei Long, Johan Herland, Marie-Christine Tessier, Darrly Naulls, Andrew Roth, Gerhard Roth, and Michael Greenspan. Robotic pool: An experiment in automatic potting. In *IROS 2004: IEEE/RSJ Intl. Conf. Intelligent Robotic Systems*, pages 361–366, Oct. 2004.

[20] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[21] Jorge J. Moré and Stefan M. Wild. Do you trust derivatives or differences? Preprint ANL/MCS-P2067-0312, Argonne Mathematics and Computer Science Division, 2012.

[22] Thomas Nierhoff, Omiros Kourakos, and Sandra Hirche. Playing pool with a dual-armed robot. In *ICRA*, pages 3445–3446. IEEE, 2011.

[23] David F. Percy. Stochastic snooker. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 43(4):pp. 585–594, 1994.

[24] Michael James David Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical report, Department of Applied Mathematics and Theoretical Physics, Cambridge England, 2009.

[25] Sang William Shu. *Automating Skills Using a Robot Snooker Player*. PhD thesis, Bristol University, 1994.

[26] Michael Smith. PickPocket: An artificial intelligence for computer billiards. Master's thesis, University of Alberta, 2006.

[27] Michael Smith. Running the table: an AI for computer billiards. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, AAAI'06, pages 994–999. AAAI Press, 2006.