Project LUDO
Finding an optimal strategy for Ludo

Chirag Dasannacharya
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92092 USA
PID: A59016593
`cdasannacharya@ucsd.edu`

Niraj Yagnik
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92092 USA
PID: A59018067
`nyagnik@ucsd.edu`

Jay Jhaveri
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92092 USA
PID: A59017531
`jjhaveri@ucsd.edu`

Andrew Ghafari
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92092 USA
PID: A59020215
`aghafari@ucsd.edu`

1 Introduction

Ludo is a board game that originated in India and is now a common game in almost all Indian households. The game is played with two or more players with its main objective being to transfer all your pawns (there are 4 per player) from the base to the center of the board (called home) before anyone else. In our project, we will analyze algorithmically the nature of the game and its tradeoffs.

The game is played on a squared grid, having 72 boxes (52 of which are common to all players), 8 of the common ones being safe spots. The game starts with each player having 4 pawns in a corner of the board, and they must move them, according to the result of a dice roll from their respective base to the center of the board. The winner is the player who finishes this task the earliest.

We decided to analyze algorithmically the game Ludo, because it involves an array of strategic decisions and tradeoffs at every move of a pawn. Players must decide which pawn to move and how to threaten their opponents strategically. The players also have numerous trade offs while playing some of them being, keep on chasing an opponent's pawn and save another pawn from being chased, take an instant kill when available or continue chasing an opponent's pawn that's about the get to the destination and many other interesting scenarios. In addition to that, there's also a randomness element, represented by the dice roll, which adds another layer of uncertainty and complexity to the game.

2 Rules Of Ludo

Before we move into analysis, we are first going to remind you of the original rules of them, then the simplifications that we decided to implement to make the analysis easier.

2.1 Basic Rules

1. The game consists of four players, and each player starts with four tokens placed on the base, the large starting area of the player's color (marked in the image below with white stars). Figurative illustration of the board is depicted in Fig 1.

2. Players roll a die to move their tokens clockwise around the board, on the gray squares and starting from the respective colored square by their base.
3. A pawn's journey can only start after getting a dice roll of 6, that takes it from its base to the associated safe spot right outside it.
4. The player decides what pawn they want to move based on their strategy and situation.
5. The chosen pawn moves as many spaces as the value on the die. In case of a six, the player gets an additional turn as well.
6. Each pawn must complete an entire lap from the starting square to the home triangle to finish the journey.
7. The player's primary goal is to move all their tokens from the starting square to the finishing or home triangle (marked with a trophy, which is of the same color as the player's).
8. If a player lands on a square occupied by another player's token, the opponent's token has to go back to its starting square (this may be termed a 'strike').
9. Strikes are incentivized by setting an opponent back and giving the striker an extra turn on the die.
10. The board consists of a total of eight safe zones on the board - the four starting squares, marked with the color of the associated base (these squares are diagonally opposite the arrows), and the four squares marked with a star.
11. Pawns inside the safe zones cannot be struck and are safe from opponents. Any safe zone can accommodate pawns of different colors, and there is no limit on the number of pawns that can land in a single safe zone. When a player's piece reaches the home column of its color, the pawn heads down the column towards the center of its home triangle. A pawn can only be moved to the home triangle with an exact roll. When a player's die roll lands its piece on the home triangle, that piece completes its journey.
12. The first player to move all their tokens to the finishing or home triangle wins the game.

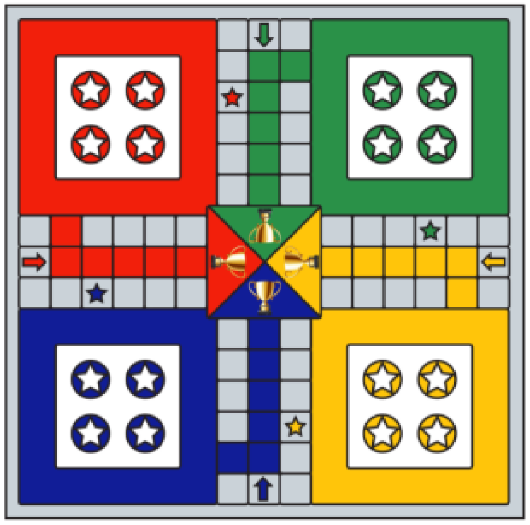
Figure 1: Ludo Board

2.2 Game simplifications

To assist with our analysis, we introduce some further simplifications to the rules:

- We no longer require pawns to start with a six to leave their start base, rather, all pawns start at the safe spot associated with what used to be their start base. Similarly, when struck, pawns return to this square.

- We remove the distinction between the home base and the home stretch. A pawn is assumed to have completed its journey as soon as it enters the home stretch, effectively merging all of the home stretch and the destination into one area. This area can be entered on any number, i.e. a pawn needing two to reach the base can enter on any dice roll exceeding or equal to 2.

The effect of these simplifications is to eliminate the possibility of invalid moves. At any stage, a player can choose any of their pawns that are still in play and choose to move it.

- One more change made to the rules of the game is removing a double chance on 6. In more details, if a player gets a 6, it is still only a one turn game, and after moving a specified pawn 6 blocks, the player's turn is gone and it is now the next player's turn.

We decided to implement this modification to simplify the mathematical analysis of the strategies that we are going to discuss further below and how they compare with one another.

3 Abstract

In the following report, we are going to introduce numerous strategies that we can implement while playing a game of Ludo, starting with the most naive one, the Random Algorithm, then followed by Fast, Aggressive and Defensive. We are going to proof optimality of these algorithms against random, and the penultimate and ultimate superiority compared to Fast. In addition to that, we are going to introduce a hybrid algorithm that accounts for risks and rewards of each pawn for its given state and after hypothetically moving it by the dice roll value, and chooses to move the pawn with the maximum value of that difference. We are also going to prove the superiority of this algorithm against Aggressive and Defensive. We are going to write a pseudocode and a time complexity analysis for each algorithm presented.

4 Related Work

Ludo is a popular board game played by people of all ages. The game is said to have originated in ancient India and soon spread to other parts of the world [1]. The game still amasses a vast audience in the current times. Bhatia et al. (2021) [2] analyzed the gameplay of Ludo on mobile devices. The study shows how the game users enjoyed the game's simplicity and social aspect.

De Voogt et al. (1998) [3] studies the role of dice in the history of board games, including Ludo. Alvi and Ahmed (2011) [4] present a study on the complexity analysis and playing strategies for Ludo. The work, however, only provides an empirical analysis of the performance of the strategy adopted to play Ludo, and no theoretical proof is provided. Our work takes inspiration from this work to get a framework to expand on. Overall, this work provides valuable insights into the complexity of Ludo and its variant race games and offers effective playing strategies for the game.

5 Input and Expected Output for the algorithms

Before we dive deep into the proposed algorithms, it is essential that we state how the input and output of the algorithm would look like:

5.1 inputs

The inputs would consist of the entire game state, and the new dice roll. In detail:

1. The new dice roll
2. The state of the board:
   - Current Player
   - Positions of all pawns with an identifier to map it with corresponding players.
   - Position of safe spots and starting positions of each player

5.2 outputs

The output of the player would simply be the selection of the pawn which would move according to the obtained dice role.

6 Analyzing the Conclusion of Random Algorithm

We are first going to discuss the baseline algorithm for a game of Ludo which is the random algorithm. The random algorithm plays as follows: after rolling the dice, the random algorithm chooses a pawn at random from his set of available pawns as a player (that haven't gotten to home destination yet) and that pawn moves the number of designated spots by the dice. We are going to first discuss if that algorithm concludes on its own, then we are going to discuss its behavior in a game against an opponent.

Playing alone, there are no strikes available as the player is only playing alone, we can then say with certainty that the game will conclude. The game play time might differ from run to run given the stochastic behavior of the random player but with 100% probability we can say that the game concludes, especially after introducing the simplifications discussed above. These remove any bottleneck where the pawn might get stuck and the game doesn't conclude. With a pawn playing alone, it is just a matter of time before it gets the designated dice roll for the game to end, since the pawns can only move forward with no obstacles along their way.

To express this in a mathematical way, we can represent a full game of ludo as getting a lump sum of 208 from dice rolls. (52 needed for each pawn, having 4 pawns as a player.) Given that we can say with certainty that the algorithm will conclude in a maximum number of 208 dice rolls, as each dice roll contributes at least 1 (and up to 6) to this number. This is the upper bound of the runtime, considering each time the dice roll comes out as 1. We can also provide a lower bound using the fact that the dice rolls to at most 6, meaning that the game must last at least 35 turns per player (in this case, with 1 player).

Now let us discuss what will happen in the context of a random player in a game against an opponent.

Since a random behavior can depict any behavior, we might encounter very niche scenarios where the random algorithm doesn't conclude. To explain this, we consider both players to be simulating very focused, intelligent lucky players playing with the specific intent to stretch the game forever. While this is obviously very unlikely, there is a non-zero probability, diminishing with the number of moves but always non-zero, of such play occurring. To stretch the game in this way, this hypothetical player always strikes the furthest pawn of the opponent, while making sure not to make useful progress otherwise (for instance, by playing only backward pawns and getting 1s). As a result, the player's own pawns are stranded where they struck opponent pawns (or earlier), waiting to get struck again. The opponent prioritizes striking this stranded forward pawn and approaches it at high speed with 6s, strikes it, then again focuses on backward pawns. The players continue this play in a cyclic fashion, with the game making no net progress towards completion.

Using the above example, we can see that it is not theoretically necessary that a game of Ludo concludes. There is a family of strategies, one of which is described above, through which extremely lucky players can keep the game at a net standstill. However, this requires a degree of luck with the dice that cannot be assumed to hold over any sustained duration. As a result, the following proofs use a probabilistic approach to describe relative strengths of different playing strategies, while keeping in mind that the stochastic nature of the game means that a significantly 'worse' strategy can still, if lucky, beat a 'better' strategy, although with low likelihood.

6.1 Pseudocode

```
a - Pick a random number between 1 and 4 (inclusive).
b - Move that pawn by dice roll value
```

6.2 Time complexity analysis

```
a - Picking a random number is O(1)
```

```
    b - Moving that pawn is also O(1)
```

The overall time complexity of the algorithm is constant time.

7 Strategies

To generate a strong algorithm, we consider a combination of the following strategies, and prove how certain combinations are better than others -

- 'Fast' - Prioritize the furthest pawn from start base
- 'Strike' - Prioritize striking an opponent, if such opportunity exists
- 'Chase' - Prioritize reducing the distance to an opponent pawn in front of you to raise the opportunity of an instant kill
- 'Safe' - Prioritize entering and remaining in a safe location, if such an opportunity exists.
- 'Run' - Prioritize maintaining distance from opponents behind the pawn, to reduce the success rate of the Chase and Strike approaches

8 Fast Alone

We first consider the initial optimization over a purely random algorithm, i.e. Fast. In Fast, We prioritize the pawn that has already made the most progress, to the exclusion of all others. In practice, this tends to mean that at any given time, only one of the player's pawns are in play, and enjoys the complete and undivided progress offered by the dice. This continues until that pawn either finishes or is struck. In the latter case, it again leaves the start base and acts in the same fashion.

We observe that a simple algorithm such as Fast is capable of being proven more optimal than the purely random algorithm described prior. The proof for the same follows below.

8.1 Pseudocode

```
Define progress array
For each pawn 'candidate' from 1 to 4:
    Get the progress of all pawns (distance from start base)
Move that pawn with the maximum progress.
```

8.2 Time complexity analysis

```
    a - Defining array is O(1)
    b - For each pawn 'candidate' from 1 to 4:
        Get the progress of all pawns (distance from start base) is O(1)
    c - Moving that pawn is O(1).
```

8.2.1 Overall time complexity

The overall time complexity is also constant time. It is in fact linear time with the number of pawns. Hypothetically if we had n pawns that would make this algorithm a linear time one O(n). Here in our case the number of pawns is fixed at 4, which is why this algorithm runs in constant time.

9 Comparing Fast with Random

Ludo is a game that can be won through strategic play, and one way to minimize risk is by using a fast algorithm. As the name suggests, a fast algorithm moves quickly and efficiently across the board, reducing the overall risk of getting killed. In contrast, a random algorithm selects pawns to move randomly, leaving multiple pawns active on the board, which increases the risk of getting captured.

To see why a fast algorithm is better, let's consider the state of the board at an arbitrary time step T before the first pawn of the fast algorithm has reached its destination. We know that a player's expected steps in a one-time step equal 3.5. For a randomized algorithm, it can randomly select one pawn to move at each time step. So for each pawn, the probability of being chosen equals 1/4, and the expected move = $3.5/4$. In the fast algorithm's case, it will only prioritize the furthest active pawn making its probability to move 1. Hence over T time steps, a random pawn will move 3.5T/4, while the average expectation of the fast algo's pawn is 3.5T. So, the fast algorithm pawn will move four times faster than a random pawn. This further reduces the probability of this pawning getting chased and being struck down by 4x. To be precise, the fast algorithm can reduce its risk by 1/4; it is also less likely to be sent back to the starting area.

Furthermore, the Fast strategy reduced the number of active pawns on the board. This reduces the overall risk of getting killed. In contrast, the Random approach does not adopt similar pawn protection policies and thus may leave multiple pawns active on the board, increasing the risk of getting captured. To illustrate this mathematically, let's assume that a player uses the Fast strategy and only has one of its pawns active. Thus the probability of any piece being captured is 1/4. Therefore we can infer that the probability that none of the pawns gets captured is $(3/4)^4$. In contrast, the random strategy does not guarantee that any particular piece will be moved more or less than the other pawn. Thus, all four pawns for a player can be active on the board simultaneously, increasing the overall risk of getting captured.

It is worth noting, however, that only having one active pawn with a fast algorithm also has its disadvantages. Firstly, the damage dealt to the fast algorithm if its solo piece dies would be equivalent to approximately 4 times the damage incurred by a random algorithm's pawn's death. Hence, this, in turn, increases the risk by a multiplicative factor of 4. Secondly, by having only one active pawn, the chance of striking an opponent's pawn is also reduced by 4. However, as stated before, because the pawn is moving 4x times faster than a regular pawn, this effect is negated.

Therefore, considering these base cases, we can safely conclude that the fast algorithm is at the very least 4x times better than the random algorithm. In fact, we consider 4x a weak lower bound because of more complex factors like the safety the fast algorithm achieves after quickly moving one of its pawns to the destination, it reduces the overall risk a lot and increases the chance of winning by a huge margin. This kind of safety mechanism is lacking in the random algorithm, which won't prioritize going home and will more often than not have all 4 active players at risk at all times.

10 Aggressive

We define a strategy named 'aggressive', using the combination of the Fast, Strike and Chase strategies. For this algorithm, we prioritize Strike, if available, followed by Chase, defaulting to Fast if neither one is a viable option.

10.1 Pseudocode

```
Define array PossibleChaser, PossibleStriker
For each pawn `candidate' from 1 to 4
    If opponent pawn present at pawn_location + DiceRoll:
        Add candidate to PossibleStriker
    If opponent present anywhere between pawn_location and destination and dice-
    -roll less than distance in between
        Add candidate to PossibleChaser
If PossibleStriker non-empty:
    Return Fast(PossibleStriker)
Else if PossibleChaser non-empty:
    Return Fast(PossibleChaser)
Else:
    Return Fast(All Pawns)
```

Here, Fast on an array of pawns returns the pawn with the greatest current progress.

10.2 Time complexity

Considering the time complexity of this algorithm, we find that we consider 4 pawns, and find the nearest opponent ahead for each. Finding the nearest opponent ahead can take up to 52 checks, a constant (this can be optimized, but is low enough not to need such optimization). Following this, choosing between pawns are quick operations that loop over arrays at most 4 long, assuming 4 pawns. This gives an overall constant time complexity.

We find that such a strategy is provably more optimal than the direct Fast algorithm described previously. To show the same, we prove that -

Strike alone is more optimal than Fast Strike, when possible, is more optimal than Chase Aggressive as a whole is more optimal that Fast

The proofs for these lemmas follow -

10.3 Comparing Strike with Fast

We compare two strategies - Strike defaulting to Fast, and Fast alone. In the former, the player plays similar to Fast with the notable exception that should an opportunity to strike arise, the player necessarily takes advantage of it. The opponent plays a simple Fast algorithm. This strike may be made by the forward pawn ('forward' here is an outcome of the players playing Fast, where one pawn moves far ahead of its peers), or by a pawn further back and close to home. The latter case can occur when the opponent's forward pawn passes a backward pawn of the player, whether one present at the start base or one stranded somewhere close to the start base by an earlier strike.

It is important to note here that there is a powerful incentive to perform such a strike. The first and most prominent is the damage it does to an opponent. When playing against Fast, all of the opponent's progress is concentrated in the one pawn being struck, causing enormous damage and indirectly resulting in a large reward to the player.

The second significant incentive to strike is that striking offers the player an extra turn. What this effectively means is that the player can choose to strike and then use the extra turn to play as they would have played in any case had they been playing Fast alone.

As a result of the incentives described above, prioritizing Strike over Fast achieves a greater total progress (as summed over pawns) thanks to the extra turn and generates an indirect reward by setting the opponent back, likely by a significant amount. These act to help the strategy dominate over a purely Fast approach, but result in a side effect - the pawn used to perform the strike, if not the forward pawn, will now be stranded till the forward pawn completes its round, and is liable to be struck by the opponent during the opponent's next circuit around the board.

While considering this side effect, however, it is also important to keep in mind that the likelihood of this second strike is of the order of 2/7 - since the opponent plays a Fast game, with no preference given to striking, the stranded pawn appears as a fixed location that the opponent's forward pawn may or may not land on. The likelihood of the strike is the same as the likelihood of a Fast pawn landing on that particular square.

We use symmetry to approximate this likelihood - assuming that most locations are uniform from the point of view of a Fast pawn, we expect a Fast pawn to move at a rate of 3.5 squares per turn (determined by the expected value of the dice being 3.5). Over a circuit, this means that one in 3.5 blocks are landed on, for a total expected strike probability of 2/7.

It is further worth noting that the strike of a stranded pawn represents a small loss to the player - such a pawn is not the player's own forward pawn, and carries relatively little progress with it. If the pawn starts the strike from the start base, the pawn's total progress does not exceed 6 (expected progress 3.5). Here, combining this progress with a 2/7 strike likelihood results in a likely progress loss of 3.5 * 2/7 = 1 square.

On the whole, we find that prioritizing a strike gives a guaranteed extra turn of net progress (at least worth 1 square, more likely 3.5 squares), damages an opponent by robbing them of all their progress (worth at least 1 square, and more likely around 52/2 = 26 squares), and at worst offers a relatively tiny additional target (likely worth around 1 square) in the opponent's following circuit. This trade-off proves that a Strike is worth prioritizing over Fast, with a very high payoff.

10.4 Comparing Strike with Chase

In this lemma, we are going to prove how Strike, if given the chance, is more optimal than Chase and hence should be prioritized over it in the overall aggressive algorithm.

In this segment, we are basically discussing the family of scenarios, where we have an instant kill available (ie where x is the distance behind an opponent pawn, $1 < x < 6$), and it is our turn to play before our opponent pawn's player turn.) We are going to compare this with the other scenario that is chase, which is defined as being also close to an opponent's pawn and getting less than the required amount to kill it, or being a bit more than 6 blocks behind it.

For instance, we can consider one of the possible scenarios and then move on to the proof. A strike example will go as follows: Player A has pawn 1 at position p, and Player B has pawn 1 at position $p + x$, where x is a number that is upper bounded by 6, i.e $1 < x < 6$) and p+x is far away from Player's B destination. We consider here the possibility of Player A getting the dice roll of x, but here we are not looking at it from the perspective of before the dice roll (i.e the possibility of a kill is $1/6$, given that the dice is fair) but we are observing it after the dice roll came out as x, i.e now the Strike opportunity is there.) In the same time, Player A has pawn 2 at position p2, and Player B has pawn 2 at position $p2 + x2$, where $x2 > x$, ie, the dice roll is not enough to kill the second pawn, but instead, in case of choosing this pawn to move, the chase will continue as the pawn 2 of Player A will now be $x2 - x$ blocks behind pawn 2 of Player B. We are going to compare these two scenarios, and prove that the optimal choice is actually to go for the Strike rather than the Chase.

The Chase strategy is actually meant to keep the pressure on an opponent player's pawn, in hopes of sometime in the near future getting just the right amount to get a kill, ie the Chase can be approximated by a futuristic opportunity of getting a kill, whereas a Strike is a current definite chance of killing a pawn. There are two arguments that can be made in favor of Strike. The first is that Chase's strike likelihood will always be less than 1, or else it would be an instant kill opportunity, and it is upper bounded by $1/6$ per turn, since we are at best, one dice roll away from our opponent's pawn and this can be modeled as a $1/6$ chance of striking per turn. In the case of a longer chase, the longer the chase lasts the higher the chance of getting an eventual strike opportunity. But still it is less than the certain probability that we have in a Strike. For instance, if I suppose that there is a 6-turn chase going on, and assuming that my pawn is always one dice roll away from hitting the opponent's pawn, now the expected probability of a strike occurring becomes $(5/6)^6$ which is equal to $1/3$. This can give us a rough idea of the trend of the probability of a Strike given the number of turns the Chase lasts for. Nonetheless, the bottom line is that it is always less than a certain probability of a Strike, being 1.

We assessed the likelihood of having a kill for the two strategies mentioned above, and now we are going to assess the damage made. We are going to measure damage made as a simple linear function $f(x) = x/52$ where x represents how far the pawn is from the start base. Here, the further away from home the more damage that is being made. It starts from 2/52, since we can't kill off the first spot (spawn spot is a safe spot), which is relatively a low number but also represents what is the damage made when you are killed at the very first block outside of home. Given that we removed the necessary '6' to break out of home, the damage made is basically negligeable. The other end of the spectrum is when the pawn is one step away from reaching destination, the damage here is maximum and represented by a damage of 1, since the pawn now has to go all the way back to the start block. Here a counter argument might arise in favor of Chase, that states that Chase is more likely to make the two pawns move forward, and the more the two pawns move forward, the more damage made once a kill is feasible. But in reality, every Strike comes from a previous Chase that turned into a strike once it got lucky, and hence that counter argument is not valid.

Even though, when given the chance to kill, we can opt not to, and make the chase longer, resulting in a bigger damage once we are given another chance of killing, we can argue that, one, the probability of this occurring, ie a second kill, is dim, given the fact that when we are chasing, we restrict ourselves from overshooting, i.e if we are 4 blocks behind an opponent, and we get a dice roll of 5, we will resort to moving a second pawn of ours rather than overshooting a getting our pawn at risk by reversing the Chase. Therefore, at any given time while we are chasing, we have two options, either keep up with the chase, or fall back by not overshooting. Hence, trading a kill for a Chase is not a good option.

In addition to that, we must also not forget the extra turn we get once killing, so in case of having two options, one of killing, one of continuing chase, the definite optimal option will be, kill the pawn, and use the turn to continue chasing the other one, instead of turning both into chases.

10.5 Comparing Aggressive with Fast

We add to the two results proven above to show that the overall, combined aggressive approach dominates a purely Fast approach. We have,

- Strike defaulting to Fast dominates Fast
- Strike dominates Chase in any turn, if available

We now work to show that Strike defaulting to Chase, with Fast as a backup choice (i.e. the aggressive) itself dominates Fast, with the fundamental suggestion that Chase (culminating in Strike) dominates Fast.

We consider the play of a game where one player plays Fast while the other plays Aggressive. Initially, both play similar to Fast. Even as the Aggressive player attempts a distant Chase, this looks identical to Fast. The game changes when the Fast player's forward pawn crosses the Aggressive player's start base. At this point, the Aggressive player's forward pawn effectively becomes stranded as a backward pawn starts a chase.

As discussed previously, a chase has a non-one likelihood of succeeding. Here, as the chase progresses, there are a number of possible outcomes. First, the Aggressive player may successfully finish the chase by striking the Fast player before the chase reaches the Aggressive player's stranded forward pawn. This is the ideal circumstance for the Aggressive player, as they now have 2 pawns with significant progress, while the opponent has no progress at all.

A second possible outcome is one where the Fast pawn stays alive and reaches the Aggressive player's forward pawn. This pawn itself, having initially traveled at a similar speed as the Fast pawn, is expected to be located near the opponent's base. This may result in a strike (with 2/7 probability as described previously), or in the Fast player bypassing the Aggressive player's forward pawn (1 - 2/7 = 5/7 probability). We consider these two sub-cases separately.

In the first sub-case, the Aggressive player loses their forward pawn, the opponent proceeds to finish, while the chaser is now the player's new forward pawn. It starts a new chase with the Fast player's next pawn, having a head start of expected 52/2 = 26 squares.

In the second sub-case, the Fast pawn finishes without striking, leaving 2 pawns near the opponent's home base with about 26 progress each, for a total of 52 progress and a headstart on the next chase.

To analyze the overall effectiveness of the two algorithms, we consider the probabilities and payoffs of these outcomes.

When the chase is successful before reaching the stranded forward pawn, the chasing pawn is now stranded at part of the way (expected 26/2 = 13 squares, midway to the forward pawn). This results in 39 progress total (expected 26 + expected 13) with expected loss of 2/7 * 13 = 3.7 (stranded pawn) for a total yield of $\approx$ 35 squares.

If the chase reaches the forward pawn with a strike, we end up with 26 progress to the opponent's 52, a losing outcome (with a lower probability of 2/7). If no strike occurs, the results are as yet inconclusive, with 52 progress on both sides, and the Aggressive player having a headstart on the next chase.

Calculating the likelihood of a chase succeeding is a challenging problem. This is because the pawn running has the freedom to move at any dice value, while the chasing pawn can only move at values that do not overshoot it. However, being in such a position means that the chaser is already in a striking window, and any sustained play in this fashion is eventually likely to result in a strike. Further complications arise when safe spots are considered. As the 2 pawns being considered have roughly equal speeds, a chaser having fallen behind is not necessarily likely to catch up. On the whole, considering the overall balanced nature of the game, we set the likelihood of a chase being successful as being close to, but less than ½.

In all, we have a potential successful chase (relative payoff 35, probability 1/2), an unsuccessful chase with no strike (relative payoff 0, probability $1/2 * 5/7 = 5/14$) and an unsuccessful chase ending with the loss of a pawn (relative payoff -26, probability $1/2 * 2/7 = 1/7$). This totals to a net payoff of $(35/2) - (26/7)$ or $\approx +14$ squares per half circuit.

11 Defensive Algorithm

We define a strategy named 'defensive', using the combination of the Fast, Defense, and Run strategies. For this algorithm, we prioritize Defense(run), if available, defaulting to Fast if it is not a viable option. Ultimately, the defense algorithm will choose the path which will lead to the least risk value possible, where risk can be quantified as the potential of losing the amount of progress made by the pawn of the player. By strategically choosing the moves that reduce the overall risk of the player i.e. cumulative risk associated with all the pieces, we ensure that at any given state of the game, the move chosen would be equal to the least risk possible for that state of the game for that player. We find that such a strategy is provably more optimal than the direct Fast algorithm described previously.

To show the same, we prove it in the following steps-

- Minimizing Risk Results in Greater Expected Progress
- Defense as a whole is just as optimal Fast

Before diving into proving the above-mentioned lemmas, let's first understand how a safe spot affects the defensive algorithm. If a pawn is landing on a safe spot, we know that the pawn can no longer be killed until it is moved out of the safe spot, and hence we can safely say that its risk is equal to 0. In the following proofs, we would be ignoring the existence of safe spots even if a safe spot exists, the defending algorithm would just consider it as a way of bringing the risk to 0 and then use the distance metric to decide. We will show how the safe spot affects all the above-mentioned lemmas later separately.

11.1 Pseudo Code

```
Define array DangerPiece, SafePiece
For each pawn 'candidate' from 1 to 4
    If opponent pawn before pawn_location
        If pawn_location is equal to a safeSpot location:
            Add candidate to DangerPiece with a value of: pawn_location/52
        Else:
            Add candidate to DangerPiece with a value of:
            pawn_location/52 + dangerRisk(opponent_location, pawn_location)
    If opponent present anywhere between pawn_location and
    destination and dice roll less than distance in between:
        Add candidate to SafePiece
If DangerPiece non-empty:
    Make a new array FinalDangerPiece with pieces from DangerPiece
    having the value equal to the max value in DangerPiece
    Return Fast(FinalDangerPiece)
Else if SafePiece non-empty:
    Return Fast(SafePiece)
Else:
    Return Fast(All Pawns)
```

11.1.1 Pseudo Code for dangerRisk($opponent\_location, pawn\_location$)

```
Diff = opponent_location - pawn_location
Return 1/6*(1/2)^(Diff%6)
```

NOTE: here the dangerRisk() function calculates the 'risk' a pawn faces because of the first opponent pawn encountered before current pawn. The formula $1/6 * (1/2)^{Diff\%6}$

as explained before represents the chance of it being a successful chase, where a successful chase indicates the death of the current_pawn. To elaborate, $1/6$ is the dice probability required for the final blow, and $1/2$ is the probability of a chase being successful which exponentially decreases as the distance between the current_pawn and opponent_pawn increases in multiples of 6.

11.1.2 Time Complexity

Coming on to the time complexity of this algorithm, we run our search for each of the 4 pawns.We find the near opponent behind and ahead of each pawn. Finding the pawn can take up to 52 checks, which is a constant. Following this, choosing between pawns are quick operations that loop over arrays at most 4 long, assuming 4 pawns. While returning if the DangerPiece array is not empty, we would run a loop of maximum iterations of 4 to add the pieces which equate to the maximum risk value. This gives an overall constant time complexity.

11.2 Minimizing Risk Results in Greater Expected Progress

In this lemma, we are going to prove that following the path that leads to the minimum overall risk will end up saving progress than an algorithm following high risk. This in turn would imply that defending the pawn further from the start base first is more optimal than defending the closer pawn in a game of ludo.

As mentioned earlier in the aggressive algorithm's proof, we are going to measure the damage dealt to a player when its pawn dies as a linear function f(x) = $x/52$ where x is the relative position of the pawn compared to that player's starting position. This is true because, the higher a pawn's relative position, i.e. x, the higher the number of rolls and turns that have been invested into moving that pawn there and hence has high damage if it dies.

Keeping this in mind, consider the state of the game where a player has two active pawns on the board with safe spots disabled. The first pawn is at relative location $x1$ and the second is at relative location $x2$, such that $x2 > x1$. There can arise a situation wherein both pawns are in spots that are considered under risk.

Note that we consider a position/spot under risk of dying if at least one opponent pawn lies anywhere behind the current pawn. This risk of dying is proportional to the current pawn's progress made and also to the distance of the opponent pawn to the current pawn. The instant death window occurs when the distance between the opponent's pawn and our pawn is $<= 6$, bounded by the maximum value a dice roll can give.

So, when we consider the situation wherein both players are at risk, we know that there is an enemy pawn that may end up killing it the next turn that the player gets. In this kind of situation, if we get a dice roll of, say r, such that $x1 + r$ and $x2 + r$ both gets the respective pawns out of their instant death window, then by the linear function of distance we mentioned above, it is mathematically more optimal to secure/save the pawn at $x2$, i.e. the pawn which is farther from the start base, than to save the pawn at $x1$, which is the one near the start base because if the one that is the most farthest dies, it would incur a bigger net loss for the player than the one closer would do if it died by the next turn. Further, in terms of risk levels, the overall risk would be at minimum when the pawn at $x2$ is moved out of the instant death range, hence keeping the hypothesis of low risk, greater progress saved true.

To generalize, by choosing to save a high value pawn over a low value one, it will bring the net risk and potential loss of progress at a lower minima than by saving a pawn before it. Hence, our original statement that given any state of the game the algorithm will choose the path which will lead to the minimum risk.

Mathematically, Let's consider two moves: Move A and Move B. For each move, we'll calculate the expected value by taking into account the probability of a successful death by an opponent pawn and the respective gains and losses.

Let P_A and P_B be the probabilities of a successful death by an opponent pawn for Move A and Move B, respectively. As given, the probability of a successful death is calculated as:

$P_x = 1/6 * (1/2)^{(Diff\%6)}$

Where Diff is the distance between the current pawn and the opponent's pawn, and x refers to the pawn position by performing either A or B moves.

Let G_A and G_B represent the expected gains for Move A and Move B, and L_A and L_B represent the expected losses for Move A and Move B.

The expected value (EV) for each move can be calculated as:

$EV_A = G_A - P_A * L_A$

$EV_B = G_B - P_B * L_B$

Note: For us, our gains G_x and losses L_x solely depend on the linear function of distance: f(x) = $x/52$ we defined before and indicates the benefits and losses a position may lead to for the player. To simplify, G_x may be seen as a term that indicates progress made, while L_x can be seen as the term that quantifies what progress a player might lose if that pawn dies.

Now let us consider 2 cases:

1. When $G_A > G_B$, i.e. pawn A is farther than pawn B. In this case, if both moves A and B have comparable losses, we would have a higher expected value of overall steps for move A, such that $EV_A > EV_B$. Hence, when risks are equal the default algorithm will default on the fast algorithm.
2. When risk by move A is higher than move B. Since Move A has a higher risk than Move B, the probability of a successful death, P_A, will be greater than P_B. Assuming the expected gains (G_A and G_B) are similar for both moves, we can conclude that the expected loss L_A will be more significant for Move A than the expected loss L_B for Move B.

Hence, we want to overall maximize the term EV_x. Because G_x and L_x are solely dependent on the pawns current position, it will be not a variable as far as making a decision goes. Hence, the sole responsible would lie on the term P_x which is dependent on the risk of the pawn at that function. Hence, by minimizing the risk, we would in turn lead to a overall higher expected value, and hence greater progress overall.

Now, when safe spots are enabled for the game, the path chosen should still be the path with least risk for the given state of the game. Again consider a similar instance as mentioned before with two pawns of the player at relative positions x1 and x2, such that $x2 > x1$ and both are under risk. For a given dice roll, the algorithm will still behave in a similar way as mentioned in the non safe spot case, except in 1 situation. For a given dice roll r, such that $x1 + r$ lands on a safe spot, while $x2 + r$ does not escape the instant death range upper bounded by 6, the new risk at $x1 + r$ is 0, while the new risk at $x2 + r$ is still >0. The logical step would be to secure the pawn closer to start, as the probability for the pawn $p2$ dying at $x2$ and $x2 + r$ will be equal to $1/6$ while the probability of dying for pawn $p1$ at $x1 + r$ drops to 0 from $1/6$ at $x1$. Our algorithm, focusing on selecting the minimal risk, would pick to move the pawn p1, which would have been the logical choice too.

Hence, even with safe spots enabled, the algorithm following the least risk path would still be more optimal than an algorithm not weighing the risk into the equation.

11.3 Defense algorithm is more optimal than Fast algorithm

To establish this proof, we will employ a series of conditions where we state that the Defense Strategy would outwit the Fast algorithm. Given the family of scenarios where the pawn of the player adopting the defense strategy, A currently needs to catch up to the pawn of the player adopting the fast strategy, B. In such cases, the pawn adopting the defensive strategy would also default to the fast strategy as it no longer has any associated risk. The pawn which had adopted the fast strategy from the start would have ideally made 4x more progress than the pawns adopting the random strategy. The same would be true for the pawn, which initially adopted the defensive strategy but defaulted to the fast algorithm while it was lagging. If the pawn of player A has an opportunity to kill the pawn of player B, then it would take this opportunity to strike the player as the risk associated with this move would still remain 0 for that player. Similar would be the case If the pawn of player A has a chance to overtake the opponent, it would instead not make the forward movement and would revert to the defense strategy to get a new pawn in the game. This would ensure that the overall risk associated with the pawns of player A don't increase.

In contrast, if the positions were reversed, where the pawn of player A is leading the pawn of player B, which is adopting the fast strategy, and the pawn of player B has the opportunity to overtake the pawn of player B, then it would make the overtaking move. By doing so, it would increase its risk of getting struck down by the pawn of player A to $1/6$. Thus we show that the defensive strategy is as good as the fast strategy for all scenarios except in scenarios when the pawn has the opportunity to overtake, where the defensive player would be a more optimal player.

To summarize we can further split the proof into 4 cases/different scenarios:

1. When an opponent's pawn is within striking distance of the current player's pawn (i.e., within 6 steps), the defensive algorithm prioritizes moving the pawn out of danger or to a safe spot. This reduces the risk of losing progress, as captured pawns must start over from their respective bases. In contrast, the fast algorithm may continue to prioritize moving forward without considering the risk, increasing the probability of the pawn being captured and losing progress.
2. When multiple pawns are at risk, the defensive algorithm considers the potential loss in progress and chooses the move that minimizes the overall risk. The fast algorithm, however, may not make such considerations and could result in suboptimal choices that lead to a greater overall loss of progress.
3. The defensive algorithm's adaptability allows it to switch between fast and defensive modes depending on the situation, while the fast algorithm lacks such flexibility.
4. In scenarios where the defensive algorithm has an opportunity to capture an opponent's pawn, it may choose to do so, as the risk associated with the move is still zero. This can further slow down the opponent's progress and indirectly contribute to the defensive player's advantage.

11.3.1 How safe spots would affect the defensive algorithm vs fast algorithm

Now, when safe spots are enabled, the defensive algorithm would be further strengthened by having the option to remain stationary at a safe spot and in turn increasing the chance that the fast algorithm's pawn would overtake it. To illustrate this, imagine a scenario where a defensive pawn $d1$ is standing on top of a safe spot at position $x1$. Now, let's say this pawn $d1$ is inside the instant death range of an opponent fast moving pawn is $f1$. In the next turn, on a dice roll of say r, if $x1 + r$ is still within the instant death range of the pawn $f1$, the defensive algorithm would choose not to take that move, but instead take a move that would reduce the overall risk instead of moving it (note: if no other pawn is available a pawn at the start base would be chosen to move!). This would mean that after that turn, the probability of pawn d1 dying would still be 0. Now, in the case where the roles were reversed such that the fast algorithm's pawn f1 is on the safe spot at position $x1$ and is in the instant death range of the defensive pawn d1. On a dice roll of r such that $x1 + r$ is still inside the instant death range of pawn $d1$, the fast algorithm will still make that move, increasing the overall risk of the fast player and increasing the probability of the fast pawn dying from a negligible amount almost equal to 0 to $1/6$. Hence, the introduction of safe spots would make the defensive algorithm more optimal against the fast player.

Therefore, no matter the scenario, a player playing using the defensive algorithm would behave as good as, if not better, than a player playing using the fast algorithm.

Finally, we can summarize by establishing the following points:

1. When there is no risk involved (i.e., no opponent pawns nearby), both strategies are essentially equivalent, as they both prioritize moving farthest pawns forward.

2. The key difference between the strategies arises when a player's pawn is at risk of being captured by an opponent, wherein the defensive would choose the move with the least risk, while the fast algorithm wouldn't be affected.

12 Hybrid Algorithm Description

We are now going to introduce a new and improved hybrid algorithm that factors risks and rewards given the state of the board (before and after moving according to dice roll) for every player's pawn and chooses to move the pawn that maximizes the function reward - risk. In case of a draw we will be resorting to the Fast Algorithm.

Let us now dive deeper into the formulas of risks and reward functions:

12.1 Risk Function

Our risk current function takes into account a boolean variable SafeSpot, which is equal to one if we are on a safe spot, and it multiplies it by the pawn's self progress (number of blocks away from the start base). This result is multiplied by the risk of dying, P(dying), which is 1 - P(staying alive). P(staying alive) is 1 - P(getting killed by any opponent's pawn) with the latter being a product of P(kill) for all opposing opponents.

P(dying) = 1- P(staying alive) = 1 - Π(1-P(kill)). Where the product is over all opposing pawns inside the 6 blocks that are immediately behind that specific pawn.

The resulting formula would turn out to be equal to:

$$Risk = (SafeSpot) * (Value(pawn)) * (1 - \prod_{\text{last 6 squares}}^{\text{all threatening pawns}} (1 - Prob(kill)))$$

Note that this function is going to be used for current state and future state (after hypothetically moving that specific pawn).

12.2 Reward function

Our reward function goes as follows:

For current state, It is the sum of Probability of Kill * Value of kill (which is defined as progress of opponent's pawn getting struck), where the sum is implemented over all killable pawns . Here we are only going to look one dice roll into the future, i.e. only consider the pawns in the next 6 blocks from that pawn as 'killable'.

$$CurrentReward = \sum_{\text{forward 6 squares}}^{\text{killable pawns}} (Prob(kill) * Value(kill))$$

For the future state (i.e after moving the hypothetical pawn by the value of the dice roll), we are going to calculate the reward function as also sum of Probability of Kill * Value of kill (here we are also going to include not only the 6 blocks from that pawn's position, but also the current block, which is at a distance 0 from that pawn.) + 3.5 * InsKill + DiceRoll.

$$FutureReward = DiceRoll + 3.5 * (InstantKill) + \sum_{\text{including new position as 0}}^{\text{killable pawns}} (Prob(kill) * Value(kill))$$

For further clarifications, we decided to also include the block at distance zero, to take into account the chance of the pawn landing on an opponent's block, where the P(kill) would be equal to one. In addition to that the 3.5 * InsKill is just the reward of the extra dice roll in case of an instant kill (represented by a boolean variable 'InsKill') and the 3.5 coefficient is the expected value of that extra turn's dice roll value.

12.3 Likelihood of chase: a linear function:

To estimate the likelihood of a successful chase, we propose a linear function that would depend on the current pawn's position and the opponent pawn's position. This linear function can be considered as a helper function to the overall risk-reward system mentioned before. To summarize, this linear function is used to calculate the probability of an opponent pawn restarting a chase our current pawn got out of.

To further elaborate, consider a situation where there is a pawn p1 at position x1 and an opponent pawn o1 at x2 such that $x1 > x2$. For a given dice roll r, such that x1+r is equal to a position that is outside the instant death range of the opponent pawn o1. In this scenario, the linear function helps in calculating the probability that in the next turn, pawn o1 will be successfully able to resume the original chase.

This can be calculated using the following formula, **Likelihood of successful chase** $= P(Chase\_Continuance\_Success) = 1/2 * P(diceRoll >= victim\_diceRoll)$

Here, 1/2, as explained in the above sections, indicates the probability of a successful chase. Here successful chase means a chase that would eventually be a successful kill from the opponent's perspective or a death from the current player/ victim's perspective. Further, the probability $P(diceRoll >= victim\_diceRoll)$ denotes the chance of the opponent getting a dice roll that is greater than equal to the victim's dice roll, such that the chase can be resumed. This is the case because to resume the chase, the attacker has to at least move the same amount of blocks the victim moved or more.

The value of $P(diceRoll >= victim\_diceRoll)$ would change based on the victim's original dice roll used to get out of the instant death range of the opponent. The values are illustrated in the following table.

Original DiceRoll	$P(diceRoll >= victim\_diceRoll)$	$P(Chase\_Continuance\_Success)$
1	1	0.5
2	5/6	5/12
3	4/6	4/12
4	3/6	3/12
5	2/6	2/12
6	1/6	1/12

NOTE: The likelihood of a successful chase when the original dice roll is equal to 0 would always be equal to 1 because the original pawn never left the instant death zone in the first place!

12.4 Putting it all together

At any given turn, the following algorithm will be implemented: Assuming it is Player A's turn, we are going to loop around all currently available to move pawns of A (excluding the ones already at destination), and calculate the risk and reward value for current state and for future state (after moving that pawn by the value of the dice roll). For each pawn, we will have a value which will be called 'MoveAdvantage' which is equal to the difference between reward and risk for both states which represents how advantageous it is to move that specific pawn.

$$MoveAdvantage = (FutureReward - FutureRisk) - (CurrentReward - CurrentRisk)$$

After doing all of that for all of the available pawns, we will choose the pawn with the highest 'MoveAdvantage'.

In case of a draw between all available pawns, i.e. all pawns have the same 'MoveAdvantage', we will resort to Fast, meaning moving the pawn that is closest to the destination. One possible scenario of this occurring is when pawns that are in play, have no one in the front 6 blocks and back 6 blocks, in the current state and the future state. In this case we will move the furthest pawn from start base.

12.4.1 Pseudocode

```
Define AdvantageArray
For pawn 'candidate' from 1 to 4:
        Calculate MoveAdvantage using candidate (using formulae above)
        Add advantage to AdvantageArray
Find index(es) with greatest value in AdvantageArray
If only one such index exists:
        Return that pawn
If multiple indices exist:
        Return(Fast(Indices))
```

Here, Fast returns the pawn with the greatest current progress from a given array.

12.5 Time Complexity

We see that the above code also runs in constant time. In more depth, this code loops around all pawns of a player (capped at 4), and for each pawn, does risk/reward calculation, for both current state and future state. These calculations are also done in constant time as well (using the formulae stated). Overall, this means that this algorithm also runs in constant time.

12.6 Comparing Hybrid with Aggressive

We proceed to show that the proposed hybrid algorithm is more optimal than the previously described aggressive algorithm in the general case. To begin, we start by comparing the behavior of the aggressive algorithm with the reward function of the hybrid strategy. The aggressive algorithm chooses to strike first at any possible opportunity, and chase down more valuable pawns if there is no such opportunity, while defaulting to fast.

The reward function of the hybrid algorithm acts in a fairly similar manner - it assesses each position by its value measured in expected progress deniable to opponents, and prioritizes kills to a large degree through the steep rise in $P\_kill(0) * Value\_kill$ at the new position 0 distance from the opponent (with $P\_kill(0) = 1$, representing the choice to kill), the boolean 'InsKill' term (set to 1 for a guaranteed kill) and the 3.5 tile additional bonus for a guaranteed kill, representing the expected progress possible in the additional turn generated by the kill. Like the aggressive strategy, the hybrid strategy too eventually defaults to fast.

There are a few notable differences between the aggressive and hybrid reward gameplay styles, however. The first is that the hybrid reward function looks only 6 tiles ahead, unlike the aggressive strategy, which looks as far ahead as it takes to find the next opponent pawn, however the hybrid strategy also considers all pawns in the current 6-window, rather than only the first. The second major difference is the integration of the strategy with the defense mechanism that results from trying to reduce risk.

We find that each of these steps is necessarily an improvement over a purely aggressive algorithm. For the first difference, we first observe the effect of restricting the pawn's vision to six squares ahead. As discussed in the justification for evaluating the p_kill values, the likelihood of a successful chase decreases steeply with the distance between the pawn and the opponent. Choosing to look far ahead without appropriately decreasing the weightage of such a far consideration effectively acts as a distraction from other, more nearby and pressing concerns as well as the need to make net progress. Cases described in the aggressive algorithm where the aggressive player's forward pawn is struck by the fast player's forward pawn while being chased by the aggressive player's second pawn become drastically less likely when the vision is restricted - with restricted vision, the chasing pawn gives up the chase when p_kill deems it relatively hopeless, and once again prioritizes moving the forward pawn, carrying it safely away from the fast player's forward pawn.

Taking all pawns in the current 6-window is also a dominant strategy compared to considering only the first one. In doing so, the hybrid algorithm does make use of all the information that the aggressive method considers, while also taking additional relevant factors into account, namely the other pawns in question. This helps the hybrid strategy, for example, choose to pursue a more rewarding target, such as a sequence of 3 enemy pawns in close proximity, over a slightly easier but less worthwhile target of a single enemy pawn that is closer. When the hybrid reward function chooses not to act the same way as the aggressive strategy, it is necessarily because it has found an even better alternative.

Finally, we consider the second and most visible difference - the hybrid strategy takes risk into consideration. In doing so, the hybrid approach is able to 'rescue' pawns that are at risk, in particular those that are of higher value. This becomes especially prominent when there is a choice between continuing a difficult, low value chase and saving a high value pawn from an opponent's chase. The risk calculations used provide a calculation of the expected loss involved with any possible choice. By offsetting the aggressive reward function's behavior with this risk, the overall result is much more careful. It continues to enjoy all the benefits of the aggressive approach, but is able to override it when the dice roll could be used to more efficiently safeguard progress than an aggressive strategy would offer new progress.

Through the above considerations, we find that each of the differences observed in the hybrid algorithm on comparing with the aggressive approach acts as an absolute positive, this explains why the approach performs significantly better than a purely aggressive one.

12.7 Comparing Hybrid with Defensive

We will now demonstrate that our proposed hybrid algorithm is more optimal than the defensive algorithm in general cases. To do this, we will compare the behavior of the defensive algorithm with the risk function of the hybrid strategy. The defensive algorithm prioritizes defense at all opportunities, runs away with more valuable pawns when there are no opportunities for defense, and defaults to the fast algorithm.

Similar to the defensive algorithm, the hybrid algorithm's risk function assesses each position based on its value in terms of progress made and places a strong emphasis on defense by evaluating pawn value and the probability of a pawn's successful elimination.

However, there are several key differences between the defensive and hybrid risk gameplay styles. Firstly, the hybrid risk function looks only six tiles behind, in contrast to the defensive strategy, which considers any distance behind necessary to find the first previous opponent pawn. The hybrid strategy also accounts for all pawns within the six-tile window, as opposed to only the first previous pawn. Secondly, the hybrid risk function employs safeSpot as a boolean, such that it would never suggest leaving that safe spot. In contrast, the defensive algorithm does not impose such restrictions, allowing the pawn to leave once the overall probability of death remains constant.

We argue that each of these differences enhances the hybrid algorithm over a purely defensive one. Regarding the first difference, limiting the pawn's vision to six squares behind avoids distraction from more immediate concerns and the need for overall progress. The probability of an opponent pawn pursuing a chase significantly decreases as distance increases, which is consistent with our previous p_kill justification.

Additionally, considering all pawns within the six-tile window is a dominant strategy compared to only evaluating the first one. This allows the hybrid algorithm to incorporate all information considered by the defensive method while also accounting for other relevant factors, such as the presence of other pawns. Consequently, the hybrid strategy can choose to protect a more vulnerable pawn when necessary. This strategy would allow the hybrid algorithm to evaluate the risk on the pawn pieces in cases where more than one opponent pawn is attacking because each of these opponent pawn would independently have a $1/6$ chance of killing!!

Lastly, the most noticeable difference is the hybrid strategy's incorporation of reward. By considering rewards, the hybrid approach can attack nearby enemy pawns, particularly those with higher value. This becomes crucial when deciding between continuing a challenging, low-value defense or pursuing a high-value opponent's pawn. The reward calculations provide an estimation of the expected gain associated with each potential choice. Balancing the defensive risk function's behavior with reward results in a more aggressive yet cautious strategy. This method retains all the benefits of the defensive approach but can override it when the dice roll could more efficiently hinder an opponent's chances of winning by sabotaging their progress rather than merely preserving one's own progress.

In conclusion, each difference between the hybrid algorithm and the defensive approach constitutes an absolute advantage, explaining why the hybrid algorithm significantly outperforms a purely defensive one. By combining elements of both defensive and aggressive strategies, the hybrid algorithm offers a more efficient and optimal solution in various situations.

13 Conclusion

The proposed work provides algorithmic solutions to win the game of ludo optimally. For the sake of the class project, we adopt a slightly simplified version of the game that eradicates the restriction of needing a six on a die roll for a player to leave their start base. We also generalize the home stretch to be the home base, where the pawn is assumed to have completed the journey as soon as it enters the home stretch. However, even with these dissimilarities, several of the characteristics of our algorithm are still applicable and can be helpful in the tactics used in a complete Ludo game.

We talk in detail about a series of initially simple algorithms while explaining their correctness and accuracy, like the fast, aggressive, and defensive algorithms. Despite proving the correctness of these approaches and establishing their superior performance over the random algorithms, we firmly believe that defining and utilizing reward and risk functions, given the state of the board, makes the next move to enhance our approach optimally. We thus showcase our proposed hybrid algorithms and indicate their overall superiority.

We first showed the Fast strategy with its ability to move quickly across the board, reducing the overall risk of getting killed, stating that the Fast algorithm is at least four times better than the random algorithm.

We next show that an aggressive strategy is more optimal than a Fast strategy. This section of the project explores two sub-strategies of Chase and Strike. The optimality of the approaches is first compared amongst each other and then with the fast strategy. The Aggressive strategy combines the best features of the Strike and Chase strategies, making it more optimal than Fast alone.

We next talk about the defensive strategy, which chooses the path that leads to the most negligible risk value possible, where risk can be quantified as the potential of losing the progress made by the player's pawn. By strategically choosing the moves that reduce the overall risk of the player, we ensure that at any given state of the game, the action chosen would be equal to the least risk possible for that state of the game for that player. The algorithm defaults to fast if defending is no longer the viable option.

Realizing the limitations of the strategies above, we propose the hybrid algorithm that factors both risks and rewards to combine the best aspects of the aggressive and defensive algorithms. The improvements made over the aggressive approach include a limited vision range, considering all pawns within the range, and the ability to override aggressive behavior when more advantageous options are available. Similarly, the hybrid strategy betters the defensive algorithm by including a limited vision range, considering all pawns within the scope and the ability to override defensive behavior when more advantageous options are available. By incorporating the best components of aggressive and defensive behaviors, the hybrid strategy proves to be more optimal than the "generic" strategies. Thus with the help of our proofs, we establish that the augmented hybrid strategy is a dominant strategy over purely aggressive or defensive algorithms.

To conclude that we discuss a series of approaches that can serve as a valuable tool for players to optimize their strategic decision-making. By using our algorithms as a starting point and adjusting them to fit their specific situations, Ludo players can increase their chances of winning and outwitting the opponent.

References

[1] Wikipedia. Ludo — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Ludo&oldid=1140972924`, 2023. [Online; accessed 15-March-2023].

[2] Arpit Bhatia, Aneesha Lakra, Rakshita Anand, and Grace Eden. An analysis of ludo board game play on smartphones. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in*

Computing Systems, pages 1–6, 2021.

[3] Alex de Voogt, Nathan Epstein, and Rachel Sherman-Presser. The role of the dice in board games history. *Board Game Studies Journal*, page 1.

[4] Faisal Alvi and Moataz Ahmed. Complexity analysis and playing strategies for ludo and its variant race games. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 134–141. IEEE, 2011.