COMS W 4444

PROGRAMMING AND PROBLEM SOLVING

Project 1: Flip

Submitted By :
Jingya Bi, Ethan Dunn,
John Daciuk

September 30, 2019

Contents

1 Introduction

Flip is a two player game, bounded in a rectangular region roughly resembling a football field. Each player starts with n coins in their own end-zone, and the goal is to move those coins into the opponent's end-zone before they can do the same. If time expires, the player with the most pieces in enemy territory wins. Pieces are moved by flipping such that the distance they can travel in one move is fixed by the piece diameter, but the angle is constrained only by nearby occupied space.[1]

There are a few features of Flip worth noting from the outset because they cannot be ignored by any reasonable strategy:

- Flip is a zero-sum game insomuch as it is only the difference in score that determines victory. Stopping one's opponent from scoring is equivalent to scoring for oneself. For this reason, we take defense as seriously as offense.
- Coordination of pieces is fundamental to success in all cases where $n > 1$. Even when a player has only 2 pieces, the decision of which piece to move may depend on how the other piece is situated. The need for coordination is only compounded for larger values of n, especially when both defense and offense are viable.
- Although in one move a piece's options all lie in a 1-dimensional space on the circumference of a circle, in two moves a piece can theoretically move to any point in nearby 2-dimensional space bounded in some circle. This fact, combined with the fact that each player has two moves per turn [2], greatly enhances the precision with which a skilled player can bring to sets of moves.
- With 11 pieces a player is able to create a vertical wall at some x-coordinate, which, if done correctly, the opponent is unable to pass.
- The possibility of scoring will inevitably disappear, either because the opponent has already scored with all their pieces or because they have built an impregnable wall with no thoughts of dissolution. For this reason, moves are of the essence in Flip, and we must often consider the efficiency with which we operate.

[1] See full description here: http://www.cs.columbia.edu/ kar/4444f19/node17.html
[2] Except for the first player on the first turn.

2 Strategies and Counter-Strategies

In the course of developing our Flip Player, we noticed a common theme of strategies and counter-strategies. It seemed that, for every strategy proposed in class, there existed a counter-strategy. When our group proposed constructing a wall, another group prioritized sending an offensive piece to score before the wall construction completed. When our group prioritized blocking the most aggressive offensive piece first, another group identified our wall structure and sent their offensive piece toward the region not yet constructed. As mentioned earlier, we could now develop strategies to confuse and disrupt this opponent's strategy. It seems that there is never a single "best" strategy, only a combination of good strategies that happen to be combined more efficiently than an opponent's strategies. The game of Flip, while relatively simple in definition, requires a very sophisticated player to perform well in all situations. In the few weeks we had to develop an efficient and robust player, we still realize that any opponent with access to our logic could easily exploit a weakness - though we could then exploit theirs. We believe it is this element of Flip that makes the game so interesting.

3 Strategy Overview

Here we summarize our player's strategy. We leave many of the details as their own sections in the report.

3.1 $n < 12$

For $n < 12$ we do not build a wall. Clearly if $n < 11$ we do not build a wall because we *cannot* build a wall, but what about $n = 11$? Then we can build a wall, but we do not have an extra piece with which to also score, making it impossible to win with this strategy. If our opponent decides to build a wall, we will likely be able to score before completion, but even if we cannot, we stand to only gain by bringing our pieces to their wall, ready to score if it should break. Thus, if $n < 12$, we abandon global defense and set our sights on offense. Nevertheless, we still consider a more localized defense. Our strategy here is guided by what we consider the most important metric contributing to our probability of winning: center of mass in the x-coordinate. We play moves to maximize the difference between our opponents average distance from their goal and our own. To do this, we move straight when we can, and when locked on opponent pieces, weigh our own advantage to moving around against the opponent's advantage of being free to then move straight if we do.

3.2 $n \geq 12$

A solid wall is now possible, and, assuming our opponent will build a wall, we only have the opportunity to score with a few pieces before they complete the wall. We sacrifice little by eventually building our own wall with the remaining pieces. If our opponent is foolish enough to not build a wall, by building ours we blunt the majority of their attack. The interesting consideration is *when* to send offense pieces and *how many* of them to send, because in the beginning of the game both building a wall and capitalizing on a fleeting opportunity for offense are competing demands we must balance.

On one hand, if wall building took an infinite number of turns, we should send all our pieces as offense in high hopes of scoring. At the other extreme, if wall building was assured in one fewer turn than it would take our opponent to score, we may not want to risk any moves on offense until our wall is secured and play for a draw. This assumes our opponent is at least as skilled as we, which should be our focus.

In practice, if $12 \leq n \leq 24$, we send 2 runners for glory before constructing fortifications, else we send only 1 runner first. At around $n \geq 45$ we would have preferred to not send any runners first; however, a bug in our code did not allow us to flip

offense pieces over our own wall in time for submission. The mathematics of this decision making will be detailed in the relevant sections.

Our offense strategy is, naturally, to attempt to slip through a crack in the opponent's partially constructed wall. At location 0 (in the middle of the playing space), our runner decides on an angle of attack based on density of opponent pieces. If our runner hits a wall, we have given it enough intelligence to crawl parallel to the wall, ready to capitalize on faults in construction. See the Implementation section for more detail as to how we coded this useful behavior.

4 Walls

In the course of strategizing for the game of Flip, the concept of a "wall" quickly became a central focus. After all, while an offensive strategy depends on the opponent's defense, a defensive strategy can be reasonably (though not completely) perfected.

4.1 Wall Specification

A wall is a number of pieces which, when combined, prevent any opponent piece from passing it. A player can guarantee a win (or tie) if they have scored more (or equal) pieces, and constructed a wall behind the opponent's next offensive piece. Once the wall is built, the player only needs to maintain the wall, and could even score additional pieces (see Wall Offense). Since the goals are in the horizontal directions, the wall should be constructed vertically to prevent horizontal progress.

4.2 Wall Separation

Logically, we want to maximize the separation between any two pieces of our wall and still prevent an opponent from moving their piece between our own pieces. The fewer pieces we need, the fewer turns it takes to construct the wall, and the more pieces we have for offense. Geometrically, the farthest apart our pieces can be is $2\sqrt{3} \approx 3.46$. There is a special edge case with the top or bottom boundary line, in which our piece may be $1 + \sqrt{3} \approx 2.73$ distance away without allowing an opponent to pass. As a result, and since the y-axis of the board is length 40, it takes 11 pieces at the same x-coordinate with the following y-coordinate values to construct a wall:

$$\pm 17.30, \pm 13.84, \pm 10.38, \pm 6.92, \pm 3.46, 0.00$$

5 Wall Construction

In order to construct a wall, we need to be able to move a piece to a precise location. With no other pieces, it is easy enough to move a piece towards a desired location based on some angle θ, though the next two moves might overshoot the destination. Still ignoring other pieces, it is also possible to move a piece to an exact location that is within 4 units away, because each move translates the coin by 2. There will be two flips, one flip of angle θ and another flip of angle ϕ. Let the current piece be centered at (x_c, y_c) and consider moving it to (x_t, y_t).

$$x_c + 2cos(\theta) + 2cos(\phi) = x_t \implies 2cos(\frac{\theta+\phi}{2})cos(\frac{\theta-\phi}{2}) = \frac{x_t - x_c}{2}$$

$$y_c + 2sin(\theta) + 2sin(\phi) = y_t \implies 2sin(\frac{\theta+\phi}{2})cos(\frac{\theta-\phi}{2}) = \frac{y_t - y_c}{2}$$

Dividing the second equation by the first, we can see that

$$\frac{\theta+\phi}{2} = arctan(\frac{y_t - y_c}{x_t - x_c})$$

Squaring both equations and adding them together, we can see that

$$\frac{\theta-\phi}{2} = arccos((\frac{x_t - x_c}{2})^2 + (\frac{y_t - y_c}{2})^2)/2)$$

At this point it is trivial to solve for θ and ϕ.

In practice, constructing a wall is not this simple, because our own or our opponent's pieces may be in the way. When this occurs, we fall back to a best effort attempt using other methods described in this paper. In a more general setting, handling obstacles, even if we control the pieces, has proven challenging.

5.1 Handling Wall Interference

A wall interference refers to when an opponent's offensive piece occupies space that we require for our wall. Dealing with wall interference is not incredibly difficult, but it is quite tedious. For this reason, coupled with the low likelihood an opponent intentionally identifies and obstructs our wall, we did not implement a solution, though one definitely exists.

There are two ways to handle wall interference. The first is to leave our own piece tangent to its desired location and the obstructing piece. Then, every turn, if it is possible to occupy the desired location, do so. This strategy is susceptible to the opponent creating a chain of pieces, so that they may permanently occupy the

obstructing space after every turn of two moves. The second method is to position another piece such that the opponent's piece cannot move forward to score, nor can any other opponent piece pass the wall due to the asymmetries created by the first obstruction. This strategy is more challenging to implement, and we did not have time to devote to its full solution. However, we are confident that one exists, and most likely requires only one more piece placement.
Whichever countermeasure one could implement for wall obstructions, the most important part is to identify when a wall cannot be completed, and to execute offensive moves when there are no more beneficial defensive moves.

5.2 Wall Prioritization

Once we are able to construct the wall, we can prioritize the construction of our wall to prevent the opponent from scoring. We implemented a simple heuristic of prioritizing wall construction locations that are not yet complete and closest to the opponent's most forward piece.
Unfortunately, even this prioritization cannot prevent an intelligent opponent from scoring. In fact, we faced an opponent who specifically located wall constructions, identified by multiple pieces at the same x-coordinate. This opponent then identified gaps in the wall, and sent an offensive piece towards the gap. Even though our simple heuristic for wall prioritization is superior to none, there is are many game parameters N for number of pieces for which the wall can be constructed, but cannot be constructed in time.
Of course, there are counter-strategies to this particular opponent's strategy. If the wall is not uniform, it becomes difficult to detect. If the wall is constructed in such a way to lure the opponent's piece in one direction, only to fill the gap before it reaches the goal, the opponent would spend many moves. In addition, an offensive strategy that makes it appear a wall exists but instead attempts to score could perform better due to the opponent's wasted moves circumnavigating the apparent wall. In either case, wall prioritization ultimately accounts for various strategies which identify and exploit wall gaps, though not perfectly. We recognize this opponent's strategy for countering our own wall.

5.3 Wall Offense

With $n > 11$ Flip permits not only perfect wall formations, but also the ability to launch offense from behind the wall with zero defensive compromise. With two moves per turn, we could move a wall piece forward into offense and exactly replace the gap with another piece behind. In many cases, this would not increase our score as these new offensive pieces have no hope if the opponent has a perfect wall. However, in cases when the other side has cracks in their defense, perhaps because of

our offensive harassment, being able to bring in more pieces to score could be crucial. One particularly tantalizing thought is the possibility of having a snake of offensive pieces gliding through a crack in opponent defense, never giving the opponent the opportunity to cut the snake and secure their border.

Alas, although we had the game state mechanics and functions ready to implement wall flipping, we did not debug the code in time for the tournament.

5.4 Wall Construction Performance

Naively, the most efficient way to construct a wall would be to iterate through the desired destinations and assign it to the nearest piece. Conversely, one could also iterate through the front-most pieces, and assign it to the nearest destination. However, neither of these options are optimal. Moreover, these solutions will often end up with crossing paths, which can create congestion while moving pieces to their destination. Any crossing path can be seen as an inefficiency in itself - it would make more sense to swap the destinations of the pieces whose paths would cross, decreasing the total distance that needs to be covered.
A far better solution is to solve the Minimum Weight Bipartite Matching (MWBM) problem. Given two sets of nodes of equal size, and at least one edge connecting each node in one set to the other, the solution will find a matching between the sets that minimizes the sum of the edge weights. If we assign the edge weights to be the distance between each piece and its potential destination, the MWBM will provide the solution that involves moving pieces the shortest distance, which is a good proxy for minimizing the number of moves (and therefore turns) it takes to construct the wall. A beneficial side effect is that pieces are less likely to cross paths with one another. However, since the input requires us to specify which pieces are part of the wall, we can only optimize the wall construction for the pieces we have selected; meanwhile, there are $\binom{n}{11}$ choices of pieces which can form the wall (though this could be reduced through geometric considerations). In retrospect, a better method to select pieces might be those selected by an ellipse with focii centered on the desired x-coordinate for the wall and ± 10 on the y-coordinate.

6 Runner Strategy

For $n \geq 12$, we can afford to build a wall and send extra pieces to attempt scoring; however, the timing is crucial. Ideally, the general process is as the following:

1. Send runners to disrupt opponent's wall building. The number of runners we send is a function of n as will be discussed in section 7.2 Number of Initial Runners.
2. Build the wall.
3. Send as many runners as we can by replacing wall pieces with pieces further back in our own territory. As long as we have a perfect wall, we can send every extra piece into the offensive front. [3]

6.1 Initial Runners to Disrupt Wall Building

At this stage, our runner strategy is to send one piece at a time (i.e., devote both steps we have on each turn to this one piece). We select the farthest forward piece to race forward. This piece will first go straight ($\theta = 0$) to the middle of the board, and then aim for the least density area by moving with an angle. We choose to let the piece aim at an angle after passing the middle of the board because the opponent's density can quickly shift. If we compute the density at the very beginning and aim for the least density, the density might increase tremendously as we get close. Meanwhile, if we update the angle at every move, we are wasting moves in the case of an angle change. Therefore, we decide that we first move straight, as there is unlikely to be a lot of pieces in the way at this point, and then resolutely aim at the least density area. This strategy works fairly well as we can tell from the tournament result. In this process, we employ two useful functions `getAroundWithPreferredAngle()` and `ForceMove()`, which helped us to implement an effective wall crawling routine. Detailed explanation of these functions is in Implementation section.

However, there are cases when a runner can not get into the goal area either because the opponent has already built a perfect wall or our runner is not intelligent enough to navigate around all obstacles. We will give up on the first piece and send the second one if the piece has moved for 100 steps. We chose 100 because this is the expected number of steps that a piece goes to the goal line and moves either to the left vertical border line or to the right border line (reach min or max y). Our argument is that if a piece cannot find a hole on its way to the left or right border line, this piece has to retrace back to where it starts close to the center goal line to

[3] We did not managed to work the bugs out of this last component, as we will reiterate.

explore the other side, and by then our opponent is likely to complete their wall. Discussion about how we choose the number of initial runners as a function of n is in section 7.2.

6.2 Runners After Wall Building

After we finish our own wall building, we can afford to send more runners. As we observed in games against other groups, most groups can not build a perfect wall if the wall is interrupted by some opponent piece. Other groups tend to leave a hole in the wall and the hole is unlikely to be filled up later. Therefore, we can send more runners through! Unfortunately, we did not get the function to release more runner from behind the wall running, so we could not implement this part even though our state machine was prepared. However, it is worth mentioning that if we did get this element to work, we would set the step limit higher than 100 steps for these future runners, because we now do not have the risk of opponent pieces passing through our wall. We can thus give a piece more time to find a hole in the opponent's wall.

6.3 Alternative Ideas

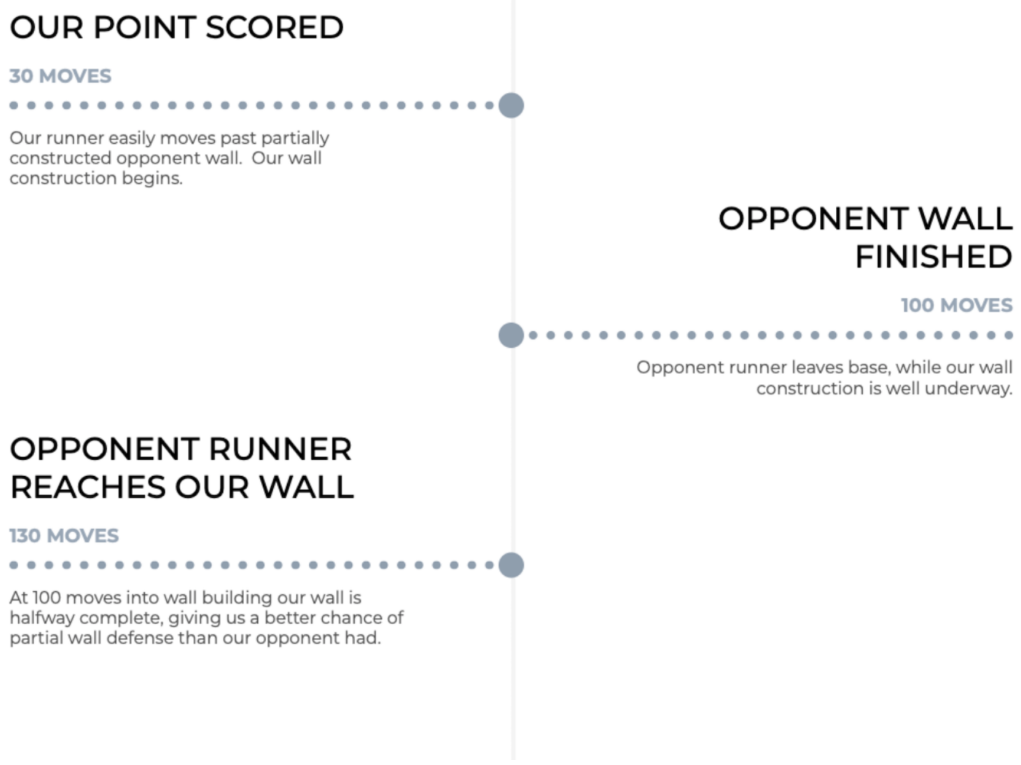
Our group discussed various runner strategies to get through the wall. One was to do path finding by Breadth First Search. This idea was abandoned because our current runner strategy is easier to implement, though not optimal. We also considered recording a piece's state, i.e., where it has been for the past few steps. This is useful if the piece is trapped in a local area. If we discover that in the past 10 to 20 steps this piece has not moved much, then we can force it to back track and move to either up or down for some consecutive number of steps along the opponent wall without determining whether it can move forward through a breach in the wall every step. This idea would potentially waste moves, but it should work well when there is hardly any way forward. We did not implement this idea due to time constraint, but we believe this is a viable option.

7 Large N ($N > 12$)

As discussed in the Strategy Overview section, we intend to send some number of offense pieces and then build our wall. We begin by considering why we do not simply build our wall first, and then move on to the mathematics of how many runners we initially send.

7.1 The Argument for Sending Runners First

Figure 1: Progression of simplified Flip game in which we send 1 runner first but the opponent plays defense before sending runners. It should be clear that our runner has better offensive odds than theirs. Assume $n \approx 12$.



Our first intuition was to build our wall first before sending runners. The idea of first locking down defense and then being free to play offense seemed safe and attractive. However, by considering the timeline above we note an obvious way to exploit this strategy by sending runners first. To summarize the diagram, where we assume $n \approx 12$ for simplicity, if our opponent is first fortifying defense while we send

a runner, our runner will more easily pass through their wall than they will ours.

Now, of course our opponent could have stopped playing wall moves once our runner scored and started playing offense. In that case they may have the same chance of scoring that we had. However, such a dynamic strategy would introduce complications. First and foremost they would a very good estimate of when our offense strike is over, e.g. we could be sending more than 1 runner. Second, we could try to exploit this by harassing their wall building, while also constructing our own.

It seems likely that a skilled player could play 1 runner move that would cause the opponent to have to respond with 2 extra defense moves; in such a way that player could use their extra moves to gain tempo for their own defense. In other words, the best Flip players might first use runners to disrupt the opponent's wall and then create their own wall while that opponent makes repairs. After all, one well-placed offense piece could cause all 11 pieces of the wall to move, or else require an extra defense piece to build around the one offense piece. For values of n that make every move count in the race to get 1 extra offense piece to the end-zone, we believe that this advantage makes for very rich strategical play and exciting games. In any case, we did not have time to explore these dynamics in depth, but these insights gave us the intuition and inclination to play offense before defense.

7.2 Number of Initial Runners Before Wall Construction

Let us work out the number of initial runners we will send as a function of n. In the process we will make a number of simplifying assumptions. Because the fates of our runners and defense line are partly determined by the play of the opponent, we expect uncertainty and only hope to get a rough idea from the following calculations.

Let us begin by assuming $n = 12$, which is a simple and concrete case. We ask the question, if all our opponent does is build a wall, can we hope to get a runner in first? The answer is yes, we likely can, and this is why:

Our end-zone has a width of 40 and our pieces spawn uniformly distributed. With 12 draws from $uniform(0, 40)$ what is the largest number we expect to draw? This is equivalent to asking how far forward our best runner will be. The probability that all draws are less than $x = P(\forall\ draws < x) = (\frac{x}{40})^{12}$, which means that the probability of at least 1 draw being greater than x is $1-(\frac{x}{40})^{12}$. So suppose by expect to draw, we mean with probability greater than $\frac{1}{2}$, then we have $(1 - \frac{x}{40})^{12} = \frac{1}{2}$ and $x = (\frac{1}{2})^{12}(40) \approx 37.75$. So at 2.25 away from neutral territory, and from there 41 away from scoring, our runner has a distance of 43.25 to travel if unobstructed. How much distance does the opponent have to travel to build their wall? We can continue with the order statistics for the next 11 pieces and be very precise about

probabilities, but let us think in terms of rough expectation. Setting aside the fact that we have already grabbed the 1st order statistic to be the runner, we expect the 11 wall builders to be in the middle of the end-zone [4], which means they each have a distance of 19 to travel for optimal location building. Thus, the opponent has to travel $(11)(19) = 209$ to have the wall complete before we get there: it will not happen.

Given the margin our runner has from this initial calculation, it is easy to see that we could even get 2 runners in if we had another available (we do not). So $n = 12$, we send the 1 runner we have [5] and for $n = 13$ we send 2. Let us calculate whether we can send our 3 available runners if $n = 14$[6]:

Split the two end-zones now into 14 vertical strips and temporarily put both in a coordinate system with x ranging from 0 to 40. We expect to have runners at the centers of the furthest forward 3 strips, putting them at distances $\frac{40}{28}$, $\frac{40}{28} + \frac{40}{14}$ and $\frac{40}{28} + 2\left(\frac{40}{14}\right)$ from neutral territory. All told those runners would have to travel on average $4.3 + 41$ to score if unobstructed, making for a total distance of about 136. But of course now the opponent's wall is presumably building faster (less than 209 moves) with closer pieces to choose from to do the building. After also accounting for each of our runners needing to run along the partial wall for some distance between 5 and 20 before scoring, we believed $n = 14$ with 3 runners might have been pushing our luck [7].

If we cannot manage 3 runners at $n = 14$, things will only be harder for $n > 14$. So we use 2 initial runners up to some value of n, at which point the opponent's wall builds so fast (all 11 wall pieces are getting closer to their destination as we increase n), that we revert to only sending 1 initial runner. Where is the cut off n? Suppose our two runners have an equal combined distance to travel as our opponent's 11 wall builders. We model cumulative distances as a function of n using similar logic as above and get the following equation:

$$2\left(\frac{\frac{40}{2n} + \frac{40}{n}}{2} + 41\right) = \frac{(11)(11)(39)}{2n}$$

Where the left hand side is the average runner distance multiplied by 2 and the right hand side is the average wall builder distance multiplied by 11. Solving this

[4] Actually, we assume the opponent to use the 11 out of 12 pieces they have furthest forward for wall construction, and their distance traveled will be slightly less than we compute.

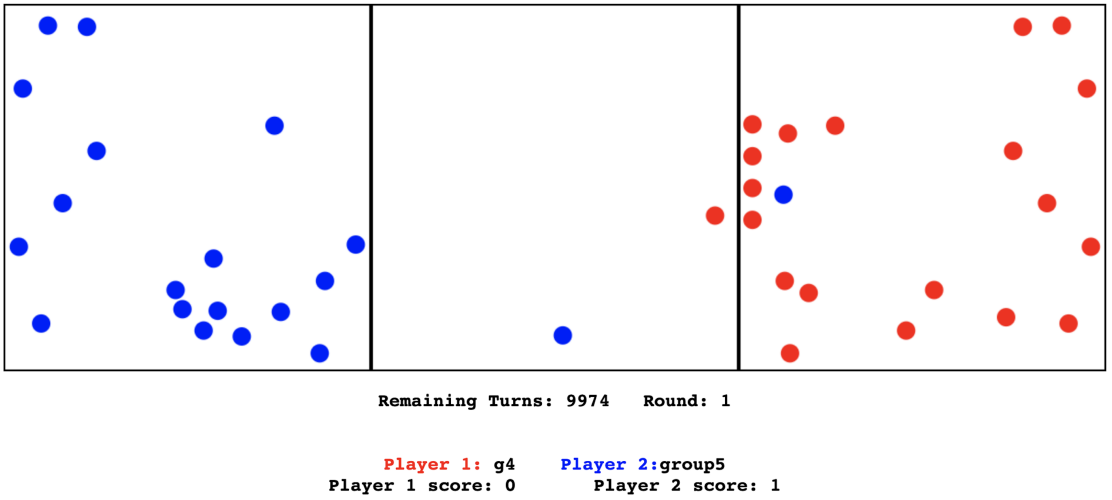
[5] In practice our code may still have tried to send 2, and would be left in an infinite loop unable to find a second available runner

[6] Remember that at $n = 13$ we did not have 3 runners because we need 11 for the wall.

[7] With better offensive path finding we probably could have pulled it off

equation we have $n \approx 28$ as the border line case. In practice we must add in the consideration that runners will likely waste moves searching for an opening and wall builders will waste moves getting blocked by runners and triangulating to precise locations. Since this all muddies the calculations, we choose $n = 25$ as the cut-off after experimentation.

Figure 2: Playing our nemesis g4 with $n = 20$. We send out two runners first and win the tournament in this domain.



So in conclusion, for $11 < n < 25$, we send 2 runners thirsty for glory before playing any defense moves; for $n \geq 25$ we only send 1 runner. If we had the kinks worked out of our code such that we could flip runners over our own wall, we would have stopped sending even 1 runner at $n \approx 55$. At this point by building a wall first, we cannot lose [8], and would prefer to reduce the probability of losing to zero before sending offensive waves.

[8] Of course all the pieces could be randomly placed all the way back, but, practically speaking, that will not happen.

8 Small N ($N <= 12$)

Clearly we would like to decrease the sum distance our pieces are from the goal, so we move pieces straight if possible. But what if we do not have a straight move available? This is where things get more interesting, and it was dealing with this situation that was the majority of coding work for small n [9]. Let us begin with theoretical considerations for $n = 1$ before further discussing our general strategy for small n. It is instructive to think through whether or not to move around the opponents single piece if stuck because we must deal with this situation also for higher values of n.

First, let us convince ourselves that we may be willing to give up two turns if head to head (tangent) with the enemy. When the two pieces meet, either player 1 or player 2 just moved, suppose it to be player 1. Now if player 1 just moved straight twice, player 1 would be exactly one turn ahead of player 2. In this extreme scenario, if player 2 now moves around player 1 at a 60° angle (optimal in the tangent scenario), player 1 makes forward progress equivalent to 1.5 straight moves [10]. At this point, player 2 is ahead, and will continue to be ahead immediately after every turn. But now, player 1 will also gain the lead immediately after each of *their* turns. Who will win depends on parity and the precise distance remaining to be traveled. If player 2 has calculated that they cannot win, what do they gain by breaking the dead lock? Although then player 1 could simply break the dead lock and then be ahead by 2.5, which is a definite win, but perhaps player 1 does not see this!

It is interesting to consider if a dead lock is possible in which it is to neither player's benefit to move around: the answer is yes. Consider that the two players meet tangent after player 1 was only able to make 1 of their two moves. Now the two players would be at exactly the same distance from their respective goals. If player 2 does not calculate a win by moving tangent, then by symmetry neither can player 1!

But now consider some complications to the above. When two pieces meet, they almost certainly will not meet tangent. Rather they will be some other distance away from each other that is non-zero yet still prevents them from flipping straight. In that situation the player who moves around losses less ground by doing so. In fact, if a player were to decide to move around an opponent, that player would actually be best off doing it from a slight angle at the start of the game because the shortest distance between any two points is a straight line.

These mechanics all be deterministic, to play optimally all we need to do is calculate

[9] To the extent that we probably got too hung up on it, especially for $n = 1$

[10] Simple special right triangle trig.

everything from the beginning! But of course it is not easy to work out or program quickly, especially when $n > 1$. Thus, after discussing the richness of even a $n = 1$ Flip game, we propose the heuristic we used for our player: for any $n < 12$, we are willing to move around opponent pieces if we have two moves that we would otherwise waste, else, we do not move around the opponent.

So, to get our bearings again, we are moving our pieces straight, unless they are stuck, at which point we have a few options. We seek to take these options in the following order:

1. Move around our own pieces
2. Move pieces in the end-zone forward if they block entry of our own pieces.
3. Move around opponent pieces if we have two moves available.

Finally, we'll address two more points to complete our $n < 12$ strategy:

- If we ever have a center of mass advantage greater than 1, we are willing to move around the opponent even if we have only 1 move left in the turn.
- When moving straight, we move the pieces furthest forward first. By doing this we have a shot at blocking multiple opponent pieces still at home with just 1 of our pieces that has already scored. Those pieces are now forced to move around us and hopefully themselves if they are bunched up enough. In practice this can be highly effective when n is close to 12 and we are playing a player who always moves their furthest piece back. In fact, for $n = 11$ we beat opponents on average by about 4 pieces [11].

[11] See chart of mean opponent scores.

9 Implementation - our code structure

9.1 Essential Functions

When coding our Flip player, we soon came to realize that in many circumstances and strategies the same set of functions would come into play. What follows are the functions that became the work horses of our program:

1. `getSingleMove(pieceId, theta)`
   This method takes in the pieceId one wants to move, and the angle θ. Note that to be more consistent, positive θ is always moving right, and negative θ is always moving to the left from the perspective of the player. This method will return a move that moves in the specified angle. We later had a modified version of this function, `getSingleValidMove(pieceId,theta)` which returns a valid move after checking validity.

2. `getAroundWithPreferredAngle(pieceId, theta)`
   This method returns a move to help the piece move around obstacles with a preferred angle. Detailed explanation can be found in the next section Move Around Tangent.

3. `forceMove(pieceId, theta)`
   This method is our fallback option when `getAroundWithPreferredAngle` can not find a valid move. It essentially searches for all angles possible, from a pre defined θ. It first tries to move with angle θ. If it fails, then it would to try widen the angle by 1° at a time.

4. `moveCurrentToTarget(pieceId, target)`
   This method has a few cases and several fallback options.

   (a) If the piece is far away from its target, this method returns moves to get the piece as close as possible - if there are any obstacles in the way, it attempts to navigate around them efficiently. In the worst case, when there are multiple obstacles with no clear solution, the piece will move at an angle that is the smallest deviation from the preferred direction, computed by brute force.

   (b) If the piece is within 2 moves from the target, this method tries to return two moves that move this piece exactly to its destination. If the first move is invalid, it switches the order of angle movements and tests that sequence. If there is still an obstacle in the way, this method tries to move the piece directly towards its target, even though it will still require two more moves to reach it. If that attempt is invalid, the method then attempts to navigate tangent to the obstacle in the closest direction to the

preferred direction. And if all else fails, the method returns a forced move described above in an attempt to position the piece for a new attempt next turn.

(c) If the piece is within 3 moves from the target but more than 2 moves away, this method will return the first move from the first part and the second move from the second part described above.

9.2 Move Around Tangent

When we face an opponent and we wish to get around, the optimal way is to first move tangent to the opponent while having the least off-course angle, and then move forward.

Our `GetAroundWithPreferredAngle` function first tries to move at the preferred angle, and if it fails, it checks for collisions in the way, and then moves tangent to the *closest opponent*. Note that there are two solutions to moving tangent. We first try the solution closest to our preferred angle.

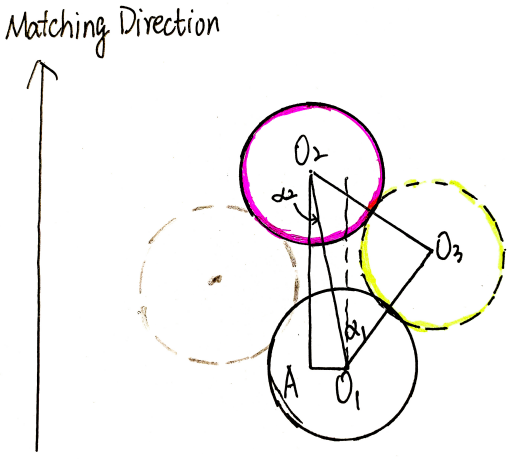


Figure 3: Geometry of flipping tangent. The black solid line circle is our piece; the pink solid line an opponent piece; the yellow dash line where we should flip; the pencil dash line circle is another choice with bigger off-course angle.

$$\begin{aligned}
\alpha_1 &= \angle O_2 O_1 O_3 \\
\alpha_2 &= \angle A O_2 A_1 \\
m_1 &= \overline{AO_1} \\
m_2 &= \overline{AO_2} \\
m_3 &= \overline{O_2 O_1}
\end{aligned}$$

When there is not enough space to move forward, the angle we need is $\alpha_1 - \alpha_2$. Note that a piece can move tangent to the other side, which is $\alpha_1 + \alpha_2$, but we choose to try $\alpha_1 - \alpha_2$ first because this would give a smaller off-course angle. The optimal angle can be easily deduced from the calculation below.

$$\begin{aligned}
m_3 &= \sqrt{m_1^2 + m_2^2} \\
cos(\alpha_2) &= \frac{m_3^2 + m_2^2 - m_1^2}{2 m_2 m_3} \\
cos(\alpha_1) &= \frac{2^2 + m_3^2 - 2^2}{2 \cdot 2 \cdot m3} = \frac{m_3}{4} \\
\alpha_1 - \alpha_2 &= cos^{-1}(\alpha_1) - cos^{-1}(\alpha_2)
\end{aligned}$$

Note that when the marching direction is not straight ahead (i.e, $\theta \neq 0$), the above calculation can still be used by simply adding the current moving angle.

9.3 Crawling Along the Wall

Crawling along the wall is an important feature when we play offense while our opponent has yet to complete a perfect wall.

We find that `getAroundWithPreferredAngle` is not sufficient in completing the task of finding a path through the wall. When there are a lot of opponents ahead, getting around becomes difficult. Our piece would get stuck if opponents are close to each other, especially when they form a half circle or a straight line (wall) with small gaps, because our piece will tend to engage in an infinite loop of contradictory moves since it only considers the closest opponent. Worse, the function might send invalid moves when our piece collides with another opponent piece while moving tangent to its closest opponent!

Therefore, we have to switch to our `forceMove` function. We predict that for $n \geq 12$, opponents are likely to build a wall, so when our piece is within distance 5 of

the scoring line, we switch to the function forceMove. As described earlier, if the preferred angle is 30°, forceMove will try $31°, 29°, 32°, 28°, ..., 210°, -150°$. Note that it essentially checks all possibilities around by fanning out by 1° at a time.

We choose the cut-off we did for determining whether to switch to `forceMove` because within three full turns, a piece can score if there is no obstacles ahead. With a closer cut-off, we would have less opportunity to explore (by this point, a wall should be in progress). We also did some empirical experiments that show a distance of 5 to be a reasonable time to switch functions. And recall, even though we switch to `forceMove`, the piece will still move at the preferred angle if possible.

Algorithm 1 Wall Crawling Algorithm

```
Input: preferred_angle θ = 45°, default_angle θ_0 = 0
Initialize: moves = []
while piece not in scoring zone do
  while piece not colliding the boundary (|piece.y| ≤ 18.5) do
    if can move with θ_0 = 0 then
      moves add move
    else
      try forceMove with θ
    end if
  end while
  θ ← −θ
end while
```

By our conventions, preferred angle $\theta = 45°$ essentially means the piece is moving up and right, when it hits the boundary of the board, $\theta = -45°$, means it moves up and left. This algorithm almost ensures that the piece gets in the hole if there is one [12].

An illustration of wall crawling is in appendix of images. The series of images showcase our process of finding our way into a hole (we are blue).

[12] We do believe that holes deliberately constructed could still thwart our deterministic runner, but they are highly unlikely

9.4 Integrating the Components

Patching the pieces of our code together to form a coherent and deliberate player was one of the most challenging parts of Flip for the entire class. During discussion sections, we watched time and time again a reasonable player break down into a string of incoherent moves the moment game state changed. Perhaps a wall would finish building and then the player would freeze, or a runner would hit a wall and then fall into an infinite loop of contradictory moves, blind to blatant holes in the defense. One of the major themes of Flip turned out to be the coordination of many interdependent mini strategies.

To create a player that could fluidly shift from one mode to another, we coded an enum to hold our game states:

```
private enum STATES [
DEFENSE, OFFENSE, RELEASE-RUNNER; ]
```

Because the mechanics of game state transitions leave a lot of room for error, we choose to reduce the number of states down to just 3. When the simulator calls `getMoves()` and we are in `OFFENSE` mode, we return runner moves. If we have not yet assigned a runner, we make sure to pick a runner id first and store it globally. Once all of our runners have expired as discussed earlier, we sign out of runner mode by setting the global `GAMESTATE` to `DEFENSE`. In `DEFENSE` mode, we return the next two wall building moves that make sense, or `null` if none do. After the wall is finished we make sure to set the state to `RELEASE-RUNNER`, which is then intended to put a runner in front of the wall before turning the state back to `OFFENSE`. Ideally `RELEASE-RUNNER` and `OFFENSE` would continue to loop until all pieces from behind the wall are released [13].

To conclude, our state mechanics were successful because we boiled them down to only three, and those three states each had clear goals in almost any position where we had moves worth taking. Well defined goals are important not only for executing during the state, but also for passing over the state when appropriate. To further improve our player, we could refine the goals of the states. For example, perhaps the goal of `DEFENSE` should not be to finish building the wall, but to work on the wall until an offensive exploit is apparent. Perhaps we should not have told the runners to expire after some number of moves, but rather made their active time a function of their opportunities to score.

[13] We did not actually get this part to work

10 Analysis of Tournament Results

10.1 Wins and Losses

At the conclusion of the Flip project, a tournament was run so that all teams could play their strategies against all others. 20 games were played for each pair of teams for each choice of n. Let us begin the discussion of tournament results by first considering the percentage of games we won, neglecting draws for the moment for simplicity.

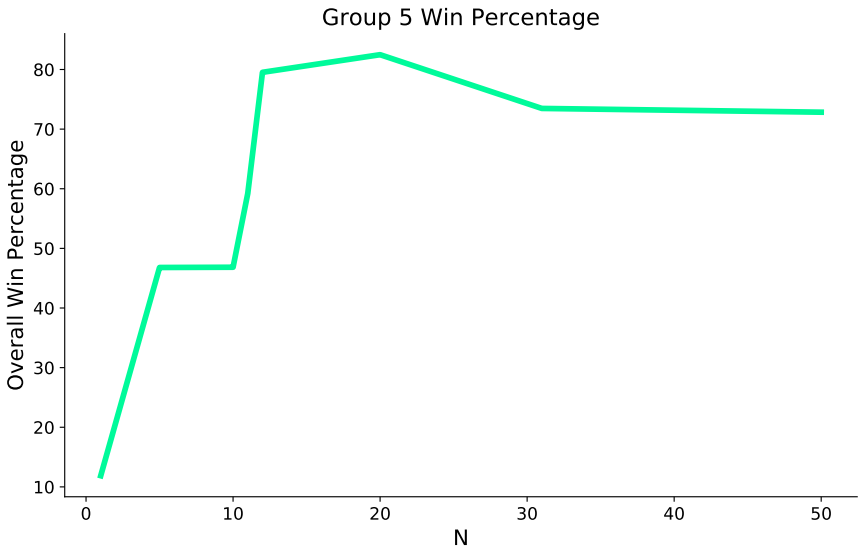


Figure 4: We see solid performance once a wall is feasible.

Once $n > 11$ our player is able to find the sweet spot between defense and offense better than most others. We were not surprised to see the strongest results for $11 < n < 25$: this was the game we were most interested in all along for reasons noted earlier. Recall in this domain we send two runners before building our wall, while some other players were only sending one if at all. Whereas our runners almost blindly slip through the enemy wall by means of the crawl routine, the enemy has a much more difficult programming challenge to smoothly finish their wall. Players who did also send two runners *were* able to aim for holes in our wall, but if they committed to a path too early, we may have filled it in before they arrived by means of wall prioritization [14]. Alternatively, if they continuously updated their trajectory based on the pattern of our construction, they may have been left with a rather

[14] See earlier discussion in the wall section

indirect path to their goal. Coupled with the fact that other players' runners were not able to recover and path find on the fly once they hit our wall, many opponents have a recipe for a losing game [15]. Some of our loses for $11 < n < 25$ were due to our runners giving up too early [16]; however, we also believe that we can credit some wins to our runners not being overly persistent, thereby giving back crucial moves to defense.

Clearly we have the poorest results for $n = 1$: in fact, in this domain we were the worst of all teams. We chalk this up to two reasons: First, we did not specialize our player for $n = 1$, instead we simply played the same strategy we use for $n < 12$. This was not wise because for $n = 1$ deterministic calculations about whether to move around the opponent become easy to do. We are giving up too many games because our player is always content to waste a move rather than move around the opponent [17].

As n goes to 5, 10 and 11 we are not shining, but we are at least holding our ground: now using heuristics is a more reasonable approach since brute calculation is likely not a game anyone endeavored to play [18].

10.2 Full Tournament Score

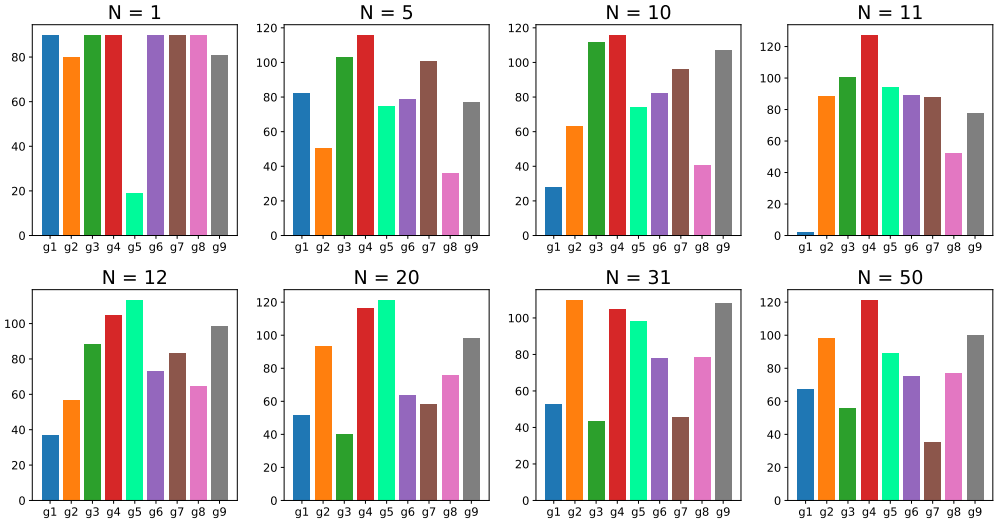


Figure 5: A more detailed look at performance in terms of points.

[15] At least this appeared to be true the morning of code submission
[16] See the Runner Strategy section for more discussion of runner stopping mechanics
[17] Recall we were at least shrewd enough to not waste two moves.
[18] Although if they did, they would have almost certainly won their match against us

We now consider group5's performance with respect to points, where we grant 1 point for a win, .5 for a draw and 0 for a loss. Having also included the performance of all other teams we have a much more detailed look. We managed to win the tournament for $11 < n < 25$, although group 4 was close behind. For $n = 1$ we had the worse performance of all teams. The low variance in scores for $n = 1$ confirms that most teams likely had a very similar strategy of calculating whether to move around the opponent, or simply always moving around. These charts also suggest that we were not the only team to focus on some range for n. For example, groups 3 and 7 were very competitive for $n < 12$, but fell apart once players were able to build walls. This supports our view that not attempting to build a wall for $n > 11$ is a non sequitur.

10.3 Mean Scores Per Game

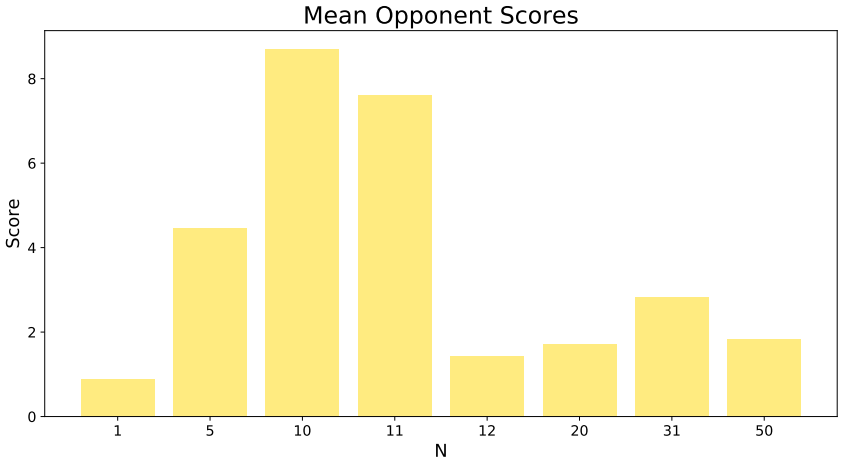


Figure 6: A consideration of scores on a per game level.

It is worth while to briefly look at the number of coins our opponents flipped across our line on a per game basis. Note that this is very different than the tournament scores we discussed by level of granularity because now we can see by how much we won or lost individual games. For example, for $n = 11$ the game tends to end when our opponents have only 7 coins in our end-zone (vs our 11 in theirs). We credit our robust play here to the number of and order of fall back methods we coded for when pieces get stuck; after all, for $n = 11$ we practically have two walls moving in on each other, so pieces *will* get stuck. As discussed in the $n < 12$ section, when our pieces can no longer move forward, we try a number of things before succumbing to moving around the opponent.

When $n = 30$ and $n = 50$, our opponents have mean scores higher than we could possibly have had, since in this domain we are only ever venturing 1 runner. Had we worked the kinks out of flipping pieces over our own wall, we would expect much better results here. Recall that if a player did wind up with a fault in their defense, we could have brought all of our pieces through with the crawl routine, except for the 11 wall pieces, effectively increasing our mean score dramatically.

10.4 Draws

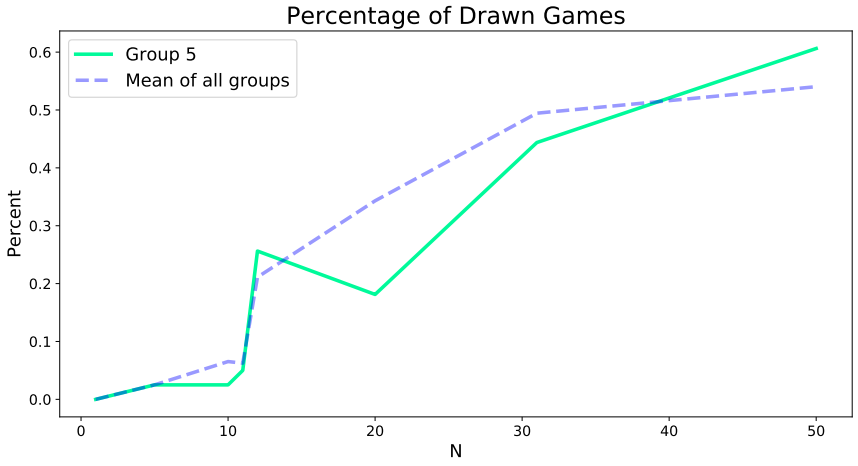


Figure 7: Draws increase with n.

We conclude our tournament remarks by inquiring as to how n affects the tendency of Flip to end in draw. For $n = 1$ almost no games ended in draw, but at the same time entire matches between two players did, with little variance in total performance. For $n = 1$ either player 1 or player 2 had too strong of a starting advantage to make for interesting games [19]. As n permits impenetrable defense, we see many more draws, especially when $n = 50$, and a wall may be able to go up before a runner can make it. We can understand why games were not played with $n > 50$. Our difference with the mean when $n = 20$ is consistent with our analysis in section 10.1.

[19] Well, interesting enough for us since we largely lost, but not interesting to the other players!

11 Lessons Learned

Our group learned several valuable strategies for the next time we encounter a multi-week coding project. To begin with, we should have spent more time identifying and communicating useful functions and concepts to minimize the amount of code we ended up writing. We should have agreed ahead of time about what fallback logic a function had, and what it would return in the event of a catastrophe. This would not only decrease the work required, but it would also improve the fidelity of the code, because more people would be debugging the same methods. We also could have structured the code better by separating out defensive strategies, offensive strategies, and utility methods. This would have simplified the logic controlling our overall strategy. Finally, we should have made immutable copies of the maps specifying player_pieces and opponent_pieces to assist in checking the validity of various move combinations without changing the base copy provided to us in the initialization method. Similarly, we could have stored whether or not we were the first player as a state variable, because it never changes. Despite these lessons, we did still manage to modularize and share a number of methods, take advantage of enums, and successfully debug almost all of the issues we encountered.

11.1 Math Lessons

The range of *arctan* is between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. For us, this corresponded to the player always moving in a "forward" direction. However, when moving a piece to a particular location during wall construction, this caused an issue because the piece would never move backwards. It was easy enough to correct this issue by checking for the proper direction and correcting the range output of *arctan*, but it took many hours to identify the true source of the issue. This issue came up for two reasons - first, the optimal location for the wall is actually almost 2 units behind our own goal line, because the opponent piece needs to be fully inside the end zone to score; second, if a piece could not get to its destination exactly due to an obstacle, we move that piece as best as possible, which could sometimes be slightly in front of its destination.

12 Future Developments

In the case where $5 < n < 11$, it is impossible to construct a wall, but it might be interesting to create a blocking structure that can be propagated forward. This structure would force the opponent to move around the blocking structure, spending valuable moves. One such structure we considered is the flying V, which would force the opponent to move away from the center of the board. The main conflict we encountered with this idea is balancing going around opponent pieces ourselves with leaving pieces to block the opponent's path. Ultimately, we decided that while it could be interesting, it intuitively seems that the moves spent to create the blocking structure would outweigh any benefits that could be found. Furthermore, we found the case where $n \geq 12$ to be more intriguing, and spent the majority of our time on these strategies. Exploring the flying V, or other such blocking structures, is left as a future consideration.

Another mechanism we considered implementing is minimizing the sum of the shortest paths for our own pieces and maximizing the sum of the shortest paths for our opponent's pieces. Optimizing this objective is clearly beneficial, but we struggled to define the shortest path for a given piece. Under the assumption that walls are constructed vertically this might not be as challenging, but this is not the case in general. We considered discretizing the board structure and using A* search for path length, but the process of discretizing the board would make every opponent piece appear as if it occupied 4 times the space, which they could certainly exploit (e.g. by constructing a wall that appears impassable with half as many pieces). Moreover, if an opponent intends to move some of the pieces currently blocking some paths (perhaps we could predict this based on recent moves), previous moves may be inefficient. We would also have to properly balance maximizing the sum of the opponent's path lengths and minimizing our own, and manage the score discontinuity that arises from crossing the goal line. In the end, we thought this approach was interesting, but could lead to a dead end.

A more challenging implementation we could consider is deep reinforcement learning. We could construct a neural net to learn the value of a particular board configuration. On each turn, we could explore various configurations that could arise after a sequence of two moves, and select the moves resulting in the optimal configuration according to the neural net. Of course, the neural net would require training. Since there are nearly infinite states of the game (as pieces are centered at coordinates of type double), we would probably need to provide training data for the neural net, which would be time-consuming and subject to our own personal biases. We could also construct our own value function to train the neural net, although then the

neural net would learn to optimize this value function rather than win the game. Nevertheless, the value function could serve as a proxy for winning. Some features we could consider for training such a neural net include:

1. Number of moves left to build a perfect wall
2. Distance runners are from goal
3. Number of breaches in our wall
4. Center of mass
5. Length of opponent wall explored by runner
6. Distance of breach in opponent wall from runner

Of course, this list is endless and feature engineering is more of an art than a science. Furthermore, once we determined that walls and runners would be part of optimal play, we abandoned reinforcement learning altogether. We thought that this approach was viable, but that it would take a significant amount of time in terms of both coding and training to train a neural net successfully. Even then, if we wanted to improve the neural net, it would be difficult to identify what training data it should learn from so that it performs better in some situations without sacrificing performance on others. Ultimately, we decided to forego deep reinforcement learning to guarantee a competitive player by the end of the project. Still, someone may someday use this technique to build the ultimate Flip AI.

12.1 Frameworks For Coding

In our implementation of the Flip Player, we iterate through our own pieces and attempt to move them to a desired destination, be it a location on the wall or the opponent's goal line. However, we do not have a good framework for coordinating these destinations across all of our pieces. If two pieces have a similar destination, but one is blocking the other, it could be beneficial to move the blocking piece first. As a result, the blocked piece will have one fewer obstacle to avoid. One framework to assist with this implementation could be to assign each piece a destination and a priority. When iterating through pieces, if an optimal move cannot be made, then identify the friendly obstructing piece, and pass some priority on to moving the obstruction.

Another framework we considered implementing was to give each piece a preference relative to other pieces, and a preference over the board as a whole. For example, if a friendly piece and an enemy piece are both nearby, then this piece may want to move closer to the friendly piece if it helps prevent the enemy from making forward progress without sacrificing too much progress towards scoring. Some pieces could

be coded for defensive strategies (attracted to opponent pieces, if they can get in the way) and others for offensive strategies (repelled from opponent pieces, but attracted to scoring). Our main concern with this approach was balancing the overall strategy, and handling the likely polarizing scenario that our opponent will either build a wall or send a runner to score.

13 Conclusion

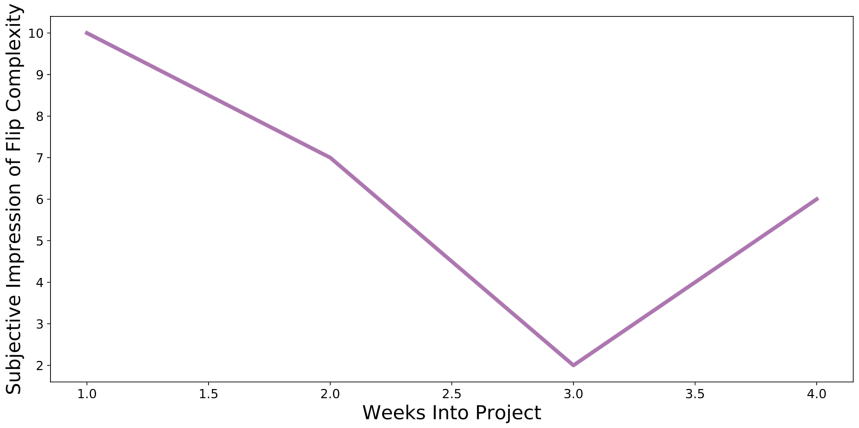


Figure 8: Our belief in the complexity of Flip vs time, on some relative scale

We conclude with 1 more graphic. When beginning the Flip project, the game seemed overwhelmingly complex, with close to perfect play far from possible. The rules were simple, but our moves would need to depend on a seemingly infinite number of configurations our opponent could be in. However, once we realized how to move with the precision and coordination necessary for an 11 piece wall, it seemed the optimal strategy was trivial: build a wall and then send offense to beat defenseless opponents. If $n < 12$ we simply use our get around tangent functions to keep making optimal progress in pushing our center of mass.

Alas, counter strategies to these naive ideas soon emerged and we were once again embroiled in an arms race with the rest of the groups. Nonetheless, in the end, we developed a balanced set of strategies, relying on an intersection of math, creativity, solid programming skills and team work. These strategies would coalesce to form a coherent and robust whole, capable of playing competitive Flip in a variety of domains for n.

14 Appendix of Images

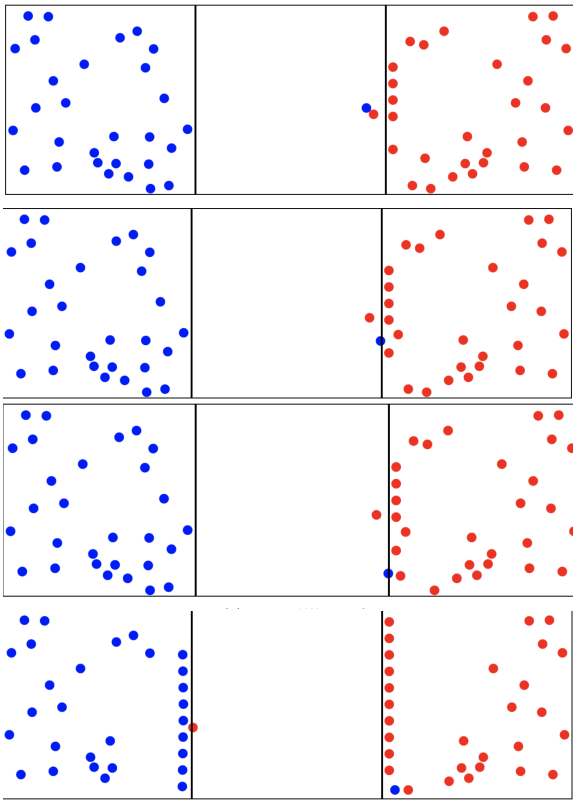


Figure 9: Wall Crawling, we are blue.

15 References

Rosianu, Alexandru. 2016, Jan 14. *Assignment-Problem-Weighted-Bipartite-Matching.* https://github.com/aluxian/Assignment-Problem-Weighted-Bipartite-Matching/tree/master/src

16 Acknowledgements

1. Roza Aceska, for providing this LaTeXtemplate
2. Group 4, for demonstrating our optimized wall prioritization technique could not stop a smart runner for some values of n