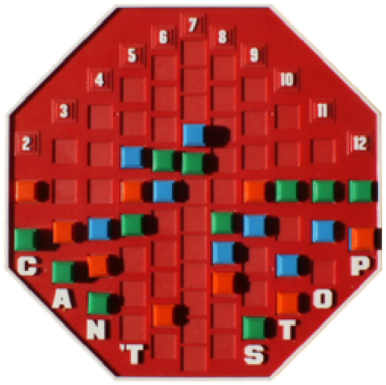
The Can't Stop Game

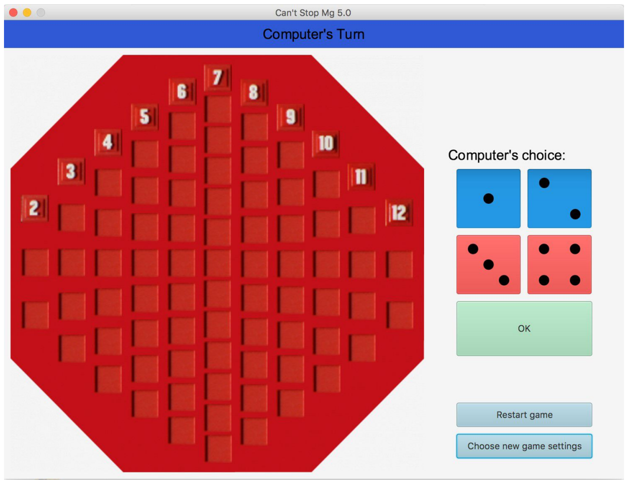
Burcu Canakci - Bilkent University (Ankara, Turkey)
Sofia Serrano - Carleton College (Northeld, Minnesota)
Olivia Roy - Stonehill College (Easton, Massachusetts)
with Dr. Clyde Kruskal (University of Maryland, College Park)

Gameplay and Decisions



The Can't Stop Game is 2-4 player game designed by Sid Sackson and played on a board like the left [1].

Goal: Reach the top of 3 columns.

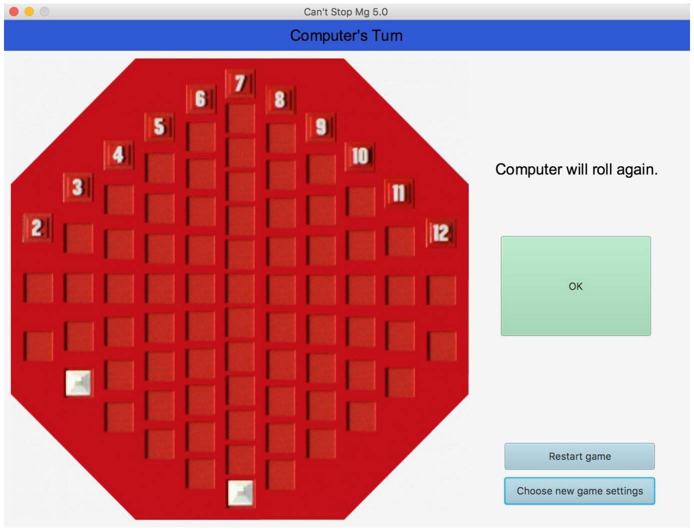


1. At the beginning of each turn a player is given 3 neutral (white) placeholders.

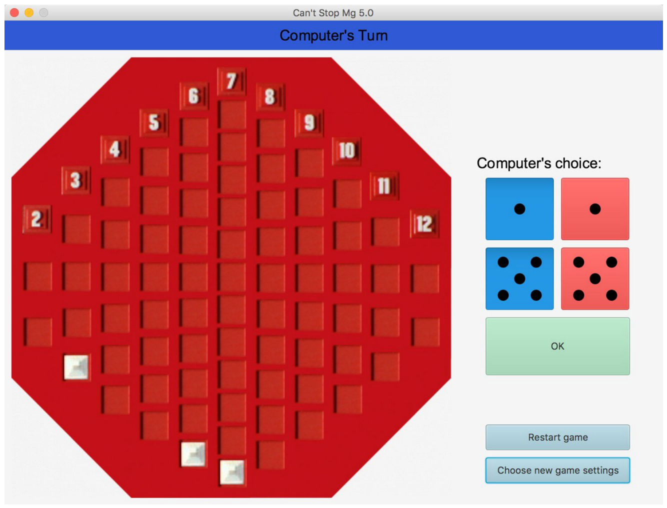
Then the player rolls 4 dice, divides them into 2 pairs and adds up these pairs to obtain 2 sums.

For example, on the right (3,7), (4,6) and (5,5) are the valid sums.

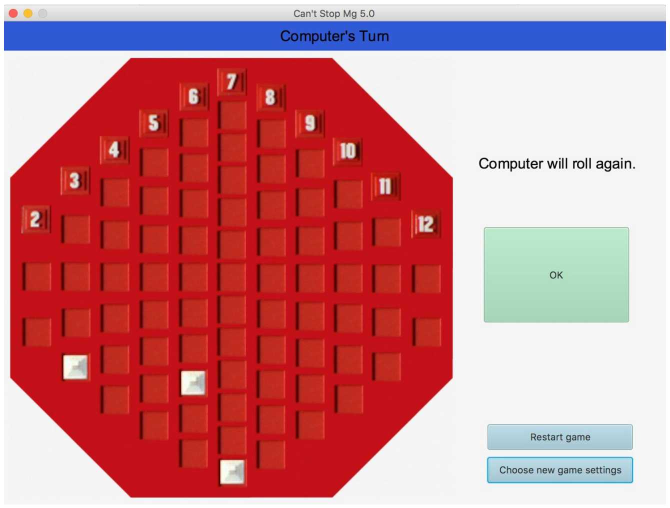
2. If the neutral placeholders are not on the board, they are put on the board according to the pair of sums that the player has chosen.



3. If the placeholders are already on the board, they are advanced one or two points (depending on the sums).

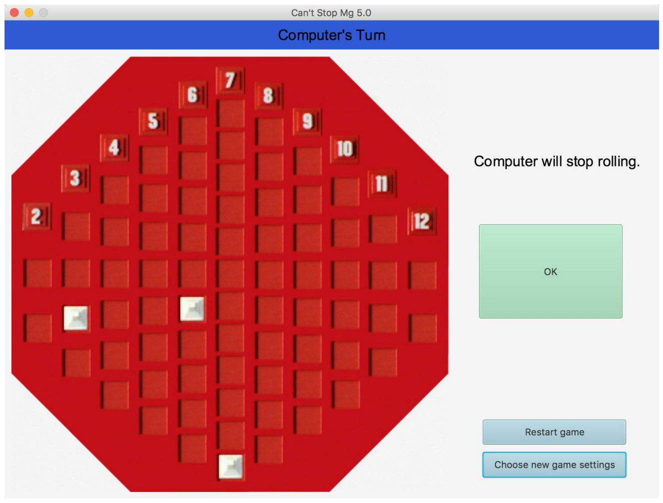


Computer's choice:

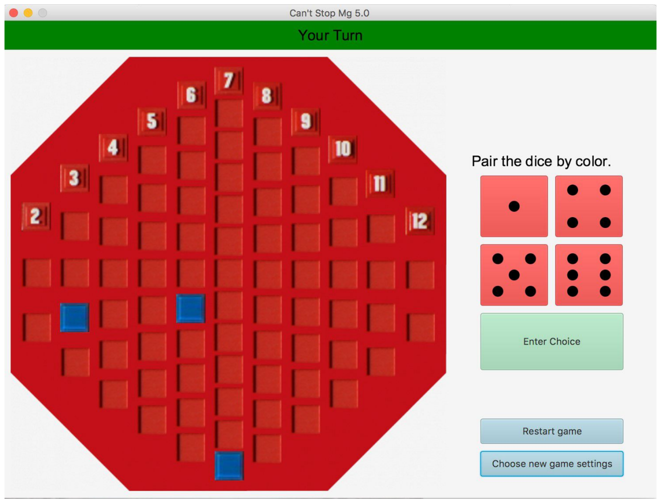


Computer will roll again.

4. After each roll the player is given a choice to **roll** or to **stop**. Once a player decides to stop, their neutral placeholders are exchanged with their permanent ones meaning that they have secured the progress they made during that turn. Once a player stops, their opponents' turns start.

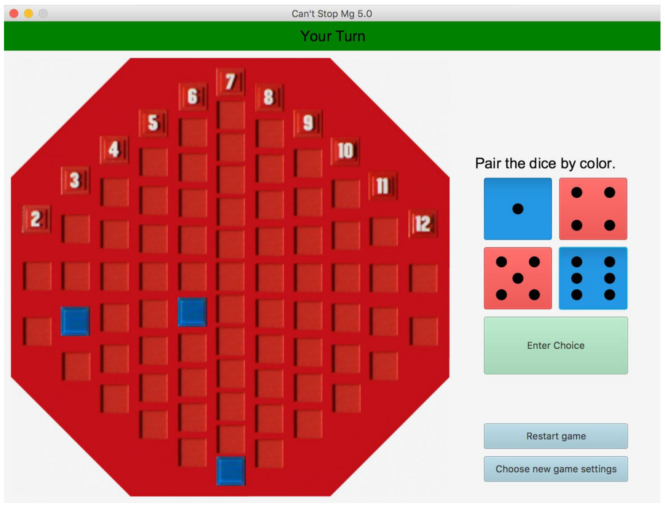


Computer will stop rolling.

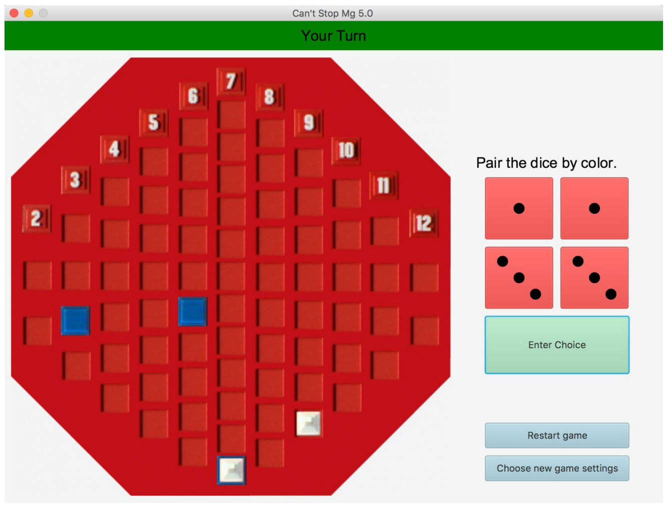


Pair the dice by color.

5. If a player's neutral placeholder gets on the top of an opponent's permanent tile, they **must roll until they get off the tile.**

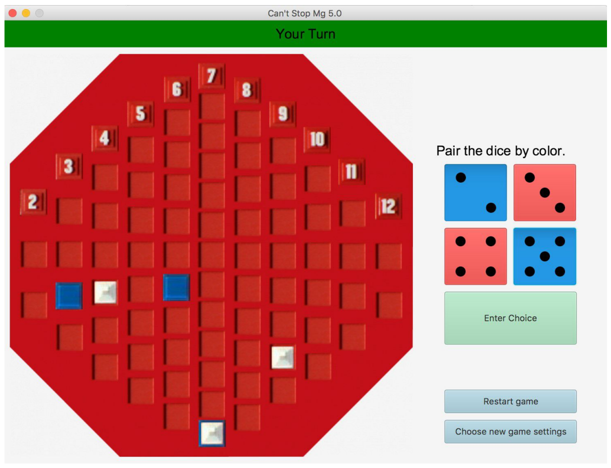


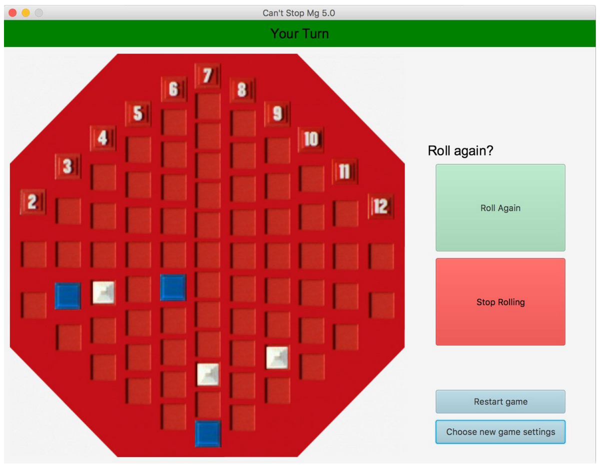
Pair the dice by color.



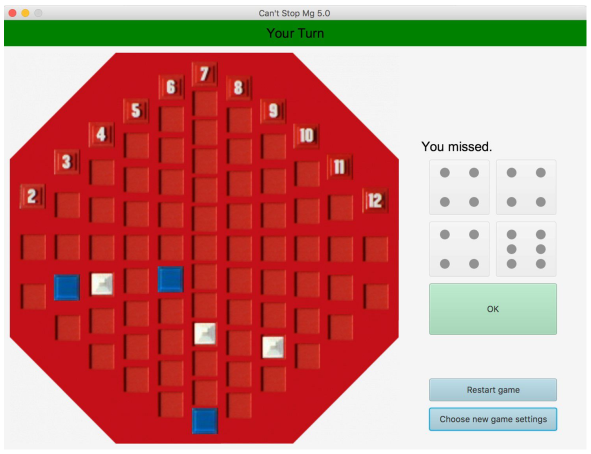
Pair the dice by color.

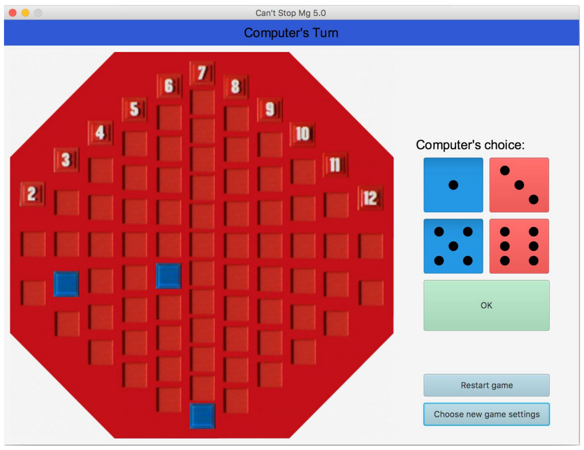
6. Once they get off the tile, they can choose to roll or stop again.





7. If a player rolls 4 dice such that they cannot make any progress on any column that their placeholders are currently on, their turn ends and they lose all progress made that turn.





8. If a player can make a move after rolling, they must make it.

9. Both of the sums in the pair may not be used.

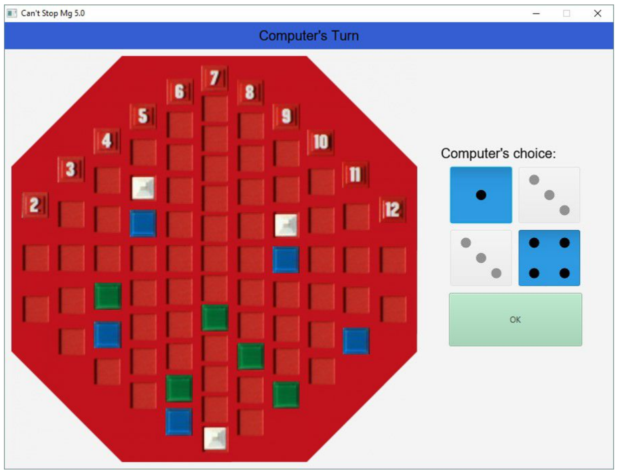
For example, if a player is on columns 2,3,4 and they roll 1,1,5,6, they will just make progress on the 2-column and discard the sum of 11.

Similarly, if they roll 1,1,2,2, they can choose which column they will make progress on.

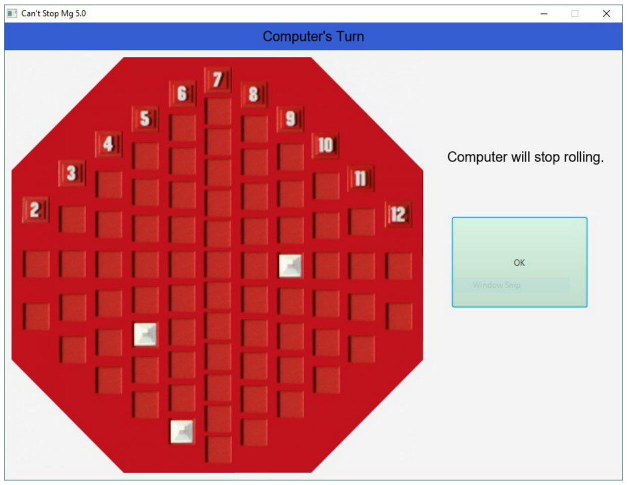
10. If a player reaches the top of a column, and stops, they capture that column, and no player can make any moves on that column any more.

Pairing Up The Dice

One of the two important decisions made in the game is pairing up the 4 dice to obtain 2 sums. The factors that need to be considered while deciding on a pair or sum depend both on probability and the current game situation.



Deciding to Stop



The other important decision the player needs to make is deciding on when to stop rolling.

This decision depends on the expected progress on columns as well as the game dynamics.

For example, a player could decide to be more liberal if they are behind or vice versa.

Computer Approaches

The Mathematical Model:

For this model, the computer makes its decisions based on probabilities and expected progress.

Deciding on stopping or rolling again:

For this problem, several different formulas were used, each building upon the previous ones.

Considering total progress on columns together

Assume a player is on 3 columns and define the following as:

p_1 = probability of making 1 unit progress on any of the 3 columns
p_2 = probability of making 2 units of progress on any of the 3 columns
c = total units of progress made on the 3 columns this turn

Then the expected progress of the player for the current turn after rolling one more time can be defined as

$$p_1 * (c + 1) + p_2 * (c + 2) + (1 - (p_1 + p_2)) * 0$$

Having calculated the expected progress, the player can be instructed to only stop if its current progress is greater than or equal to its expected progress after the next roll. So the decision to roll again is the boolean result of the inequality

$$p_1 * (c + 1) + p_2 * (c + 2) > c$$

Note: Given 3 columns, it is possible to find the maximum value of c that would satisfy this inequality, yielding the total progress value required per turn to stop.

$$\frac{p1 + 2 * p2}{(1 - p1 - p2)} > c$$

Considering total progress on columns individually

Doing more careful probability analysis, the formula in the previous section can be extended into the following form. Assuming, again, that a player is on 3 columns, define the following as:

p_i = probability of making 1 unit progress on only the ith column
p_{ii}= probability of making 2 units of progress on the ith column
p_{ij} = probability of making 1 unit progress on both the ith and jth columns
c_i = total units of progress made on the the ith column this turn

Then we can express the expected progress for the current turn with

$$\begin{gathered}
p_1 * ((c_1 + 1) + c_2 + c_3) + \\
p_2 * (c_1 + (c_2 + 1) + c_3) + \\
p_3 * (c_1 + c_2 + (c_3 + 1)) + \\
p_{12} * ((c_1 + 1) + (c_2 + 1) + c_3) + \\
p_{13} * ((c_1 + 1) + c_2 + (c_3 + 1)) + \\
p_{23} * (c_1 + (c_2 + 1) + (c_3 + 1)) + \\
p_{11} * ((c_1 + 2) + c_2 + c_3) + \\
p_{22} * (c_1 + (c_2 + 2) + c_3) + \\
p_{33} * (c_1 + c_2 + (c_3 + 2)) + \\
(1 - (p_1 + ... + p_{33})) * 0
\end{gathered}$$

Also, we can express the total progress made on this turn with

$$c_1 + c_2 + c_3$$

and we can compare these expressions to come to a decision of rolling or stopping.

Considering total progress on columns individually considering weights

Since the columns on the game board have different lengths, one unit progress on a column has different value for each column. For example, advancing one unit on the 2-column is ⅓ of the way to capture the column whereas you need to make 13 unit progress on the 7-column in order to capture it.

In this probabilistic model, each column is given a weight value. This value is defined as

$$w_i = 1/p_i$$

where p_i = the possibility of getting a sum of *i* after rolling 4 dice

Now we can extend the previous expression for the expected progress with

$$\begin{gathered}
p_1 * (w_1 * (c_1 + 1) + w_2 * c_2 + w_3 * c_3) + \\
p_2 * (w_1 * c_1 + w_2 * (c_2 + 1) + w_3 * c_3) + \\
p_3 * (w_1 * c_1 + w_2 * c_2 + w_3 * (c_3 + 1)) + \\
p_{12} * (w_1 * (c_1 + 1) + w_2 * (c_2 + 1) + w_3 * c_3) + \\
p_{13} * (w_1 * (c_1 + 1) + w_2 * c_2 + w_3 * (c_3 + 1)) + \\
p_{23} * (w_1 * c_1 + w_2 * (c_2 + 1) + w_3 * (c_3 + 1)) + \\
p_{11} * (w_1 * (c_1 + 2) + w_2 * c_2 + w_3 * c_3) + \\
p_{22} * (w_1 * c_1 + w_2 * (c_2 + 2) + w_3 * c_3) + \\
p_{33} * (w_1 * c_1 + w_2 * c_2 + w_3 * (c_3 + 2)) + \\
(1 - (p_1 + ... + p_{23}))*0
\end{gathered}$$

and the current progress as

$$w_1.c_1 + w_2.c_2 + w_3.c_3$$

We can, again, compare the values of these expressions to decide whether to roll or stop.

After comparing this version with the previous ones, we came to the conclusion that this one makes decisions that coincide with our intuition the most and decided to do work with it for our mathematical approach.

Comparing the mathematical model with exhaustive search

In order to get an idea of how well the rolling/stopping approach worked, the following simulation was performed.

Considering the 6,7,8-columns, define the following

l_7: limit of unit progress made on the 7-column
l_{68}: limit of unit progress made on the 6,8-columns
c_7: unit progress made on the 7-column
c_{68}: unit progress made on the 6,8-columns
w_7, w_{68}: relative weights of the columns

and assume a program keeps on rolling until

$$w_7.c_7 + w_{68}.c_{68} >= w_7.l_7 + w_{68}.l_{68}$$

and calculate the weighted total progress this program makes over 1000000 turns.

Now, letting l_7 go from 1 to 13, look for the value of l_{68} (which is between 1-22) that yields the maximum weighted total progress.

Doing this we get

l_7	ideal l_{68}^{*}	total c_7	total c_{68}	total weighted progress
1	15	1524905	4732089	3238416.5
2	14	1520130	4736400	3238495.9
3	13	1512206	4739533	3236479.4
4	12	1516493	4740706	3239102.6
5	12	1519482	4738609	3239374.3
6	10	1526175	4733003	3239496.3
7	10	1525346	4727844	3236353.9
8	8	1529201	4733986	3241430.8
9	8	1525945	4734284	3240073.6
10	7	1522858	4747752	3245831.5
11	7	1517469	4743007	3240786.5
12	6	1513820	4738304	3236574.4
13	4	1515006	4737573	3236736.2
			Mean	**3239165.5**

(* +- 1 as observed from different runs)

The mathematical model we used also stops rolling around the given limits. More importantly, over 1000000 turns the mathematical model makes the following progress.

total c_7	total c_{68}	total weighted progress
1526867	4724978	**3235531.0**

As seen, the progress values that the mathematical method produces match to the ones obtained by brute-force search.

Deciding on how to pair the dice

Given 4 rolls, deciding on how to pair the dice is a relatively harder problem and it is more difficult to come up with a robust formula that models the game situation as before. Some important factors to consider are

- ➔ closeness to finishing a column
- ➔ avoiding getting stuck on an opponent’s tile (and losing the chance to stop)
- ➔ the likelihood of winning the column (e.g. not ideal to start the 7-column if the opponent is very close to winning it)

To solve this problem, a few different pair deciding algorithms were implemented.

Deciding based on relative final progress (m1)

1. Given two different choices of columns to advance on, compare the relative total progress that will have been made on the columns if they are chosen. To do this, use

$$\frac{\textit{total unit progress on i}}{\textit{length of i}} = \frac{\textit{relative total progress on i}}{\textit{length of the longest column}}$$

Deciding based on relative progress and considering getting stuck (m2)

1. While considering a column *i*, calculate the rough expected progress that would be made on *i* at the end of the turn. (This is done using the first (simple) formula mentioned in the previous section and the weights of the current columns.)
2. Add this expected progress value to the current progress made on *i* and see if any opponent tiles are there.
    a. If there are, set *p* to the probability of getting off the opponent tiles on *i.* (Use the probability of rolling a sum of *i* for this.)
    b. If not set *p* to 1.
3. Calculate *p.(relative total progress on i)* and use this value to compare between columns.

Deciding based on progress, getting stuck and likelihood of winning (m3)

1. Use the above method to calculate *p.(relative total progress on i).*
2. Get the *likelihood of winning a column* by calculating

$$0.5 + \frac{(\textit{the player's progress once the column is chosen}) - (\textit{opponent's progress})}{\textit{length of the column}}$$

3. Multiply the values in (1) and (2) and use them for comparison.

Comparison of m1, m2, m3

It is possible to simulate games where these methods play against one another, so we made them play each other 1000 times. Below is a list of results.

m1 & m2 (m1 goes first)	m1 (559) m2 (441)
m1 & m2 (m2 first)	m2 (602) m1 (398)
m1 & m3 (m1 first)	m1 (543) m3 (457)
m1 & m3 (m3 first)	m3 (579) m1 (421)
m2 & m3 (m2 first)	m2 (593) m3 (407)
m2 & m3 (m3 first)	m3 (537) m2 (463)

Since there was no significant difference between how well different methods played, we decided to use m1 while doing rollouts as m1 is simpler and significantly faster than the other methods. (1000 games using m1 takes half as much as 1000 games using m2.)

Rollouts

The second computer approach we implemented was the rollout technique. Before making a decision, for each option, the computer plays thousands of games against itself starting from the game state after having chosen that option, counting the number of wins along the way. Against its opponent, after these simulations, the computer picks the option that wins the most games. This technique produces decent choices because if two players of equal ability play the same game state with each other a sufficient number of times (this number decreases as player ability increases), the *correct* decision will win more games. The rollout method is used and has shown success in computer programs that play backgammon [2].

The rollout games are played by the computer using the mathematical model.

For roll/stop decisions: We use the last formula that considers columns individually with weights. For simplification and better performance, the implementation of this formula

stops if any column is won

rolls if any placeholder is available

uses the formula in all other cases

For pairing up the dice: We use the method that makes decisions only based on relative progress.

We use rollouts while making both of the game decisions. In our implementation, for each option, the computer plays 60,000 games. Playing a set of 60,000 games takes ~3 seconds so depending on the number of options the computer has, which ranges from 2 to 6, the computer takes from 6 to 18 seconds while making a decision[1].

Opponent Error

Some of the optimal game decisions change depending on how the opponent is playing. For example, if the computer's opponent is a conservative player who often stops earlier than they should, then the computer can also afford to play safer than usual, in fact, this could work out better.

In the formula we defined for making the roll/stop decisions, we compare the expected progress to the current progress of the player and stop if the expected progress is less than or equal to the current progress. We alter this formula to include an *error* value as follows.

$$(expected\ progress) > (current\ progress) + (current\ progress).error$$

The error-values we test for the opponent range between 0 and 1.5, with 0.05 increments.

For each test, starting from the game state where the opponent stopped, 1500 games are played each for the decision of stopping and rolling at that state, between an errorless computer program and and a program with error equal to the test value. The *error* of the opponent is set to the smallest test value that results in the opponent's stopping decision to be justified according to the rollout.

The error of the opponent is recalculated after their every stop decision, and an average value is kept.

After the error of the opponent is calculated, the computer does the 60,000-game rollouts for the game decisions playing against a version with error rather than *itself*. This results in the computer playing relatively more conservative if the opponent's error is high.

Using Neural Networks

While looking for other computer approaches to implement, we considered looking into techniques that are used in programs playing Backgammon, since we thought Backgammon and The Can't Stop Game were alike in terms of incorporating both luck and strategy. One technique we found interesting was using self-teaching neural networks like TDGammon, which is a very successful Backgammon program [2].

For this we implemented a program to

1. Play a game between two players using the mathematical model, recording the board information and decision after every move.
2. Record the game result.

[1] using an Intel^R Core™ i7-6500U CPU @ 2.50 GHz and 8GB of RAM.

3. Encode the board (for deciding on pairs) and the board & roll/stop combination (for deciding on rolling/stopping) and feed them into two different networks as training data with output as the corresponding game result (win, lose).
4. Repeat this process

The input encodings were 448-bit representations of the game board (449 in the case of including the roll decision). The output was a 1-bit value representing win or loss.

Although we found this method promising, it required considerably more time than we had left to show desirable results (the training period). Therefore, we eventually decided to go back to the rollout approach and try to speed up/optimize our program.

Result

We found that the rollout approach resulted in a computer program that plays the game very well and wins most of the time.

References

[1] Wikipedia, *The Can't Stop Board Game*. 2016.
[2] G. Tesauro, "Programming backgammon using self-teaching neural nets", *Artificial Intelligence*, vol. 134, no. 1-2, pp. 181-199, 2002.