GROUP 16: Mastermind

Ryan Jacobson (21rkj6)

Evan Kreutzwiser (21ejk12)

Liam Beenken (22ltb1)

Jacob McMullen (21jwm21)

Course Modelling Project

CISC/CMPE 204

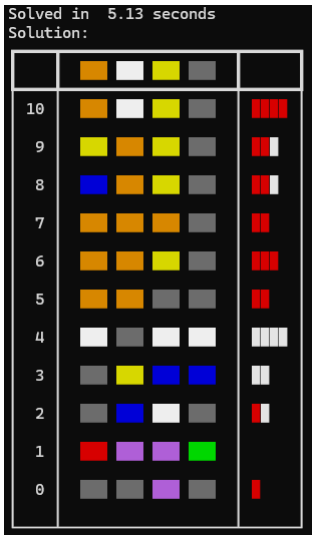
Logic for Computing Science

December 7, 2023

Abstract

Our setting models the board game Mastermind. Mastermind is a code guessing game where a solution made of 4 pegs of 8 colors in a specific order is generated before the game begins. It is the players job to make guesses, receive feedback, and guess again, with the goal of making a guess which matches the solution. For each guess, a player is given red and white feedback pegs. For each red feedback peg, one peg in the guess matches the color and position of a peg in the solution. For each white feedback peg, one peg in the guess matches a color, but not the position, of a peg in the solution.

Looking at the example below, guess 8 has a blue peg in the first column, an orange in the second, a yellow in the third, and a grey in the forth. Since the yellow and grey pegs in the guess match the position in the solution, the player receives 2 red feedback pegs. Since an orange peg is present in the solution, but is in the first column as opposed to the second, the player also receives 1 white feedback peg.



Our model attempts to win Mastermind by making a guess which matches the solution. Like a human player, our model uses logic to interpret previous guesses and the feedback received from those guesses to make new informed guesses. Our constraints serve to dictate how the board develops and how the program knows which colors and positions remain as valid guesses.

Propositions

There are 4 types of propositions in our model. The X propositions represent the state of the game board, and the C propositions store the correct answer, which the model aims to reach. It receives feedback about its guesses in the form of R and W propositions representing how many colors of the guess were in the right position or included somewhere else in the answer.

X_{rnc}: True if the color c is present in the slot at row r, column n of the board.

C_{nc}: True the answer has color c in column n

R_{rk}: True if guess r contains k colors that are in the correct position. Represents the reg pegs in a Mastermind game

W_{rk}: True if guess r contains k colors that are present in the answer but not in the correct position. Represents the white pegs in a Mastermind game

For each guess, board information is recorded in the X_{rnc} propositions, and a row of these propositions without a preexisting value are where the solver's next guess is recorded. After the guess is complete, the solver is given feedback to use in the next guess, telling it how many colors it guessed correctly.

We use a set of 8 colors, chosen such that the first letter of each color is unique:

Red, Orange, Yellow, Green, Blue, Purple, White, and Silver

The subscript c can be r, o, y, g, b, p, w, or s.

Constraints

Capital R as a row number refers to the row the model is current making a guess for.

In any position on the board, no more than a single color can be present

E.g.: $X_{rnp} \rightarrow \neg X_{rnr} \wedge \neg X_{rno} \wedge \neg X_{rny} \wedge \neg X_{rng} \wedge \neg X_{rnb} \wedge \neg X_{rnw} \wedge \neg X_{rns}$

A guess cannot match any previous guess. E.g. If a previous guess was red, blue, purple, orange:

$\neg(X_{r0r} \wedge X_{r1b} \wedge X_{r2p} \wedge X_{r3o})$

The rest of the constraints use helper functions designed to ensure a specific amount of constraints in a list are met, without requiring that all of them be true. The constraints generated are very long, and will be written using these shorthands instead:

exactly_k(number, constraint list): True when any combination of k constraints in the list are met.

at_least_k(number, constraint list): True when at least k of the constraints are met, introducing an element of uncertainty that allows the model to randomly "guess".

There are k red pegs if k colors in the guess match the answer

Constraints: $X_{rnc} \wedge C_{nc}$

$(\neg R_{rk}, \neg exactly\_k$(number of pegs, constraints))$\vee(R_{rk}, exactly\_k$(number of pegs, constraints))

There are k white pegs if k colors in the guess are in a the answer but not the correct position.

Constraints (E.g. for a color in column 0): $X_{r0c} \wedge (C_{1c} \vee C_{2c} \vee C_{3c})$

$(\neg W_{nk} \wedge \neg exactly\_k$(number of pegs, constraints))$\vee (W_{nk} \wedge exactly\_k$(number of pegs, constraints))

To respond to red peg feedback (Color is in the right position), pick k colors from the guess and reuse them in the same position.

Constraints: $X_{rnc} \wedge X_{Rnc}$

$R_n k \rightarrow at\_least\_k$(number of pegs, constraints)

To respond to white peg feedback (Color is in the answer, but in a different position), pick k colors from the guess and reuse them in any other position.

Constraints: (E.g. for a color in column 0): $X_{r0c} \wedge (X_{R1c} \vee X_{R2c} \vee X_{R3c})$

$W_{nk} \rightarrow at\_least\_k$(number of pegs, constraints)

Additionally, R_{n0} being true (no red pegs for a guess) implies that none of the colors in that guess can be used in the same position again.

Early Constraints ("Wordle-style")

These constraints were used in an earlier version of the model, which make feedback pegs specific to the column the color is in, similar to how the game Wordle provides feedback that is specific to the position the letter is used. The code still exists in the GitHub repository, and can be accessed by changing the column specific feedback pegs argument to True in the the main function of run.py.

A color in the correct position in a previous guess is used in the same spot again

$(X_{rnc} \wedge C_{nc}) \rightarrow X_{Rnc}$

If a color is present in the answer but used in the wrong position in a previous guess, the color must be used in one of the other 3 positions in the next one.

E.g., for a color in column 0:

$(X_{r0c} \wedge (C_{1c} \vee (C_{2c} \vee C_{3c}))) \rightarrow ((X_{R1c} \vee X_{R2c} \vee X_{R3c}) \wedge \neg X_{R0c})$

A guess must contain 1 color in each column.

E.g., if a guess contains color r in column 0:

$X_{R0r} \rightarrow \neg X_{R0o} \wedge \neg X_{R0y} \wedge \neg X_{R0g} \wedge \neg X_{R0b} \wedge \neg X_{R0p} \wedge \neg X_{R0w} \wedge \neg X_{R0s}$

This pairs with the constraint preventing multiple colors from sharing a position to ensure exactly one color is present in each position of the guess.

Model Exploration

When deciding our constraints, one potential issue became apparent quickly; how long would it take a Python model using logic to solve a full game of Mastermind? This question then became: how much RAM would it take to solve a full game? We knew this would be a problem from the start so we had to change a rule of the game, deciding that each feedback peg would correspond to a specific guessing position. Eventually we overcame this, but we referred to the early models that used this change as "Wordle-style" models. This new rule would make it much easier for the Python model using logic to solve a game of Mastermind, though time and memory issues still occurred.

In the first functioning model that was made, we had the model play the whole game at once. The model would always start with a random guess since it had no information to start with. Then it would proceed to make more guesses based off of the constraints such as using new colors when others were not in the solution, moving colors around from the wrong positions until they're in the correct positions, and keeping colors in place when in the correct positions. However, this model came with a major flaw where it used up too much RAM and took too long to execute. Essentially, the model would run the entire board through the constraints all at once, and calculate every possibility that would lead to winning the game. The massive amount of possible solutions and time it took the model to process it all quickly consumed too many resources. Due to this issue, all of our following models processed one row at a time.

One reoccurring issue we had with our early models was that it would "cheat" the game. This issue occurred in both the model that ran all rows at once and and early models that executed one row at a time. The way we implemented the constraints at first was causing the model to calculate the feedback pegs that would satisfy the constraints for winning the game at the same time it was calculating what colors and positions to choose. This led to the model having the output of feedback pegs for each position and color before actually choosing the positions and colors. As a result, the model would always beat the game in around 1 to 3 guesses. It took a while for us to notice that this was happening, since there were other smaller issues that we thought we causing the model to solve the game it such few attempts. Our fix for this was to delay the calculation of the feedback pegs until the next row.

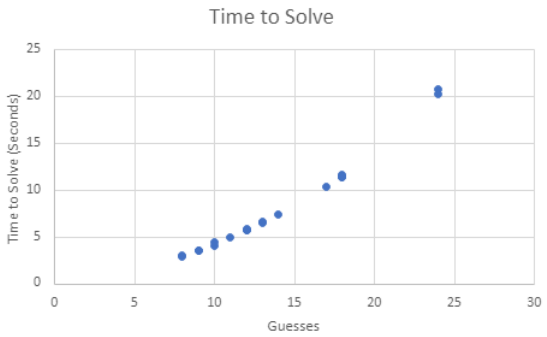
After solving the cheating issue and making other various improvements to run time and resource use, we decided it was time to have the model play Mastermind correctly by removing the rule of each feedback peg corresponding to a certain position. The biggest hurdle with making this advancement was allowing the model to pick which rows it wanted to apply the feedback to. The breakthrough we had that solved this were creating constraint-generating helper functions that add functionality not provided by the library. The two helpers take in a list of constraints and generate large constraints ensuring that exactly or at least a certain number of the given constraints are met, without requiring all of them to be true. We used the exactly_k helper to count the number of feedback pegs to give the solver, and used the at_least_k helper for responding to feedback pegs. It allowed us to create constraints for every peg describing

how the feedback would be applied, and to make sure that the feedback is considered for at least k of the colors in that guess. Unfortunately, the generated constraints were very long and required a significant amount of resources in the solver because they enumerate every valid combination of constraints being satisfied or not, and the number of combinations exponentially increases with the length of the constraint list. The generated constraints were horribly long, and although the SAT solver made no complaints it left us with the same performance problems as before.

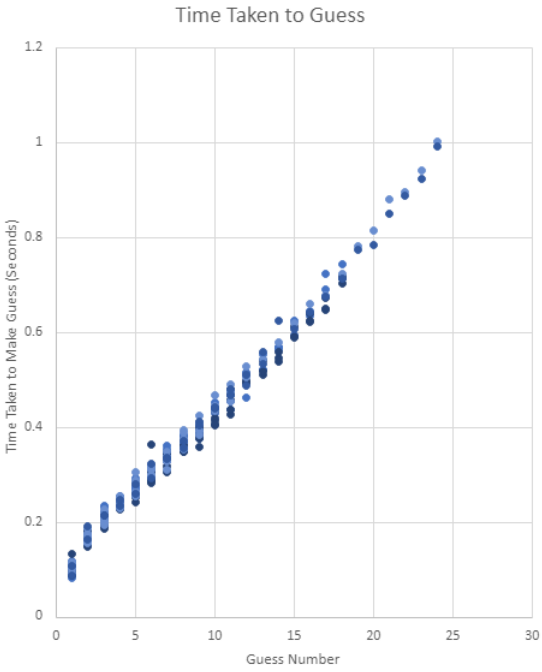
The workaround that solved our performance issues was moving a part of the logic constraints used by the SAT solver into Python. Specifically, it was the part of the constraints that look at the previous position of colors to help generate the next guess. Being generated by the helpers from long lists, the constraints would significantly slow down the model, but were crucial for running Mastermind correctly as it allowed the model to use feedback from previous guesses to create better guesses. However, the SAT solver could not process the constraints efficiently. Moving the piece of the constraints that checks whether a color is present into Python allowed for the SAT solver to continue processing the logic of everything else without having to be significantly slowed down by the oversized constraints. This workaround reduced the size of the constraint lists and allowed our final model to run Mastermind much more quickly without sacrificing guess quality.

Our final model is able to run Mastermind very smoothly, although it is not optimal. It is easily able to play a standard 4 column game faster than a human typically can, but it is not perfectly efficient. It will often make strange guesses that are not entirely necessary, or that don't provide new information. This is somewhat of a byproduct of allowing the model to run quickly while using the SAT solver's logic.

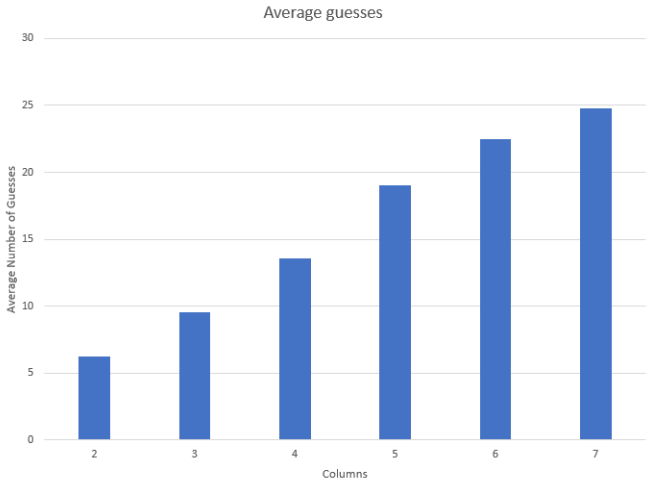
The model is able to run Mastermind with anywhere between 2 to 8 columns. All colors are always included regardless of how many columns there are, and there are no duplicate colors allowed. A user can also set exactly what the winning colors and positions are, pick the number of columns, and output results of repeated tests in CSV files for analysis. From testing a standard 4 column game, the model makes an average of 13.9 guesses and takes an average of 9.5 seconds to win depending on hardware capabilities. It can solve any 4 column game in as little as 1 guess (this is just pure luck), and we have seen the model take upwards of 24 guesses.



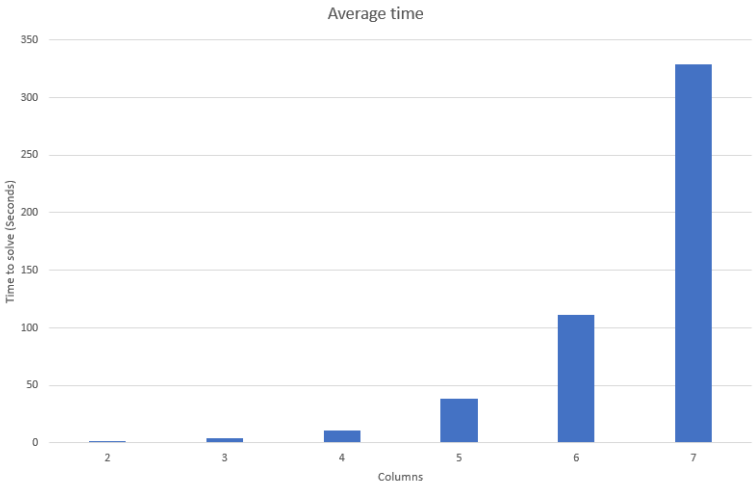
The time for any individual guess depends on how many previous guesses have been made. Each guess tends to take longer than the last.



We have also tested games with different numbers of columns. However, We didn't run enough games with 8 columns to include in the graphs because it can take upwards of 25 minutes for the model to solve an 8 column game. As more columns were used, the average number of guesses increased quite linearly.



However, as more columns were used, the average time to solve the game scaled exponentially.



First-Order Extension

In a predicate logic based system, we would be working with a domain of numbers and colors, specified by the user at the start of the game.

To differentiate numbers from colors, we would have a partition $\{C, N\}$ of our domain U:

C: $\{red, blue, green...\}$, containing the colors that the user elects to use
N: $\{0, 1, 2...n\}$, where n is the maximum amount of guesses per game
$C \cup N = U$
$C \cap N = \emptyset$

For simplicity when creating constraints, it would also help to define color and number equality logically.
We can create a predicate $E(c_1, c_2)$, which would contain the tuples:
$\{(red, red), (blue, blue), (green, green), (1, 1), (2, 2), (3, 3)...\}$, equating to true if c_1 and c_2 are equal, and false if they are not

We would also be able to model the game in simpler terms, using predicates in place of our propositions.

Instead of tracking previous guesses with a proposition for each individual spot on the board, we could use a predicate to contain both the row number, and order of colors. This can be written as:
$G(r, c_1, c_2, c_3, c_4)$
Where r is the row number, and c_1 to c_4 are the colors guessed on that row. After each guess, a new tuple would be added.

We could also simply model the correct code with: $C(c_1, c_2, c_3, c_4)$, only true for a single tuple, which is the final code.

Additionally, in a predicate logic setting, we would be able to use more advanced guess making algorithms. In his paper *The Computer as Mastermind*, Knuth (1976) outlines a near optimized strategy, which uses an application of the minimax algorithm, always choosing the guess that eliminates the highest number of possible solutions. We would start with the set of all possible solutions, and narrow it down based on the feedback from each guess. For example, if a guess responds with one red feedback peg, we can eliminate all solutions that do not contain exactly one of the four colors, in the same position as the guess. To model this elimination in a predicate logic setting, we would have to slightly change the way we model feedback. Instead of using a row and count variable, we could define predicates labelling each row with the property of having 1, 2, 3, or 4 red/white/empty feedback pegs.
$R_1(r)$, $R_2(r)$, $R_3(r)$, $R_4(r)$
$W_1(r)$, $W_2(r)$, $W_3(r)$, $W_4(r)$
$E_1(r)$, $E_2(r)$, $E_3(r)$, $E_4(r)$

Then we could use that information to set up the elimination constraints:
In the case of only one red feedback peg, we can narrow down our possible solutions as shown in the following large constraint. Essentially it states that if one red peg is present in the feedback for a row with the guess a, b, c, d, then the correct code is exclusively one of (a, x_1, x_2, x_3), (x_1, b, x_2, x_3), (x_1, x_2, c, x_3), or (x_1, x_2, x_3, d), where each of the x's cannot be equal to any of the other three colors in the initial guess:

$$\forall r.\forall c_1.\forall c_2.\forall c_3.\forall c_4.(G(r, c_1, c_2, c_3, c_4 \wedge R_1(r) \rightarrow$$
$$\exists x_1\exists x_2\exists x_3(\neg E(x_1, b)\wedge\neg E(x_1, c)\wedge\neg E(x_1, d)\wedge\neg E(x_2, b)\wedge\neg E(x_2, c)\wedge\neg E(x_2, d)\wedge$$
$$\neg E(x_3, b) \wedge \neg E(x_3, c) \wedge \neg E(x_3, d) \wedge C(a, x_1, x_2, x_3))$$
$$\oplus$$
$$\exists x_1\exists x_2\exists x_3(\neg E(x_1, a)\wedge\neg E(x_1, c)\wedge\neg E(x_1, d)\wedge\neg E(x_2, a)\wedge\neg E(x_2, c)\wedge\neg E(x_2, d)\wedge$$
$$\neg E(x_3, a) \wedge \neg E(x_3, c) \wedge \neg E(x_3, d) \wedge C(x_1, b, x_2, x_3))$$
$$\oplus$$

$\exists x_1 \exists x_2 \exists x_3 (\neg E(x_1, a) \wedge \neg E(x_1, b) \wedge \neg E(x_1, d) \wedge \neg E(x_2, a) \wedge \neg E(x_2, b) \wedge \neg E(x_2, d) \wedge$
$\neg E(x_3, a) \wedge \neg E(x_3, b) \wedge \neg E(x_3, d) \wedge C(x_1, x_2, c, x_3))$
$\oplus$
$\exists x_1 \exists x_2 \exists x_3 (\neg E(x_1, a) \wedge \neg E(x_1, b) \wedge \neg E(x_1, c) \wedge \neg E(x_2, a) \wedge \neg E(x_2, b) \wedge \neg E(x_2, c) \wedge$
$\neg E(x_3, a) \wedge \neg E(x_3, b) \wedge \neg E(x_3, c) \wedge C(x_1, x_2, x_3, d)))$
Where $r \in N$ and $c_1, c_2, c_3, c_4, x_1, x_2, x_3 \in C$

Similar constraints could be set up for each of the 15 possible feedback responses (2^4 - 1, since 3 red 1 white is impossible), and as the game progresses, the list of constraints for a valid solution will become smaller and smaller. To make each guess, we would have to find the guess that would eliminate the most solutions in the worst case. We would append the constraints that would theoretically be applied for each possible next guess, and the one eliminates the most solutions on it's worst case feedback would be the guess that the solver chooses. This would likely be too computationally intensive to effectively run on a laptop.

Many of the constraints defined in our propositional logic setting would still be applicable in this first order extension.

For example, the constraint stating that a guess cannot match any previous guess, can be adapted to a predicate form:

$\forall r_1 . \forall r_2 . \forall c_1 . \forall c_2 . \forall c_3 . \forall c_4 (G(r_1, c_1, c_2, c_3, c_4) \wedge \neg E(r_1, r_2) \rightarrow \neg G(r_2, c_1, c_2, c_3, c_4))$
Where $r_1, r_2 \in N$ and $c_1, c_2, c_3, c_4 \in C$

This states that if a guess (c_1, c_2, c_3, c_4) has been made on row r_1, then it cannot be made on row $r2$, as long as r_1 != r_2

Additionally, the constraint stating that in any position on the board, no more than a single color can be present, can be adapted to the predicate form:
$\forall r . \forall c_1 . \forall c_2 . \forall c_3 . \forall c_4 . \forall x .$
$(G(r, c_1, c_2, c_3, c_4) \rightarrow$
$(\neg(\neg E(x, c_1) \wedge (G(r, x, c_2, c_3, c_4)))$
$\wedge$
$\neg(\neg E(x, c_2) \wedge (G(r, c_1, x, c_3, c_4)))$
$\wedge$
$\neg(\neg E(x, c_3) \wedge (G(r, c_1, c_2, x, c_4)))$
$\wedge$
$\neg(\neg E(x, c_4) \wedge (G(r, c_1, c_2, c_3, x))))$
Where $r \in N$ and $c_1, c_2, c_3, c_4, x \in C$

This states that if $G(r, c_1, c_2, c_3, c_4)$ is true, then it is not true that a color x can be both a different color from one of c_1, c_2, c_3, or c_4, and be in the same place as that color in row r.

Jape Proofs

Jape restricts the letters which can be used to represent variables and functions. To make our proofs easier to read and to make them match the propositions and constraints listed above, we have modified Jape to allow for the use of any lowercase letter as a variable and any uppercase letter as a function. We have included the files needed to make this modification in our /documents/final folder on GitHub, along with instructions.

Look for the file 'INSTRUCTIONS_natural_deduction_more_vars.txt' to learn how to use our modifications and load our proofs in Jape.

Our project has undergone many changes regarding how the game is played. As explored above, the game used to follow "Wordle-style" rules where the next guess would be made based on comparisons between the current guess and the correct solution. Since the Wordle-style model is more easily translated into handwritten proofs, that is what we have decided to use for the following sections.

Proof 1:

1:	actual j, X(a,j), ¬C(a,j), ∃y.(C(y,j)), ∀y.∀z.(X(y,z))	premises
2:	actual i, C(i,j)	assumptions
3:	∀z.(X(i,z))	∀ elim 1.5,2.1
4:	X(i,j)	∀ elim 3,1.1
5:	X(i,j)∧C(i,j)	∧ intro 4,2.2
6:	∃y.(X(y,j)∧C(y,j))	∃ intro 5,2.1
7:	∃y.(X(y,j)∧C(y,j))	∃ elim 1.4,2-6

"If a guess peg matches the color of one of correct solution pegs, but they're not in the same column, then there exists a guess of the same color which can be made in a different column that matches the solution peg."

Since this proof deals with the implications of a single guess, the row of the guess has been omitted from the proof. So X(a,j) means a guess of color j in column a, and C(b,i) means that the correct solution has color i in column b.

Propositions:
actual j (a specific color j)

X(a,j) (a guess of color j in column a)
¬C(a,j) (the correct solution doesn't have color j in column a)

∃y.(C(y,j)) (there exists a column y where the solution has color j)
∀y.∀z.(X(y,z)) (a guess can be made with any color in any column)

Conclusion:
∃y.(X(y,j)∧C(y,j)) (there exists a guess of color j which has same column as color j in the solution)

Proof 2:

1:	actual a, actual i, X(a,i), C(a,i)	premises
2:	∀y.∀z.((X(y,z)∧C(y,z))→R(y,z))	premise
3:	∀y.(R(y,i)→(¬R(y,j)∧¬R(y,k)))	premise
4:	X(a,i)∧C(a,i)	∧ intro 1.3,1.4
5:	∀z.((X(a,z)∧C(a,z))→R(a,z))	∀ elim 2,1.1
6:	(X(a,i)∧C(a,i))→R(a,i)	∀ elim 5,1.2
7:	R(a,i)	→ elim 6,4
8:	R(a,i)→(¬R(a,j)∧¬R(a,k))	∀ elim 3,1.1
9:	¬R(a,j)∧¬R(a,k)	→ elim 8,7
10:	R(a,i)∧(¬R(a,j)∧¬R(a,k))	∧ intro 7,9

"If a peg in a guess matches the color and column of one of the solution pegs, then the next guess should have the same color, and not any other color, in the same column."

This proof relies on the relation between a previous guess and the next guess the solver will make. So X(n,c) represents the previous guess, and formula R(n,c) represents the next guess the solver will make of color c in column n. This proof also limits the scope of the number of colors. For the sake of this proof, Mastermind involves only 3 colors: i, j, and k.

Propositions:
actual a (a specific column a)
actual i (a specific color i)

X(a,i) (a guess peg of color i in column a)
C(a,i) (the correct solution has color i in column a)

∀y.∀z.((X(y,z)∧C(y,z))→R(y,z)) (for all columns y and all colors z, if a guess and the solution both have color z in column y, then the next guess will have color z in column y)

∀y.(R(y,i)→(¬R(y,j)∧¬R(y,k))) (for all columns y, if the next guess has color i in column y, then the next guess can't have color k and color j in column y)

Conclusion:
R(a,i)∧(¬R(a,j)∧¬R(a,k)) (the next guess can only have color i in column a; no other color in the next guess can be in column a)

Proof 3:

1:	actual j, actual k, X(a,i), X(b,k), X(c,j), C(a,i), C(b,j)	premises
2:	C(c,k), (X(a,i)∧C(a,i))→(R(a,i)∧(¬R(a,j)∧¬R(a,k)))	premises
3:	∀z.((X(b,z)∧(C(a,z)∨C(c,z)))→((R(a,z)∨R(c,z))∧¬R(b,z)))	premise
4:	∀z.((X(c,z)∧(C(b,z)∨C(a,z)))→((R(b,z)∨R(a,z))∧¬R(c,z)))	premise
5:	X(a,i)∧C(a,i)	∧ intro 1.3,1.6
6:	R(a,i)∧(¬R(a,j)∧¬R(a,k))	→ elim 2.2,5
7:	R(a,i)	∧ elim 6
8:	¬R(a,j)∧¬R(a,k)	∧ elim 6
9:	¬R(a,k)	∧ elim 8
10:	¬R(a,j)	∧ elim 8
11:	C(b,j)∨C(a,j)	∨ intro 1.7
12:	X(c,j)∧(C(b,j)∨C(a,j))	∧ intro 1.5,11
13:	(X(c,j)∧(C(b,j)∨C(a,j)))→((R(b,j)∨R(a,j))∧¬R(c,j))	∀ elim 4,1.1
14:	(R(b,j)∨R(a,j))∧¬R(c,j)	→ elim 13,12
15:	R(b,j)∨R(a,j)	∧ elim 14
16:	C(a,k)∨C(c,k)	∨ intro 2.1
17:	X(b,k)∧(C(a,k)∨C(c,k))	∧ intro 1.4,16
18:	(X(b,k)∧(C(a,k)∨C(c,k)))→((R(a,k)∨R(c,k))∧¬R(b,k))	∀ elim 3,1.2
19:	(R(a,k)∨R(c,k))∧¬R(b,k)	→ elim 18,17
20:	R(a,k)∨R(c,k)	∧ elim 19
21:	R(b,j)	assumption
22:	R(a,j)	assumption
23:	⊥	¬ elim 22,10
24:	R(b,j)	contra (constructive) 23
25:	R(b,j)	∨ elim 15,21-21,22-24
26:	R(a,i)∧R(b,j)	∧ intro 7,25
27:	R(a,k)	assumption
28:	⊥	¬ elim 27,9
29:	R(c,k)	contra (constructive) 28
30:	R(c,k)	assumption
31:	R(c,k)	∨ elim 20,27-29,30-30
32:	(R(a,i)∧R(b,j))∧R(c,k)	∧ intro 26,31

"If a guess is made which returns the feedback that two pegs are the correct color but in the wrong column, and the remaining guess pegs are the correct color and correct column, then the next guess can only be a guess which matches the solution and wins the game."

This proof uses a modified form of the conclusion found in the previous proof. When a guess peg matches the color and column of a solution peg, that implies that the next guess should have that correct color, and no other color, in that column. This proof is also limits the scope of mastermind to having only three columns, a, b, & c, and at least 3 colors, i, j, and k.

Propositions:
actual j, actual k (specific colors j, k)

X(a,i),X(b,k),X(c,j) (guess pegs, color i in column a, k in b, and j in c)
C(a,i),C(b,j),C(c,k) (solution pegs, color i in column a, j in b, and k in c)

$(X(a,i) \wedge C(a,i)) \rightarrow (R(a,i) \wedge (\neg R(a,j) \wedge \neg R(a,k)))$ (if a guess and the solution are both color a in column i, then the next guess should be color i, and no other color, in column a)

$\forall z.((X(b,z) \wedge (C(a,z) \vee C(c,z))) \rightarrow ((R(a,z) \vee R(c,z)) \wedge \neg R(b,z)))$ (for all colors z, if a guess peg is color z in column b, and the solution contains color z in a column other then b, then the next guess should have color z in a column other then b)

$\forall z.((X(c,z) \wedge (C(b,z) \vee C(a,z))) \rightarrow ((R(b,z) \vee R(a,z)) \wedge \neg R(c,z)))$ (for all colors z, if a guess peg is color z in column c, and the solution contains color z in a column other then c, then the next guess should have color z in a column other then c)

Conclusion:
$(R(a,i) \wedge R(b,j)) \wedge R(c,k)$ (The next Guess is a perfect match to the solution found in the propositions, and the game is won)

References

Knuth, D. (1976). The Computer as Mastermind. *Journal of Recreational Mathematics*, 9(1). https://www.cs.uni.edu/ wallingf/teaching/cs3530/resources/knuth-mastermind.pdf