Project Title:

"What's Missing?"

New Mexico

Supercomputing Challenge

Final Report

April 4, 2018

Team Members:

Jen Marie Phifer

Forest Good

Teacher:

Anne Loveless

Project Mentors:

Creighton Edington

Annette Hatch

Nick Bennett

James Taylor

Acknowledgements:

We would like to thank the following people for assisting us in our project:

Nick Bennett: For mentoring us and assisting us with the “Set-finding code” of our project and for encouraging us to delve deeper into some of the questions this project posed

Creighton Edington: For mentoring us and always assisting us with any questions we had and for guiding our project to higher levels

Annett Hatch: For mentoring us, helping us think about the different ways to approach this problem, and for always welcoming any questions we had

Anne Loveless: For sponsoring us

James Taylor: For helping us review the basic conceptual theories of this problem

Table of Contents

Introduction

Information is powerful; it is a retention of knowledge, an instructional guide, and a key to learning. Therefore, databases that hold information are crucial. However, what happens when pieces of data go missing? How would these pieces be recovered? Typically, if one missing piece of information is lost, it's not too hard to fill in. We would simply go through all the records, and based on what we have, we can fill in the missing "blanks." However, what happens when there are two missing pieces of information? At this point, the problem is that the missing pieces of information can be a combination of different things.

The question is, how do we determine what the missing pieces are? How do we use what we know to find what we don't know? To explore this problem, we looked at the card game SET. Specifically, we looked at situations when one or two cards went missing from the deck and how we could determine what those cards were without having to go through the whole deck.

Materials and Methods:

To explore this problem, we created a computer simulation using the software NetLogo.

The three general steps involved in solving this problem were:

1. Finding a way to create a program that plays the game
2. Finding a way to find what one missing card is
3. Finding a way to find what two missing cards are

We conducted our experiment by using NetLogo's build in "BehaviorSpace" function for our trials.

Regular Game Play

There are 81 cards, each with 4 attributes (number, color, shading, and shape) that each have 3 variations (3 different numbers, 3 different colors, 3 different shadings, and 3 different shapes). The variation for each attribute is as follows:

Attributes	Number	Color	Shading	Shape
Variation 1	1	Green	Open	Diamond
Variation 2	2	Purple	Hashed	Oval
Variation 3	3	Red	Solid	Squiggle

The game starts by placing 12 cards face up on the table. From there, players try to find a "SET," which is comprised of 3 cards in which the attributes have either all the same variation or all different variations. For example, take the layout below:

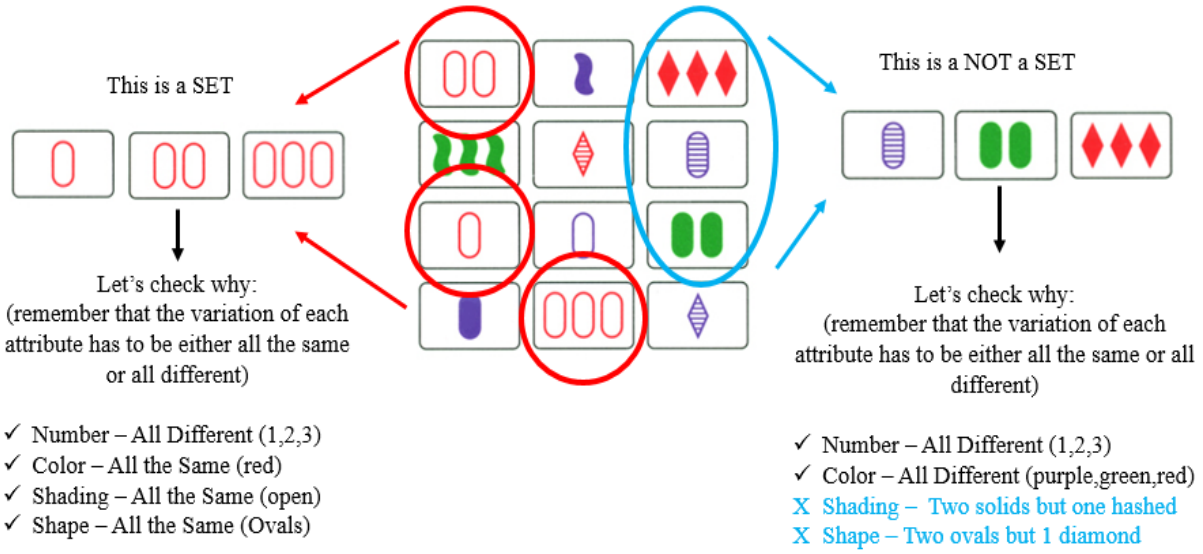


Figure 1: The SET graphics are taken from ("Harris Spiral"). The arrows, circles and explanations are the competitors'

Essentially, there are 4 different variation combinations for each attribute. Using “D” as an abbreviation for “Different” and “S” as an abbreviation for “Same,” the 4 combinations are:

- 4D
- 3D, 1S
- 2D, 2S
- 1D, 3S

**Note: The SET shown in Figure 1 was combination 1D, 3S. All the variations for one attribute (number) was different while the rest were all the same.*

*Hint/side note for newer players: The most common way most people find SETs is by pairing 2 cards together and looking for the third card that will complete the SET. This is known as the Fundamental Theorem of SET: “Given any pair of cards, there is a unique card that completes a SET with the pair” (Gordon, 13).

Once a player spots a SET, the player yells “SET” and proceeds to collect his or her cards. From there, the dealer deals 3 more cards. If there are no sets, the dealer deals out groups of 3 cards until there is a SET, going only up to a maximum of 21 cards on the table — research guarantees a SET at 20 cards (Klarreich). The players go through the whole deck and whoever has the most sets wins.

The Mathematics of a SET

A SET’s validity (whether it is a SET or not) can not only be visually checked, but also mathematically checked. For example, say the following cards were grouped as a SET:

- Card 1: 1 purple solid diamond

- Card 2: 3 red solid diamonds
- Card 3: 2 green hashed diamonds

Typically, a visual checklist is conducted for all the attributes: *Are all the number variations either all the same or all different? Are all the color variations either all the same or all different? Etc.* In this case, the three cards are not a SET because the shading is not all the same or all different. Visually, this would be easy enough to see.

However, a mathematical check can also be conducted. To do this, each attribute variation must be assigned a value of 0, 1, or 2. For this project, the following assignment variation was used:

Attributes	Number	Color	Shading	Shape
Value 0	3	Green	Open	Diamond
Value 1	1	Purple	Hashed	Oval
Value 2	2	Red	Solid	Squiggle

**Note: It does not matter how the values are assigned. For example, whether red is 2 instead of 0 is of no consequence.*

After an assignment is made, the sum of each attribute's variation must be totaled. If the modulo 3 (the remainder after dividing by 3) of each attribute's sum equals 0, then it is a valid SET. Looking at the previous card example (which was *not* a SET), if each card's attribute variation is added, the following result is obtained:

- Number sum: 3
- Color sum: 3
- Shading sum: 5

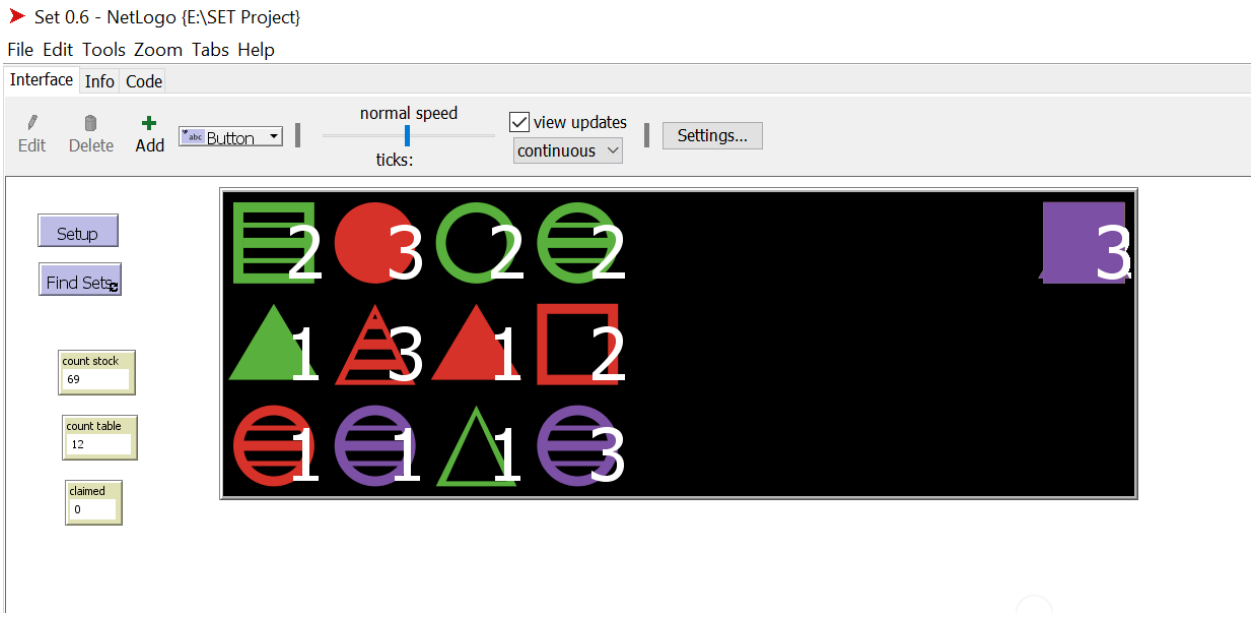
- Shape sum: 0

The modulo 3 of each sum with respect to the order above is: 0, 0, 2, and 0. Because the shading's modulo 3 is not equal to zero, it is not a valid SET. However, if Card 3 was replaced with 2 green *solid* diamonds, it would become a valid SET because the shading sum would change to 6, making the modulo 3 equal 0.

*Note: Every valid SET follows this mathematical rule.

Code

Using the mathematics of a SET, we created a program that played the game. The following is a screenshot of the first working version we created:



Graphically, it is slightly different than the original game. Instead of the shapes being oval, diamond, or squiggle, it is now circle, triangle, and square respectively. Furthermore, instead of having multiple shapes (1, 2, or 3) there is a label signifying the quantity.

The following is a screenshot of the set-finding code:

```
10
11  to-report is-a-set? [:candidates]                          ;; looks at all three attribute
12    report (                                                 ;; if yes, it is a set
13      ((sum [coloring] of :candidates) mod 3 = 0)
14      and ((sum [shading] of :candidates) mod 3 = 0)
15      and ((sum [form] of :candidates) mod 3 = 0)
16      and ((sum [number] of :candidates) mod 3 = 0))
17  end
18
19  to-report all-sets [:pool]
20    let :sets-found []                                                          ;; Creat
21    let :pool-list (sort :pool)
22    let :card1-list (sublist :pool-list 0 (length :pool-list - 2))              ;; Proce
23    let :card1-indexes (n-values (length :card1-list) [i -> i])                 ;; as a
24    (foreach :card1-list :card1-indexes [                                       ;; value
25      [:card1 :index1] ->                                                       ;; a "hc
26      let :card2-list (sublist :pool-list (1 + :index1) (length :pool-list - 1))  ;; all 3
27      let :card2-indexes (n-values (length :card2-list) [i -> i + 1 + :index1])
28      (foreach :card2-list :card2-indexes [                                     ;; For e
29        [:card2 :index2] ->                                                     ;; Cards
30        let :card3-list (sublist :pool-list (1 + :index2) (length :pool-list))  ;; the r
31        let :card3-indexes (n-values (length :card3-list) [i -> i + 1 + :index2])  ;; formi
32        (foreach :card3-list :card3-indexes [                                   ;; the s
33          [:card3 :index3] ->
34          let :candidates (turtle-set :card1 :card2 :card3)
35          if is-a-set? :candidates [
36            set :sets-found (lput :candidates :sets-found)
37          ]
38        ])
39      ])
40    ])
41    report :sets-found
42  end
```

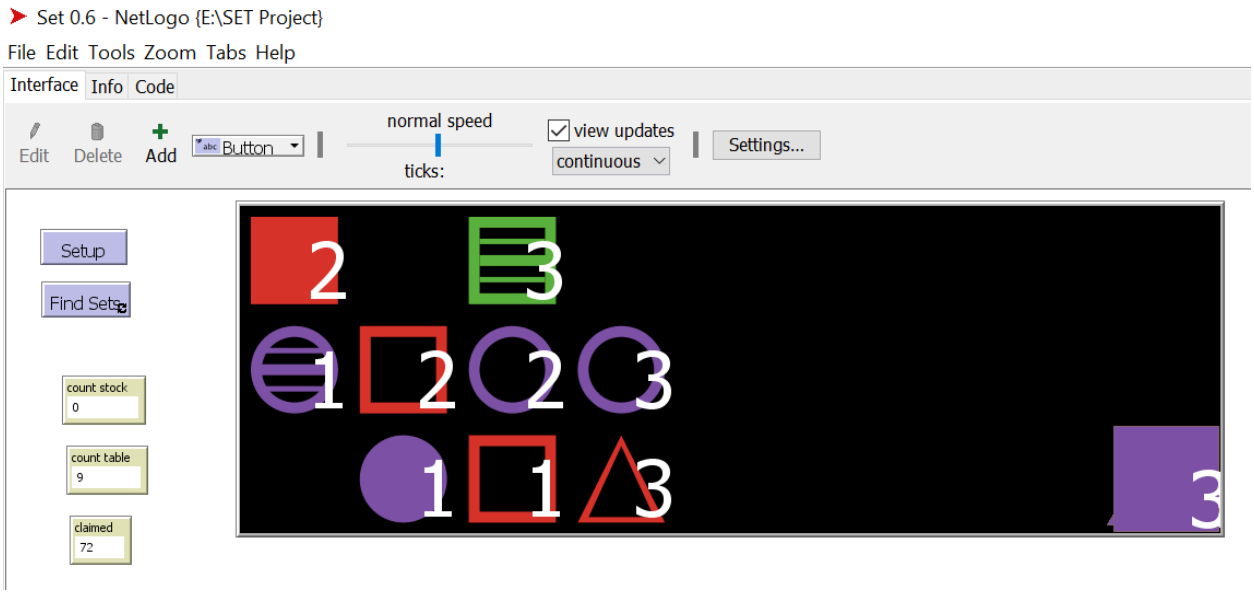
**Quick background* — In this code, there were 3 distinct variables we used to classify the cards: table, stock, and claimed. “Table” refers to the cards that have been dealt, “stock” refers to the deck, and “claimed” refers to the cards that have been grouped into SETs.

To find SETs, we used anonymous procedures. The 12 cards on the table are first placed into all the possible 3-card combinations. The variation of each attribute is then summed, and if the modulo 3 of each of the sums is 0, then it is considered a SET and placed into the list “sets-found.”

Once all the possible SETs are found, the program removes the first SET in the list and places them in the claimed category. More cards are then dealt to replace the cards claimed and

the process repeats, looping until there are no more possible SETs and no more cards left in the stock. If there is not a SET when 12 cards are on the table and there are still cards in the stock, it will deal more cards until there is a SET, going only up to 21 cards since there is guaranteed a SET at 20 cards (Klarreich).

The following is a screenshot of the program after it has been played:



**Note: When the game is played, there are typically cards left over that did not get grouped into a SET and cannot be grouped into a SET with one another; rarely are there no leftover cards.*

One Missing Card

Aside from the regular game play, there is a version called “End Game SET,” where the only difference is that a random card is taken out at the beginning of the game. After the card is taken out, the game is played normally. At the end, using the cards left over, an algorithm can be used to find the missing card.

To do this, a modified process of checking a SET’s validity is used. The value of each attribute’s variation must still be assigned, and the sum of each card’s attributes must also still be added. However, because we want to find the missing card, we must find the value (when added to the current sum) that causes the modulo 3 of each attribute’s total sum to equal 0, meaning that the total sum must equal a multiple of three to have a modulo 3 equal to zero. Once we know the values, we can use the assignment variation to find the corresponding attributes of the missing card.

**Note: The algorithm and method for finding one missing card has already been previously discovered.*

An example:

Say the following 5 cards are left:

- 3 red open squiggles
- 1 green solid oval
- 3 green solid diamonds
- 3 green open ovals
- 3 red solid ovals

The sum of all the cards' attribute is:

Number: 0 + 1 + 0 + 0 + 0 = 1

Color: 2 + 0 + 0 + 0 + 2 = 4

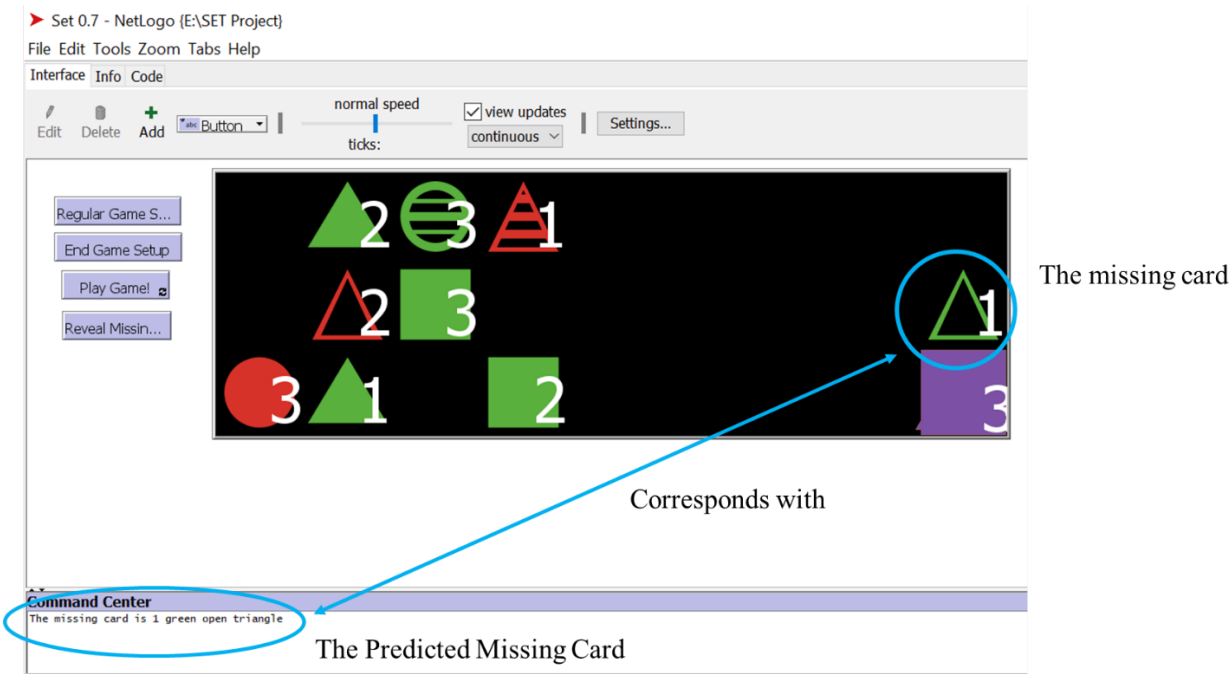
Shading: 0 + 2 + 2 + 0 + 2 = 6

Shape: 2 + 0 + 1 + 0 + 0 = 5

In order of the above attributes, the value that will cause the modulo 3 of each of the above sums to equal zero are: 2, 2, 0, 1. Looking at the assignment variation, the missing card is 2 red open ovals.

Code

We added the algorithm for finding a missing card into our program. The following screenshot is of the updated version:



```
131 to missing-card                                    ;; Finds the
132   let needed_number 0                              ;; Creates
133   let needed_coloring 0
134   let needed_shading 0
135   let needed_form 0
136   let total 0
137
138   set total (sum [number] of table)                ;; Sets the
139   let y 0
140   foreach [1 2 3]                                  ;; Finds the
141   [                                                ;; mod 3 of
142    ifelse ((y + total) mod 3 = 0)
143      [
144       set needed_number y
145      ]
146      [
147       set y (y + 1)
148      ]
149   ]
150
```

A random card is taken out at the beginning of the game. The game is then played normally. Using the cards left over, the variation value of each card's attribute is summed. In the screenshot above, the program tests all the values (0, 1, 2) to see what value, when added to the total value for number, gives back a modulo 3 equal to 0. This same algorithm is used to find the color, shading, and shape of the missing card.

Two Missing Cards

**Note: As mentioned before, the method for finding one missing card has already been discovered. However, a method for finding two missing cards has not been previously discovered.*

Problem Analysis

Finding one missing card is fairly easy. However, the problem is when we try to look at two missing cards. The possible combination choices are too ambiguous; there are too many ways to make the sum of the variations equal a modulo 3 of zero. For example, in a hypothetical situation, say two cards at the beginning of the game are taken out. After playing, one card is left, meaning there are two possible combinations that will return a modulo 3 equal to zero. In this case, assume the leftover card is 1 green hashed oval. The variation assignment would be (with the same order of number, color, shading, and shape): 0, 1, 1, 0.

For each attribute variation there can be two possibilities that give a modulo 3 equal to zero:

Number: 0 and 0 / 1 and 2

Color: 0 and 2 / 1 and 1

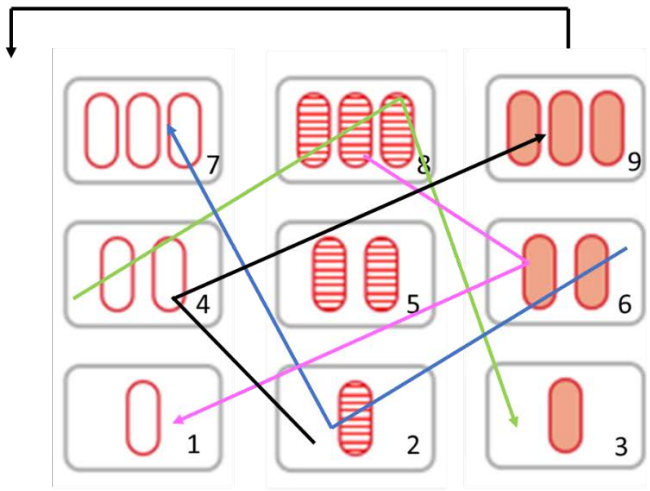
Shading: 1 and 1 / 0 and 2

Shape: 0 and 0 / 1 and 2

It would be difficult to narrow down the choices, and even it was narrowed down (say it's 0 and 0 instead of 1 and 2 for number), it is still unclear which card has which variations. For example, say the first possibility for each attribute is right. It is unclear if Missing Card 1 is (0, 0, 1, 0) or if it's (0, 2, 1, 0).

Problem Exploration and Solution

To tackle this problem, we tried simplifying it and broke it down to a group of 9 cards with the same shape and color. If lain out correctly, a hyperplane that shows the 12 different sets that can be made with the 9 cards is created (see Figure 2 below).



To the left, each line represents a set. However, in addition to the four sets outlined in the diagram, each row (3), column (3), and diagonal (2) also makes a set.

*Note: There is a "wrap around" effect. Let's label each column from left to right 1, 2, and 3 respectively (just like the card numbers on the bottom). If we shift column 3 to the left so it is before column 1, the set marked with a black line would align diagonally, following the consistent pattern.

Figure 2: The SET graphics are taken from ("Game of Set"). The arrows and explanations are the competitors'

A list of the 12 possible SETs:

[1 2 3]	[1 5 9]
[4 5 6]	[3 5 9]
[7 8 9]	[2 6 7]
[1 4 7]	[3 4 8]
[2 5 8]	[1 6 8]
[3 6 9]	[2 4 9]

Using the cards shown in Figure 2, if we only had 9 cards and 2 are missing, we would figure out what the missing cards are by grouping the cards that could form a SET together, as if playing a regular game. Then, we would look at all the possible set combinations and start drawing cards from the claimed pile. Using the cards that were drawn, we would begin

eliminating some of the 12 set possibilities. In our case (see Figure 3 below), we decided to make cards 2 and 5 the missing cards. In the best-case scenario, cards [1, 4, 7] would make a set and [9, 6, 3] would also make a set, leaving card 8 behind.

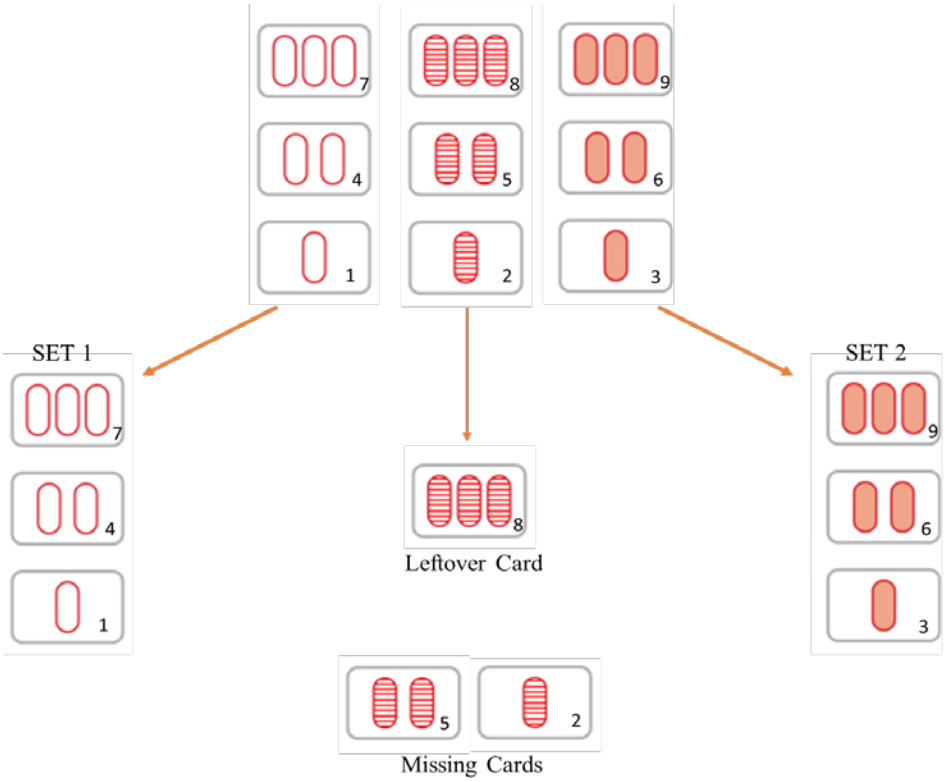


Figure 3: The SET graphics are taken from ("Game of Set"). The arrows and explanations are the competitors'

Now, when we look at the 12 possible sets that can be made, we must look for sets containing an 8 because the missing cards must form a SET with card 8 (we know this based on our hyperplane). Those sets are:

The 4 possible SETs:

[7 8 9] [2 5 8]

[3 4 8] [1 6 8]

We then go through the claimed cards to start eliminating the answer choices. For example, if card 7 was the first card that was revealed, then we can eliminate [7, 8, 9] as an answer choice. If card 3 was revealed next then we can eliminate [3, 4, 8]. If after that card 6 was revealed, then we can eliminate [1, 6, 8] and be left with [2, 5, 8]. That means the missing cards would have to be 2 and 5. However, this again is the best-case scenario in that no card in a set already eliminated appeared. For example, if card 4 was revealed after card 3 was revealed, it would not help because we had already eliminated set [3, 4, 8]. In other words, this all depends on how lucky we are when we draw the cards.

Overall, while this method did provide us with some valuable insight, it was difficult to apply it to a real game. To begin with, only having 9 cards that form a hyperplane is too convenient. In a real game, there are 81 cards — which makes 1080 total possible sets — and when we play the game, we are most likely not going to be left with only one card. Therefore, we would have great difficulty trying to narrow down which of the 1080 set possibilities holds the two missing cards. Furthermore, there is no guarantee that one of the cards left over will form a set with the two missing cards (like card 8 did in the previous example); the card that would complete the two missing cards' SET could have been grouped into another SET. In the previous example, we could have easily taken out SET [7, 8, 9] and had cards 1, 3, 4, and 6 leftover instead.

As a result, we were simply left to ponder for a while. However, we managed to realize something. Using the Fundamental Theorem of Set, we knew that for any pair of cards, there is only one card that makes the pair a SET. Therefore, we know that for the two missing cards, there is only one unique card that forms its SET. As we were looking at the mathematics of the

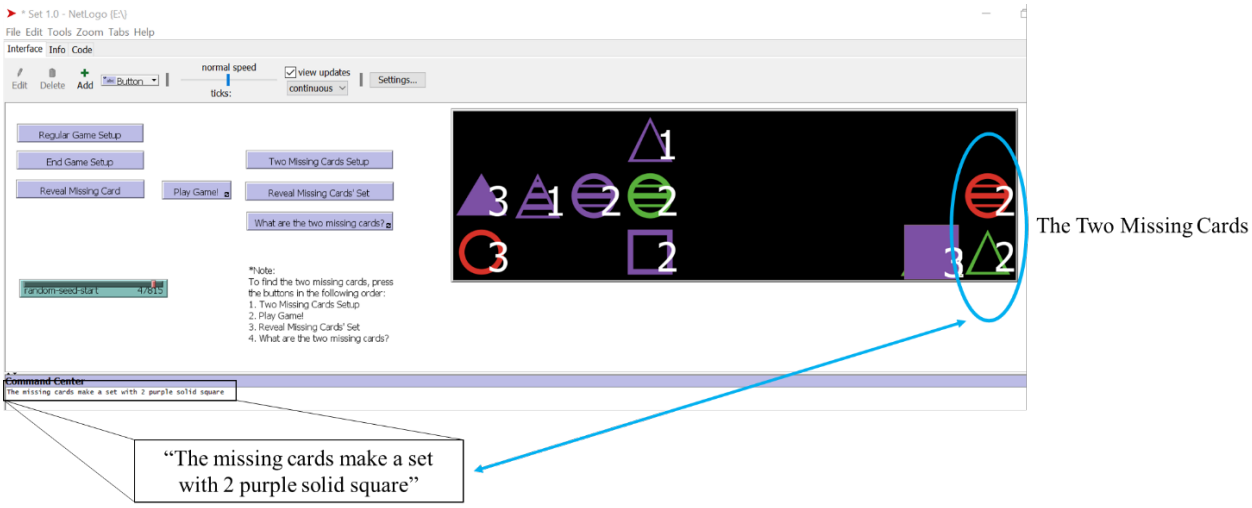
game, we saw that we could determine what this unique card was simply by “reversing” the formula used to find one missing card.

To determine the attribute of one missing card, we essentially look for the higher multiple of 3. However, in the case of two missing cards, to find the unique third card, we must look at the lower multiple of 3 instead. For example, for one missing card, if the sum of the color attribute was 10, then we would say the missing card’s color is red because the next multiple of 3 is 12, and we must add 2 to 10 get there; 2 corresponds to red. However, in the case of two missing cards, we would say that the unique third card’s color is purple because we look at the lower multiple of 3, which is 9. We must subtract 1 from 10 to get there, and 1 corresponds to purple.

The significance of this find is that if we know what that third card is, then instead of having to look through all 1080 set possibilities, we would only have to go through 40 set possibilities because each card can make 40 different sets (Gordon, 31). Furthermore, after narrowing it down to the 40 sets, we can then use the same method of drawing cards from the claimed pile to start narrowing down the 40 possible sets.

Code

The following is a screenshot of the updated version, version 1.0, we created that finds the 2 missing cards:



Before the two missing cards are determined, the unique third card is determined first. As mentioned previously, this is done by "reversing" the algorithm originally used to find one missing card.

Original Algorithm

```
131 to missing-card                                   ;; Finds the
132   let needed_number 0                             ;; Creates
133   let needed_coloring 0
134   let needed_shading 0
135   let needed_form 0
136   let total 0
137
138   set total (sum [number] of table)               ;; Sets the
139   let y 0
140   foreach [1 2 3]                                 ;; Finds the
141   [                                               ;; mod 3 of
142    ifelse ((y + total) mod 3 = 0)
143      [
144       set needed_number y
145      ]
146      [
147       set y (y + 1)
148      ]
149   ]
150
```

"Reversed" Algorithm

```
364 to missing-cards                                  ;; Finds
365   let needed_number2 0                            ;; Creat
366   let needed_coloring2 0
367   let needed_shading2 0
368   let needed_form2 0
369   let total 0
370
371   set total (sum [number] of table)               ;; Sets t
372   let y 0
373   foreach [1 2 3]                                 ;; Finds
374   [                                               ;; mod 3
375    ifelse ((total - y) mod 3 = 0)
376      [                                            ;; Change
377       set needed_number2 y                         ;; set v
378      ]
379      [
380       set y (y + 1)
381      ]
382   ]
383
```

Then, we created an algorithm that generated all the set possibilities for that one card (done by modifying the original set finding code as show below).

Original Set-finding Code

```
to-report is-a-set? [:candidates]                    ;; looks at all three attribute
  report (                                           ;; if yes, it is a set
    ((sum [coloring] of :candidates) mod 3 = 0)
    and ((sum [shading] of :candidates) mod 3 = 0)
    and ((sum [form] of :candidates) mod 3 = 0)
    and ((sum [number] of :candidates) mod 3 = 0))
end

to-report all-sets [:pool]
  let :sets-found []                                 ;; Creat
  let :pool-list (sort :pool)
  let :card1-list (sublist :pool-list 0 (length :pool-list - 2))   ;; Proce
  let :card1-indexes (n-values (length :card1-list) [i -> i])      ;; as a
  (foreach :card1-list :card1-indexes [              ;; value
    [:card1 :index1] ->                              ;; a "hc
    let :card2-list (sublist :pool-list (1 + :index1) (length :pool-list - 1))   ;; all
    let :card2-indexes (n-values (length :card2-list) [i -> i + 1 + :index1])
    (foreach :card2-list :card2-indexes [            ;; For e
      [:card2 :index2] ->                            ;; Cards
      let :card3-list (sublist :pool-list (1 + :index2) (length :pool-list))   ;; the r
      let :card3-indexes (n-values (length :card3-list) [i -> i + 1 + :index2])   ;; form
      (foreach :card3-list :card3-indexes [          ;; the
        [:card3 :index3] ->
        let :candidates (turtle-set :card1 :card2 :card3)
        if is-a-set? :candidates [
          set :sets-found (lput :candidates :sets-found)
        ]
      ])
    ])
  ])
  report :sets-found
end
```

Modified Set-finding Code

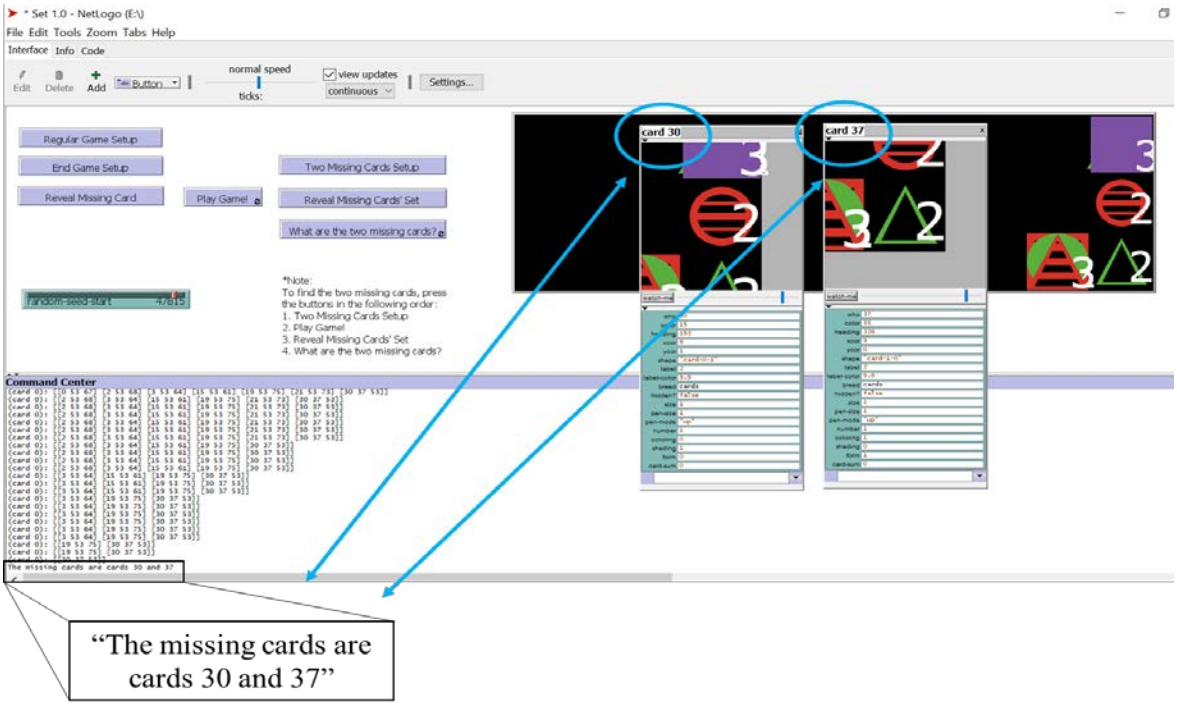
```
to-report all-sets2 [:pool]                                                  ;; Fo
  let :sets-found []
  let :pool-list (sort :pool)
  let :card1-list (sublist :pool-list 0 (length :pool-list - 2))
  let :card1-indexes (n-values (length :card1-list) [i -> i])
  (foreach :card1-list :card1-indexes [
    [:card1 :index1] ->
    let :card2-list (sublist :pool-list (1 + :index1) (length :pool-list - 1))
    let :card2-indexes (n-values (length :card2-list) [i -> i + 1 + :index1])
    (foreach :card2-list :card2-indexes [
      [:card2 :index2] ->
      let :card3-list (sublist :pool-list (1 + :index2) (length :pool-list))
      let :card3-indexes (n-values (length :card3-list) [i -> i + 1 + :index2])
      (foreach :card3-list :card3-indexes [
        [:card3 :index3] ->
        let :candidates (turtle-set :card1 :card2 :card3)
        ifelse is-a-set? :candidates and member? third_card [who] of :candidates [   ;; **To
          ifelse member? revealed_card [who] of :candidates
          [
            set :sets-found (fput sort [who] of :candidates :sets-found)             ;; if r
          ]
          [
            set :sets-found (lput sort [who] of :candidates :sets-found)             ;; othe
          ]

        ]
      ])
    ])
  ])
  report :sets-found
end
```

From there, the program starts eliminating the sets the leftover cards make. Once all the leftover cards' SET have been checked for, the program then draws cards from the claimed pile to start eliminating possibilities.

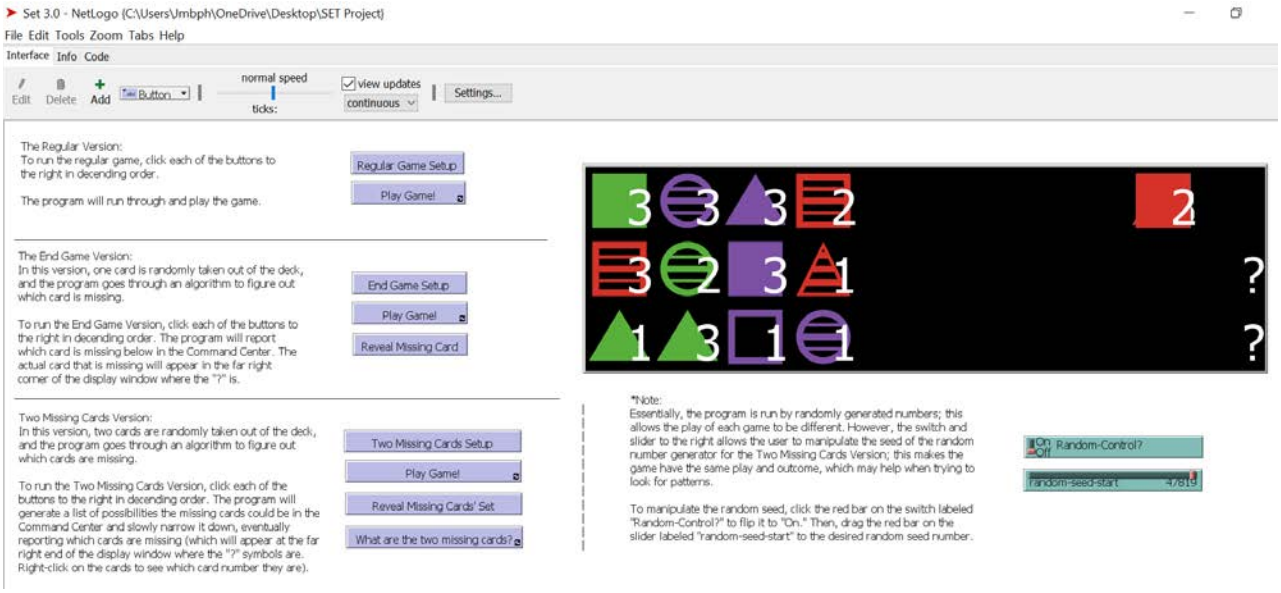
```
  ask one-of claimed
  [
   set revealed_card [who] of self
   setxy 9 2
  ]
 ]

 ask turtle 0
 [
 set stock (cards with [xcor <= 9])
 set all-1080-sets all-sets2 stock
 if repetition = 0
 [
  set all-1080-sets2 all-1080-sets
 ]
 set repetition repetition + 1
 let check []
 set check sublist all-1080-sets 0 1
 let check2 []
 set check2 item 0 check
 set all-1080-sets2 remove check2 all-1080-sets2
 show all-1080-sets2
 ]

 if length all-1080-sets2 = 1
 [
  let check3 []
  set check3 item 0 all-1080-sets2
  set check3 remove third_card check3
  type "The missing cards are cards " type item 0 check3 type " "
  ask cards with [xcor = 9 and ycor >= 0]
 [
  show-turtle
```

The code draws a random card from the claimed pile and sets it as the "revealed card." One of the turtles then checks to see if the revealed card is in one of the 40 possible SETs list, labeled "all-1080-sets2". If it is, the turtle removes the SET from the list. Once the list has been narrowed down to one SET, the program types out which cards are the missing cards.

Below is a screenshot of the interface at the end of the program's run:



Below is a screenshot of the "final version" (3.0) of our program. It's been cleaned up and organized to make it more user-friendly:



Results

The program was able to accurately determine what the two missing cards were. However, we wanted to determine how many claimed cards the program had to go through before it was able to determine what the two missing cards were. To do this, we ran the program 100 times through NetLogo's built-in BehviorSpace feature. This allowed us to record the average number of claimed cards left after each trial. On average, the number of claimed cards left was approximately 11. The number of cards on the table that did not make a SET was approximately 8. Therefore, it had to go through approximately 85% of the claimed cards before it was able to figure out which cards the missing cards were.

Discussion

By finding out which card the two missing cards group with to make a SET, we could determine the possible set combinations the two missing cards have to be in. From there, using the claimed cards, we could slowly start narrowing down the possibilities. Through this method, we only had to go through approximately 85% of the claimed cards. Questions about how helpful this is can vary.

Depending on how large the claimed cards (the equivalent of records/what we know in the real world) are, it can be fairly helpful or not helpful at all. For example, if our claimed cards stack was 10000 cards, then saving 15% would help quite a bit. We would only have to go through 8,500 cards, saving ourselves from looking through 1,500 of them. However, if our claimed cards stack was 10, then saving 15% would not be helpful, saving ourselves from having to go through about 2 cards.

Optimization

We were dissatisfied that there were 11 cards left in the claimed pile; it seemed like too low of a number. Therefore, we tried looking at alternate ways to narrow down which cards could be the missing cards.

We had two ideas in mind when we were looking at how we could optimize this process so we would have to go through less cards. One idea was to examine a cube so we could determine if the connections a SET can make can eliminate other SETs. A cube is three hyperplanes stacked on top of one another. When that happens, SETs can not only be made on one plane, but across planes as well. Figure 4 demonstrates a few different SETs the middle card on Plane 1 can make across all three planes.

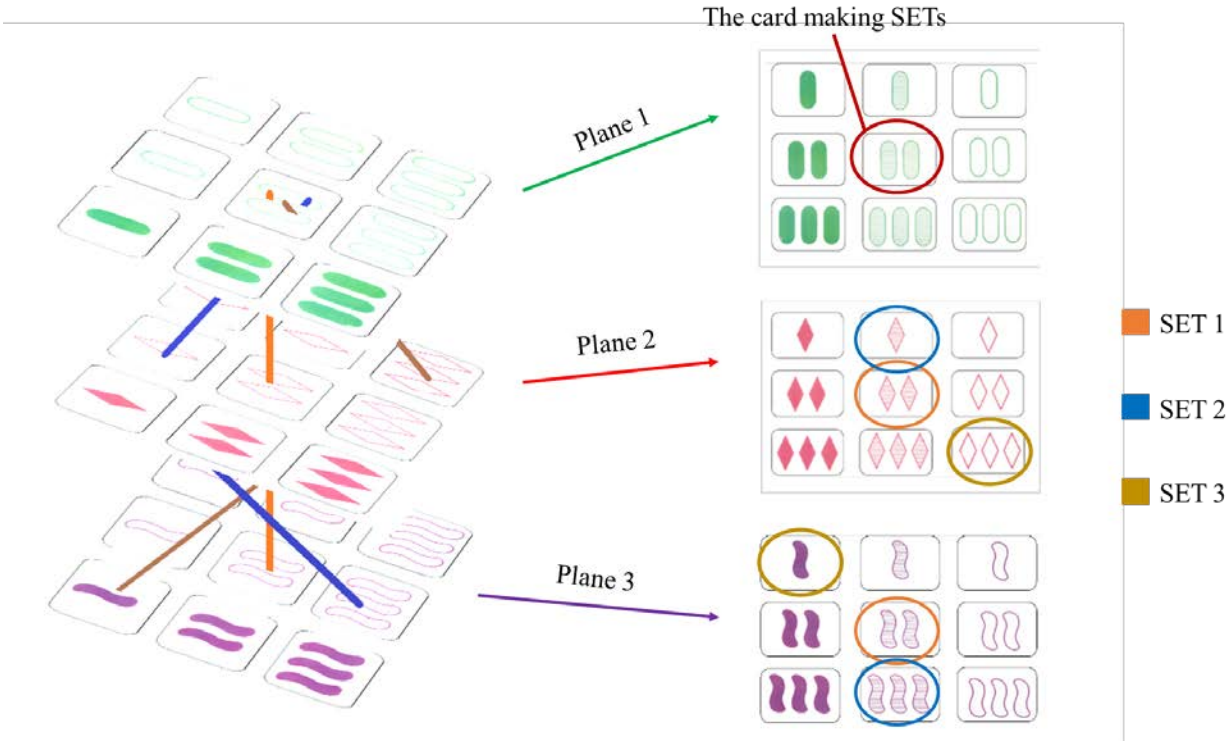


Figure 4: The SET graphics are cards that have been scanned from the original card game. The arrows, circles, and explanations are the competitors'

Figure 4 only outlines a few SETs one card can make. However, if all the SETs were "connected" in the figure above, there would be many lines. If one of those lines were to intersect with another line, could we eliminate the other SET connected to that line? That was one idea we had. However, as we were trying to test it, we came across incidents where we would have eliminated a SET that contained one of the missing cards. Therefore, we had to move on to our second idea.

The second idea was to look at probabilities. Instead of having a 100% accuracy rate where we can confirm which two cards are missing, we thought we could have a 70 – 90% accuracy rate instead. This way, we could say something similar to "Cards 1, 4, 7, 8" have an 80% chance of being the missing cards. This would likely increase the number of cards we wouldn't have to go through.

Doing some more research, we found the following table:

TABLE 2.1.
Final counts (including percentage of total deck).

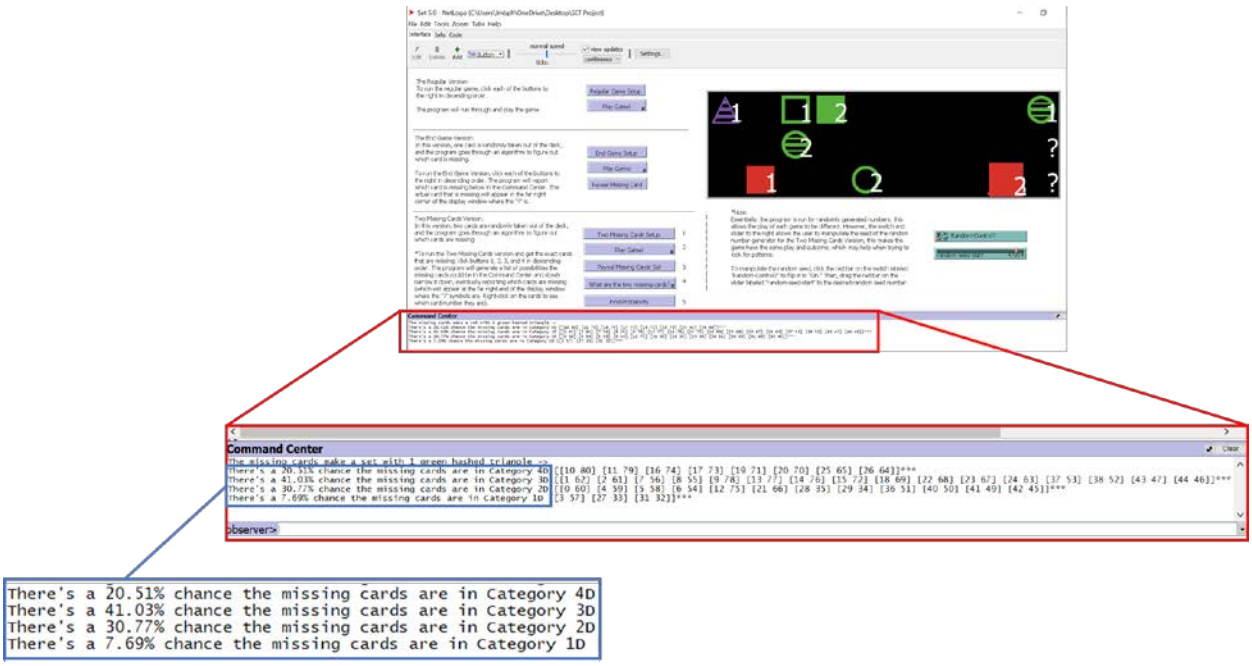
Kind of SET	*Number*	*Fraction*	*Percentage*
All different	216	$\frac{216}{1080}$	20%
Three different	432	$\frac{432}{1080}$	40%
Two different	324	$\frac{324}{1080}$	30%
One different	108	$\frac{108}{1080}$	10%
Total	1080	$\frac{1080}{1080}$	100%

Figure 5: Table taken from page 38 of The Joy of SET book listed in the bibliography

With reference to the explanation on page 6 of this report explaining the types of SETs that can be made (such as "4D," "3D," etc.), Figure 5 shows the probability for each type of SET, which we incorporated into our model.

Code

The following are screenshots of the optimized version of the program and part of its code:




```
800   set prob-length ((length D4-pt) + (length D3-pt) + (length D2-pt) + (length D1-pt))
801   type "There's a " type (precision (((length D4-pt) / prob-length) * 100) 2) type "% chance the missing cards are in Category 4D " type D4-pt
802   print "***"
803   type "There's a " type (precision (((length D3-pt) / prob-length) * 100) 2) type "% chance the missing cards are in Category 3D " type D3-pt
804   print "***"
805   type "There's a " type (precision (((length D2-pt) / prob-length) * 100) 2) type "% chance the missing cards are in Category 2D " type D2-pt
806   print "***"
807   type "There's a " type (precision (((length D1-pt) / prob-length) * 100) 2) type "% chance the missing cards are in Category 1D " type D1-pt
808   print "***"
809
```

The interface shows the probability of the missing cards being in each category. This is done by separating the 40 possible SETs into their respective categories then dividing the amount of SETs in each category by the total number of SETs. For example, when dividing the 40 SETs into their categories, the number of SETs initially in each category is as follows:

SET Category	Number of SETs	Explanation
4D	8	20% of 40 = 8
3D	16	40% of 40 = 16
2D	12	30% of 40 = 12
1D	4	10% of 40 = 4

When we divide the number of initial SETs in 4D by the total number of SETs, we would get 20% (corresponding to its initial percentage). Therefore, there is a beginning 20% chance the missing cards will be in the 4D category. To narrow down which category the missing cards are in, we follow the same algorithm of drawing a card from the claimed pile to eliminate SETs. As SETs are eliminated, the number of SETs in each category will decrease, therefore also decreasing or increasing the chances the missing cards are in each category. For example, let's say the first card we draw from the claimed pile is in category 4D. That would mean there are now 7 SETs in 4D and 39 total SETs. Therefore, if we divide 7 by 39, we would get a percentage of approximately 17.95; its chances have decreased. However, the chances of the missing cards being in the other categories will increase because their SET number stayed the same while the total number of SETs decreased.

Trial, Results, and Discussion

We ran the model in BehaviorSpace 1000 times. This yielded a significant amount of data. However, due to our limited time constraint, we were not able to conduct an in-depth analysis of our data. Nevertheless, skimming through some of the data, we did not see any obvious patterns that stood out to us. However, before looking through our data, we hoped that the 1D column could provide us with some useful information. Because the chances of the missing cards being in that category was so low, we expected that after we drew a certain number of cards and 1D had not been completely eliminated yet, then it had a high likelihood of being the missing card. We did not have the chance to truly try to prove this theory, but we did see some trends to suggest this. Overall, we have the intention of seeking mathematicians who would be willing to help us analyze our data.

Applications and Final Conclusions

Because finding missing pieces of information is crucial, there are many potential applications for this project. For fields such as accounting or logistics, missing pieces of data can be a hassle to deal with. Take accounting for example. What happens if money, receipts, or invoices are missing? Could we use the method for finding a missing card in this field to find the missing pieces of information on those receipts or invoices? If we managed to find a way to scale it up, perhaps we can.

Additionally, this might be applied to cybersecurity as well. Missing pieces of data can be vital in dealing with threats. Often times, there are known variables and unknown variables that must be taken into account of. There are the *knowns* (what we know), the *unknowns* (what we don't know), the *known unknowns* (what we know we don't know), and the *unknown unknowns* (what we don't know we don't know). In the case of the card game SET, when we had one or two missing cards, we realized that we had a known unknown (we knew that we were missing 'x' number of cards but didn't know which card(s) it was). However, we managed to use our knowns (cards left over or the cards we had) to find our unknown (the missing card(s)).

To conclude, we were able to accurately determine what the two missing cards were by generating the possibilities they could be then using the information we had to narrow down those possibilities. We were not able to analyze our optimization method. However, our overall objective of this project was to look at a way to create a systemized method for finding missing pieces of data.

Bibliography

End Game SET. 2017. Web. <https://www.setgame.com/end-game-set>.

Game of Set. 8 August 2013. Web. <https://whieldon.wordpress.com/2013/08/08/game-of-set/>.

Gordon, Gary, et al. *The Joy of SET*. Princeton: Princeton University Press, 2017. Print.

Harriss Spiral, Math Snacks, and SET. 15 Jan 2015. Web. <https://mathmunch.org/2015/01/15/harriss-spiral-math-snacks-and-set/>.

Klarreich, Erica. *Simple Set Game Proof Stuns Mathematicians*. 31 My 2016. Web. <https://www.quantamagazine.org/set-proof-stuns-mathematicians-20160531/>.

What is modular arithmetic? 2017. Web. <https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic>.