– Independent Work Report Spring, 2017 –

# Solving the Generalized Form of the Game of Set Efficiently

Steven Takeshita
Adviser: Zachary Kincaid

## Abstract

*This paper examines three implementations of a solver to efficiently find sets in the generalized version of the Game of Set. The three techniques are brute force, SMT solver, and dynamic algorithm. The brute force solution is ruled out, as its growth is exponential on small test cases. Through timing tests, it is shown that though a reduction to the SMT solver should create an efficient solution to the NP complete game, the dynamic algorithm approach yields the fastest solver for relatively smaller test cases. Interestingly, the order of growth of the SMT solver follows a linear progression, whereas the dynamic algorithm exponentially grows as parameters are increased, most likely due to its heavy memory usage. At high numbers of values and properties, the SMT solver yields a more efficient solver than the dynamic algorithm.*

## 1. Introduction

### 1.1. Motivation and Goal

The Game of Set was created in 1974 and published in 1991. The game consists of $3^4 = 81$ unique cards. Each card has four properties and one of three values. A valid set of three cards is one in which for each property, they all either have the same or different values. See Figure 1 for an overview of the game and to see how the properties and values are visually represented for all cards. At the beginning of the game, 12 cards are shown, and players must locate valid sets. Once a set is found or no set exists, three new cards are added. The person who collects the most sets wins.

When I used to play this game with my family, I had difficulty finding sets and always lost to my much faster sister. A natural extension of this game is to determine how a computer can find sets. The game itself is relatively limited with only three values and four properties, and therefore

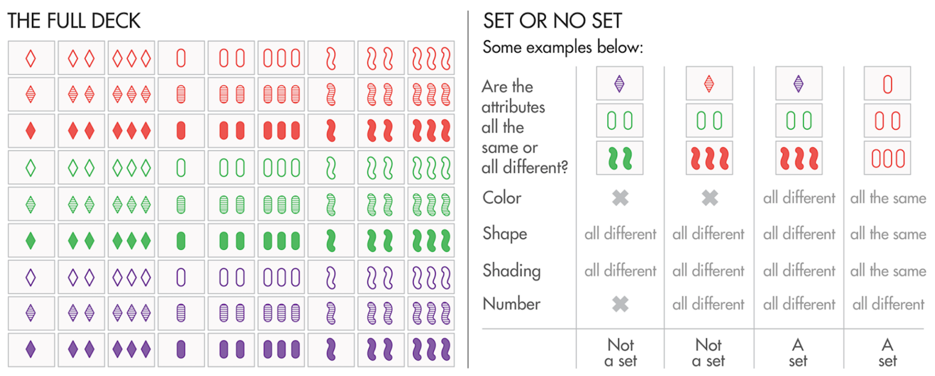


**Figure 1: Brief Overview of the Game of Set with $v = 3$ and $p = 4$ [8]**

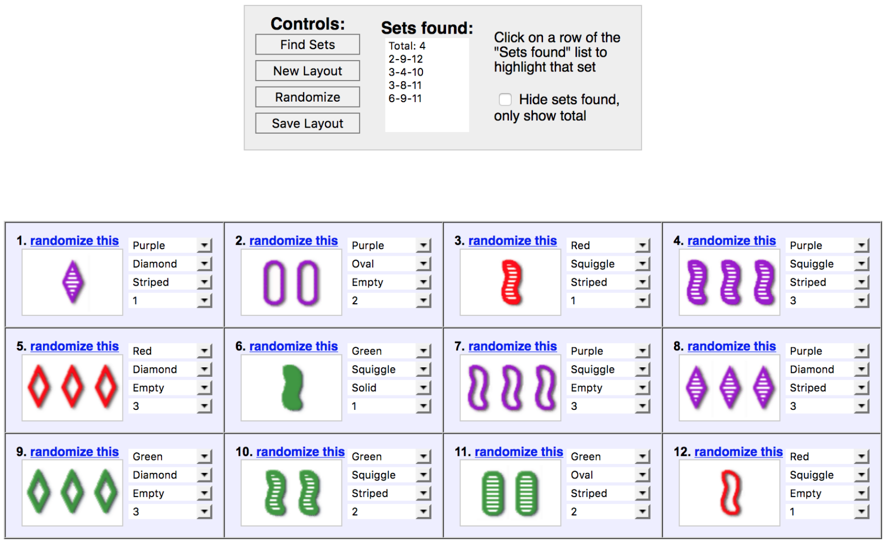
generalizing the number of values and properties will make the game much more difficult for a computer to solve. The Game of Set is an interesting problem for dynamic algorithms in that when three new cards are added, the algorithm should be able to build off of previous knowledge. A way to utilize this past information could be helpful in applying similar principles to other problems that could lend itself to dynamic programming. The goal of my project is to create a solver that locates $n$ sets efficiently in practice for a Game of Set with $p$ properties and $v$ values.

### 1.2. Problem Definition

A deck will contain $v^p$ possible cards where the cards have $p$ properties and $v$ values. Initially, $v * p$ cards will be outputted as the starting layout. Identify an arbitrary set of $v$ cards, in which for all properties the values are either the same or all different. Remove this set of size $v$ and count it as another set found, or if no set exists, add $v$ more cards from the remaining $v^p - v$ cards. Find a total of $n$ sets, where $n \leq v^{p-1}$ and each time a set is found, $v$ cards will be immediately added. Therefore, there are 2 update functions that the algorithms must handle: add $v$ cards to the board and remove $v$ cards that formed a set.

## 2. Problem Background and Related Work

The Game of Set has been researched thoroughly for education, but little research exists in regards to solving the generalized version. The generalized version of Set is in fact NP-Complete through a reduction from perfect-Dimensional Matching, a known NP-Hard problem [1]. However, this paper does not solve the problem in practice. On the Internet, solvers do exist and can be readily found. For example, this online JavaScript solver takes as input the cards that appear on the board and outputs the total number of sets that exist on the board [5]. The solver has a great UI as seen in Figure 2. However, this solver can only find a set within the given board and outputs overlapping sets which are not valid sets, as can be seen below (sets 3-8-11 and 6-9-11 both contain card 11).



Sets found:

```
Total: 4
2-9-12
3-4-10
3-8-11
6-9-11
```

Click on a row of the "Sets found" list to highlight that set

☐ Hide sets found, only show total

| | | | | |
|---|---|---|---|---|
| 1. randomize this | Purple | Diamond | Striped | 1 |
| 2. randomize this | Purple | Oval | Empty | 2 |
| 3. randomize this | Red | Squiggle | Striped | 1 |
| 4. randomize this | Purple | Squiggle | Striped | 3 |
| 5. randomize this | Red | Diamond | Empty | 3 |
| 6. randomize this | Green | Squiggle | Solid | 1 |
| 7. randomize this | Purple | Squiggle | Empty | 3 |
| 8. randomize this | Purple | Diamond | Striped | 3 |
| 9. randomize this | Green | Diamond | Empty | 3 |
| 10. randomize this | Green | Squiggle | Striped | 2 |
| 11. randomize this | Green | Oval | Striped | 2 |
| 12. randomize this | Red | Squiggle | Empty | 1 |

**Figure 2: Javascript SET Game Solver by Steve Nolte [5]**

Some solvers even employ image processing to identify sets [4]. These more advanced programs parse the cards from an image using computer vision to determine sets. Again, this solver only supports use for one iteration of the game and will need to be updated manually to load in new cards. These solvers are constrained to have four properties and three values as they only apply to the original Game of Set. Moreover, all these solvers restrict their mechanism to the brute force solution. They iteratively check all possible combinations of cards to determine satisfying sets.

## 3. Approach

### 3.1. SMT Solver

Satisfiability modulo theories (SMT) are efficient solvers for constraint satisfaction problems (CSP). CSP problems arise in many areas from software verification to graph problems, and need an efficient solver to determine whether the variables and constraints are satisfiable. The most basic CSP problem is propositional satisfiability (SAT), which takes in a logical formula of boolean variables and outputs whether it is possible to satisfy the formula assigning the variables to either true or false. SMT solvers provide a more natural and richer language to encode the formula. For example, it is possible to use variables that are integers or real numbers and the constraints can be arithmetic constraints rather than a pure logical formula. The SMT solver will output whether the set of constraints can be satisfied with some assignment of the variables and can output these variables if satisfied. The basis for these SMT solvers is the use of an approach called systematic search. The search space is represented as a tree with each branch representing an assignment of a variable. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is used to efficiently search this space, and speedups can be gained by restricting this search tree as much as possible to reduce the number of branches that the algorithm must traverse [2].

Generalizing the Game of Set has been done academically, but no practical solution to locate sets exists. My first implementation to solve this problem is through a reduction to SMT. SMT solvers are an efficient solver in practice for an NP-Complete problem, and I leverage this speed to solve the generalized Game of Set. Below is my reduction to SMT from the Game of Set to verify and find whether a set exists in a given board with $v$ values and $p$ properties.

## 3.2. Reduction to SMT

**3.2.1. Set Up** There exists $v * p$ cards forming a starting board $B$, but in general, there will be $n$ cards on a board at any moment. These cards can be represented as vectors where all entries $b_{i,j} \in \{0, 1, ..., v-1\}$:

$$B = \begin{bmatrix} b_{1,1} \\ b_{1,2} \\ \vdots \\ b_{1,p} \end{bmatrix} \begin{bmatrix} b_{2,1} \\ b_{2,2} \\ \vdots \\ b_{2,p} \end{bmatrix} \cdots \begin{bmatrix} b_{n,1} \\ b_{n,2} \\ \vdots \\ b_{n,p} \end{bmatrix} \tag{1}$$

The SMT Solver will then attempt to find a set of $v$ cards, labelled as $K$. The satisfying set can be denoted as vectors where $k_{i,j} \in \{0,1,...,v-1\}$:

$$K = \begin{bmatrix} k_{1,1} \\ k_{1,2} \\ \vdots \\ k_{1,p} \end{bmatrix} \begin{bmatrix} k_{2,1} \\ k_{2,2} \\ \vdots \\ k_{2,p} \end{bmatrix} \cdots \begin{bmatrix} k_{v,1} \\ k_{v,2} \\ \vdots \\ k_{v,p} \end{bmatrix} \tag{2}$$

**3.2.2. All Different or All Same Constraint** The first set of constraints will only be satisfied when the cards found in $K$ represent a set, where for all properties, they have either the same or all different value. This constraint can be written as two cases for each property of the cards:

**The values are all the same for a given property i:**

$$(k_{1,i} = k_{2,i}) \land (k_{2,i} = k_{3,i}) \land ... \land (k_{v-1,i} = k_{v,i}) \tag{3}$$

$$\bigwedge_{m=1}^{v-1} k_{m,i} = k_{m+1,i} \tag{4}$$

**The values are all different for a given property i:**

$$((k_{1,i} \neq k_{2,i}) \land (k_{1,i} \neq k_{3,i}) \land ... \land (k_{1,i} \neq k_{v,i})) \\ \land ((k_{2,i} \neq k_{3,i}) \land (k_{2,i} \neq k_{4,i}) \land ... \land (k_{2,i} \neq k_{v,i})) \land ... \land (k_{v-1,i} \neq k_{v,i}) \tag{5}$$

$$\bigwedge_{m=1}^{v-1} \bigwedge_{j=m+1}^{v} k_{m,i} \neq k_{j,i} \tag{6}$$

Therefore, we can write more concisely that for all properties of the cards, the values must all be the same or all different:

$$\bigwedge_{i=1}^{p}\left(\left(\bigwedge_{m=1}^{v-1}\bigwedge_{j=m+1}^{v} k_{m,i} \neq k_{j,i}\right)\bigvee\left(\bigwedge_{m=1}^{v-1} k_{m,i} = k_{m+1,i}\right)\right) \tag{7}$$

**3.2.3. The Cards Must Be From The Board** The second set of constraints is that the cards selected in the set $K$ must all be from the board $B$, which is of size $n$. This constraint can be encoded into the SMT solver as:

A given card $i$ from $K$ must be from the board $B$:

$$\begin{multline}((k_{i,1} = b_{1,1}) \wedge (k_{i,2} = b_{1,2}) \wedge \ldots \wedge (k_{i,p} = b_{1,p}))\vee \\ ((k_{i,1} = b_{2,1}) \wedge (k_{i,2} = b_{2,2}) \wedge \ldots \wedge (k_{i,p} = b_{2,p})) \vee \ldots\vee \\ ((k_{i,1} = b_{n,1}) \wedge (k_{i,2} = b_{n,2}) \wedge \ldots \wedge (k_{i,p} = b_{n,p})) \tag{8}\end{multline}$$

Therefore, all cards $i \in K$ must be from the board $B$:

$$\bigwedge_{i=1}^{v}\left(\bigvee_{j=1}^{n}\left(\bigwedge_{m=1}^{p} k_{i,m} = b_{j,m}\right)\right) \tag{9}$$

**3.2.4. All Distinct Cards** We constrain the possible cards in the set to be from $B$, but this includes using duplicate cards. A possible set could be $v$ of the exact same card and would satisfy the above constraints but does not represent a real set in the game. Therefore, the cards selected to be in $K$ must all be distinct cards. The cards of a set are considered all distinct if for any two cards, they have at least one property that has a differing value. The constraint can be written as:

A given card, $i$, differs with respect to at least one property compared to all cards after $i$ :

$$((k_{i,1} \neq k_{i+1,1}) \vee (k_{i,2} \neq k_{i+1,2}) \vee \ldots \vee (k_{i,p} \neq k_{i+1,p})) \wedge$$
$$((k_{i,1} \neq k_{i+2,1}) \vee (k_{i,2} \neq k_{i+2,2}) \vee \ldots \vee (k_{i,p} \neq k_{i+2,p})) \wedge \ldots \wedge$$
$$((k_{i,1} \neq k_{v,1}) \vee (k_{i,2} \neq k_{v,2}) \vee \ldots \vee (k_{i,p} \neq k_{v,p})) \quad (10)$$

Written more succinctly:

$$\bigwedge_{i=1}^{v-1} \left( \bigwedge_{j=i+1}^{v} \left( \bigvee_{m=1}^{p} k_{i,m} \neq k_{j,m} \right) \right) \quad (11)$$

**3.2.5. Symmetry Breaking** The first constraint I used to break the symmetry of the search space is for every possible set, the cards need to be in sorted order by their first property, but if the values are all the same, then sorted by the second property, and so forth if the cards in the set have equivalent value for a given property. This symmetry breaking constraint is labelled as the relative sorted constraint and can be encoded as follows:

$$(k_{1,1} \leq k_{2,1} \leq \ldots \leq k_{v,1}) \wedge ((k_{1,2} \leq k_{2,2} \leq \ldots \leq k_{v,2}) \vee \neg(k_{1,1} = k_{2,1} = \ldots = k_{v,1}))$$
$$\wedge ((k_{1,3} \leq k_{2,3} \leq \ldots \leq k_{v,3}) \vee \neg(k_{1,2} = k_{2,2} = \ldots = k_{v,2}) \vee \neg(k_{1,1} = k_{2,1} = \ldots = k_{v,1}))$$
$$\wedge \ldots \wedge ((k_{1,p} \leq k_{2,p} \leq \ldots \leq k_{v,p}) \vee \neg(k_{1,p-1} = k_{2,p-1} = \ldots = k_{v,p-1}) \vee \ldots \vee \neg(k_{1,1} = k_{2,1} = \ldots = k_{v,1}))$$
$$(12)$$

Written more compactly:

$$\bigwedge_{i=1}^{p} \left( (k_{1,i} \leq k_{2,i} \leq \ldots \leq k_{v,i}) \bigvee_{j=1}^{i-1} \left( \neg(k_{1,j} = k_{2,j} = \ldots = k_{v,j}) \right) \right) \quad (13)$$

The above symmetry breaking constraint condenses the search tree to a certain degree, but still searches some branches unnecessarily. For example, if a set is cards [0,1,2], the solver could potentially first check [1,2], then add 0 at the end when it would finally realize that this is not sorted. The algorithm would waste time as we know the first card must have the lowest value, 0, to be considered sorted. Therefore, we can rigidly constrain it such that for the first property in which they differ, the first card's value must be 0, the second card's 1, and so forth. This restricts the search space further. This symmetry breaking constraint is labelled as the rigid sorted constraint and can be encoded as follows:

$$
\begin{gathered}
(k_{1,1}=0,k_{2,1}=1,...,k_{v,1}=v-1)\wedge((k_{1,2}=0,k_{2,2}=1,...,k_{v,2}=v-1)\vee\neg(k_{1,1}=k_{2,1}=...=k_{v,1})) \\
\wedge((k_{1,3}=0,k_{2,3}=1,...,k_{v,3}=v-1)\vee\neg(k_{1,2}=k_{2,2}=...=k_{v,2})\vee\neg(k_{1,1}=k_{2,1}=...=k_{v,1})) \\
\wedge...\wedge((k_{1,p}=0,k_{2,p}=1,...,k_{v,p}=v-1)\vee\neg(k_{1,p-1}=k_{2,p-1}=...=k_{v,p-1})\vee...\vee\neg(k_{1,1}=k_{2,1}=...=k_{v,1}))
\end{gathered}
\tag{14}
$$

Written more compactly:

$$
\bigwedge_{i=1}^{p}\left((k_{1,i}=0,k_{2,i}=1,...,k_{v,i}=v-1)\bigvee_{j=1}^{i-1}\left(\neg(k_{1,j}=k_{2,j}=...=k_{v,j})\right)\right) \tag{15}
$$

I will explore the relative effectiveness of both strategies in the implementation section.

By building these four constraints (7, 9, 11, and 13 or 15) from a given board, we can reduce finding an arbitrary set to SMT and use a solver to locate a set efficiently.

**3.2.6. Update Functions** The solver to find sets must also support two update functions. The first update function of removing cards can be easily encoded into the SMT constraints. Let set $V$ be an arbitrary set that was located by the SMT solver. We can add new constraints such that the new set to be found cannot be equal to any of the cards found in $V$, where $V$ contains $v$ cards. Again, to be a distinct card, at least one property must be differing.

A given card $i$ from $K$ must not be equivalent to a card in $V$:

$$\begin{aligned}((k_{i,1} \neq v_{1,1}) \vee (k_{i,2} \neq v_{1,2}) \vee \ldots \vee (k_{i,p} \neq v_{1,p}))\wedge \\ ((k_{i,1} \neq v_{2,1}) \vee (k_{i,2} \neq v_{2,2}) \vee \ldots \vee (k_{i,p} \neq v_{2,p})) \wedge \ldots \wedge \\ ((k_{i,1} \neq v_{v,1}) \vee (k_{i,2} \neq v_{v,2}) \vee \ldots \vee (k_{i,p} \neq v_{v,p}))\end{aligned} \tag{16}$$

All cards $i \in K$ must not be equivalent to a card in $V$:

$$\bigwedge_{i=1}^{v} \left( \bigwedge_{j=1}^{v} \left( \bigvee_{m=1}^{p} k_{i,m} \neq v_{j,m} \right) \right) \tag{17}$$

For the second update function in which new cards may be added to the deck, we will need to create a new set of constraints (7, 9, 11, and 13 or 15) from above to locate a complete set. Therefore, this is represents a full reduction to SMT from the Game of Set.

The SMT solver might not locate a set within reasonable time in practice if we consider the the poly-time transformation and the fact that it needs to recalculate a set every time new cards are added. Therefore, I also explore the use of a dynamic algorithm to find sets.

## 3.3. Dynamic Algorithm

The dynamic algorithm approach uses previous knowledge of the board between draws to quickly discover sets. This approach will be useful because of the nature of the game's changing probabilities as it is played. It has been shown that on an initial layout of the game with 12 cards, the ratio of games in which there exists a set to those that do not contain a set, is 29:1. But as the game plays on, the ratio drops to 15:1, a 50% less chance of finding a set. For 15 cards, the ratio drops to 4% of its original ratio as the game is played [6]. Therefore, as the game is iteratively played in which sets are taken out, the game becomes immensely harder. Finding a high number of sets $n$ will be hard if many cards must be drawn. Remembering partial sets across draws can greatly speed up the algorithm if we know that it will take many draws to find complete sets. A similar symmetry breaking constraint from the SMT solver will be used to store only one set that all permutations of a given partial set will condense to. The dynamic algorithm solver I build should be able to beat the SMT solver for some cases of $v, p$, and $n$ considering the added difficulty of the game as we add new cards.

## 4. Implementation

The randomizer to create cards and solvers were all written in Python. Considering Python's lightweight feel and ease of iterating over all variables in a list, it is the natural choice to represent a deck and board of cards, where we must iterate over them regularly.

### 4.1. Randomizer

The randomization algorithm for creating a randomized deck is very important in terms of creating real gameplay. The algorithm must uniformly distribute all of the cards so that no solver can have an unfair advantage knowing what cards are more likely to appear. I used the Random library in Python to generate cards. To ensure the cards drawn were unbiased, I employed the use of a modified version of Fisher Yates Shuffle. Fisher Yates Shuffle works by iteratively picking random indices from a finite sequence and produces an unbiased permutation [3]. To convert the algorithm for use on a deck, I used rejection sampling to determine new cards to add to the board. I created a card by randomly generating a string of length $p$ and, for each property, randomly generating a value from 0 to $v-1$. If the card was not in the board already and not in the removed cards set (ie. part of a set that was already found), I added it. If it was a duplicate of a previous card, I generated another card until I created $v*p$, for the initial board, or $v$, for a draw, cards that are all distinct. Since the algorithm uses a randomized strategy but will always produce a board whose cards are all distinct and from the deck with equal probabilities, it is an example of a Las Vegas algorithm. To support the update function of removing cards, I appended those cards to a set of removed cards that cannot be used for new cards. This strategy yields a randomized $v*p$ cards for the starting board and unbiased drawing of $v$ cards from the deck.

To confirm that this in fact created random initial boards and $v$ new cards, I graphed the distribution of the beginning board for 1000 trials with two different cases, $v = 4$ and $p = 3$ and $v = 5$ and $p = 4$. I then used numpy and matplotlib to count all the occurrences of each card and graph them as a histogram. In Figures 3 and 4, we can see that in both cases (3 properties, 4 values and 4 properties, 5 values), the distribution of cards that were initially selected to be on the board form a uniform distribution over 10,000 trials.

The distribution of cards drawn (excluding the starting board and only counting the $v$ new cards drawn) also forms a uniform distribution as can be seen in Figures 5 and 6, when the boards have 3 properties, 4 values and 4 properties, 5 values over 10,000 trials. Therefore, this confirms that the implementation of the Fisher Yates shuffle in creating a perfectly random board and draws was successful.

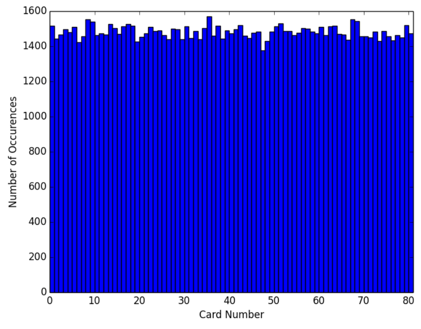


**Figure 3: 3 properties, 4 values, 10,000 trials Initial Board Distribution**

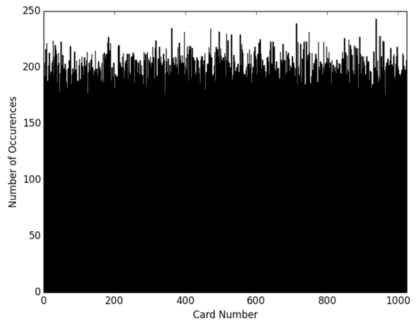


**Figure 4: 4 properties, 5 values, 10,000 trials Initial Board Distribution**

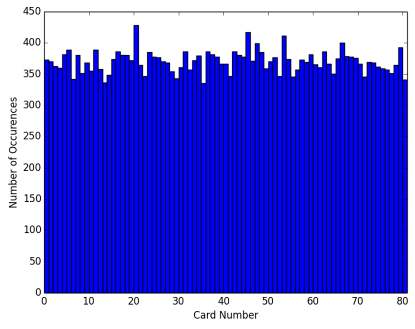


**Figure 5: 3 properties, 4 values, 10,000 trials Distribution of $4$ New Cards Drawn**

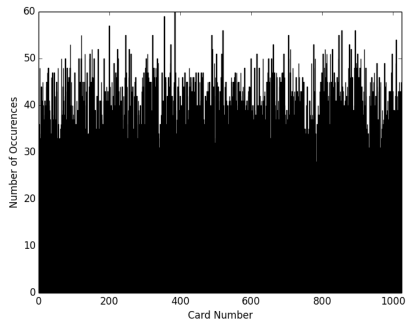


**Figure 6: 4 properties, 5 values, 10,000 trials Distribution of $5$ New Cards Drawn**

## 4.2. Brute Force Implementation

As a naive implementation, I created a brute force solution that tests all possible combinations of $v$ cards till a set is found and returns this set. A maximum number of $\binom{v*p}{v}$ possible sets will have to be examined before a set is found. This can be carried out $n$ times to find $n$ sets and therefore does not build on any previous knowledge, completely checking all possible sets each iteration, though it may have checked the possible set in a previous search. Though not optimized in any way, the brute force implementation provides a benchmark algorithm to compare the SMT based and dynamic algorithm against.

## 4.3. SMT Solver Implementation

**4.3.1. Z3** I used Z3, a high performance theorem prover created by Microsoft Research. The software allows me to easily create variables and constraints in Python. Each card that needs to be found represents $p$ variables, or $v*p$ total variables for all cards. I coded the above constraints in Z3 and if the SMT solver outputs "sat," then we know there exists $v$ cards that compose a set. If the SMT solver is "unsat," then there exists no sets and more cards must be drawn.

**4.3.2. Delayed Deletion Optimization** To optimize the SMT Solver implementation, I queue up the set to be removed and add constraint 17 (a satisfying set cannot include any card that was found to be part of a set) to the SMT Solver. If there exists many sets already on the board, this strategy will iteratively find them quickly. Once the SMT Solver no longer says a set exists, I then add $v$ new cards to hopefully create a new set. I will only do this once no set exists on the board after removing many cards to provide marginal speedup to the program as the solver will not have to rebuild many constraints every time a set is found.

**4.3.3. Symmetry Breaking Optimization** The SMT solver iteratively searches for a solution in a search tree, through a process called forward checking, attempting to constrain the domain for each variable until a satisfying assignment has been discovered [7]. If all possible assignments have been exhausted, then the constraints cannot be satisfied. In this game, the SMT solver will be forced to check every permutation of cards for a set, when in reality, all permutations of a given set are equivalent because order does not matter in a set of cards to satisfy the constraints. To break this symmetry for multiple sets, constraint 13 and 15 enforce that the cards in a set are in sorted order (ie. by the first property or if all equal then by the second property and so forth for all properties). Since the SMT solver will no longer search duplicate branches of its search tree, the SMT solver will find sets faster.

## 4.4. Dynamic Algorithm Implementation

The dynamic algorithm approach hinges on the tradeoff between memory and speed. The brute force solution retains no information and iterates over all combos as fast possible. The dynamic algorithm is on the other side of the spectrum and utilizes much more space, recording all possible sets that can be quickly completed, but does so in hopes for a benefit in overall speed.

The dynamic algorithm's main speedup is in the way that it creates a list of partial sets and a dictionary mapping the missing card to an almost complete partial set, in the hopes that it can quickly finish these sets when new cards are drawn. Outlined below is the general algorithm.

**4.4.1. Set Up** To create all partial sets, the dynamic approach begins similarly to the brute force solution. When the board is created, the solver will initially create partial sets for all $\binom{v*p}{2}$ combinations of two cards using Python's built in itertools library since a possible set can be determined by any two cards (ie. if the first two cards have the same value for a given property, the satisfying cards must have the same value for that property or if different than all the values need to be different). These two card partial sets will be the basis for creating larger partial sets and will ensure that all combinations of cards are checked to complete a set. I also include a symmetry breaking constraint, identical to the optimal SMT solver of retaining only sorted sets, as adding cards to a partial set can create many duplicate partial sets. Consider this example of its effect: Starting with partial sets [0,1] and [1,2], when I add 2 to the first set and 0 to the second, they will both yield set [0,1,2]. However, these are identical partial sets and would cause the search space to expand unnecessarily. I keep the partial sets sorted at all times. When I add a new partial set, I check whether it already exists to not create duplicates in the partial set list.

**4.4.2. Partial Set List and Quick Complete Dictionary Creation** For each partial set, the algorithm determines which cards can be added to the partial set from the board that satisfy the constraints that for each property, the values are either the same or all different and must not be in the set of cards that we flag as unusable cards as they have been used to complete a set in a previous iteration.

If a card can be added, I copy the given partial set and add the new card to it (if I added card 2 to a set [0, 1], I append [0,1,2] to the partial set list and keep [0,1]), creating a new partial set that is again sorted and verified that it is not in the partial set list already. Then, I append it at the end of the partial set list. This practice also ensures that new partial sets I append will not interfere with any previous sets. I must keep the original set because depending on how new cards are drawn, the original partial set can be satisfied differently (ie. I can add [0,1,3] while not interfering with [0,1,2] if 3 and 2 conflict with each other as [0,1] is left after matching to 2 so it can be matched with 3). By adding the cards incrementally, if it does happen that both cards satisfy the set, (ie. [0,1,2,3] is a set), then when I add the last card to the set it will see that it already exists in the partial set list by the symmetry breaking constraint and will delete one of the partial sets.

When I add cards to a partial set, its size can be in one of two cases:

**Case 1: Size *v*:** If the partial set is completely full and at size *v*, then the partial set is a complete satisfying set and can therefore be queued as a found set. These cards will also be queued up as cards that will need to be deleted from the partial sets and board and cannot be used to complete other partial sets.

**Case 2: Otherwise:** This partial set is missing more than one card. Therefore, we must append this new set to the end of the partial sets as this is a partial set that can be satisfied by adding new cards from the board. Python for loop construction is also very powerful and supports iterating over an expanding list. Therefore, it will reach all partial sets, even if added after the initialization of the for loop.

At the end of iterating through all partial sets, I will have had attempted to add all possible cards to all possible sets. Therefore, no cards could be added to any partial set without violating the satisfying constraint. We will have found all sets that could have been finished in the given board. Now, I iterate over all partial sets to see which sets are almost completed (size $v-1$). If they are missing one card, I create the quick complete data structure. The last card can be wholly determined by the cards currently in the set as the value will be the same if the set has the same value for a given property and if they are all different then it will be the last missing value. I add to the dictionary the missing card mapped to the partial set it completes.

**4.4.3. Quick Complete Upon Drawing Cards** When more sets are needed to be found than exist on the board, new cards must be drawn to create an opportunity for more sets. When these $v$ new cards are drawn, the dynamic algorithm will use its quick complete dictionary to quickly search to see if any of the $v$ cards drawn satisfy any of the partial sets that are missing one card. If they complete a partial set, the dictionary will delete the card from its table, append it to the partial set it completes, and add it to the queue of found cards. It is possible that cards used in a quick completed set could be used for another set. Before a set is completed, we need to ensure that any card was not used to complete a previous set.

After we have completed all of the partial sets we could have with the new drawn cards, we must create new partial sets of size 2 from these new cards. I create combinations over the new cards and all cards currently on the board (ie. if there are $v'$ new cards left after quick completing and $n'$ cards on the board, I create $v' * (n' + v' - 1)$ new partial sets to be added to the partial sets list). With these new partial sets appended, the program then reruns the algorithm described above in Partial Set List and Quick Complete Dictionary Creation to find new sets and create the appropriate data structures for future iterations.

**4.4.4. Deletion Bookkeeping** After both of the above steps, we need to ensure that any found set's cards are appropriately removed from the partial sets. Therefore, I iterate over all partial sets and see if any of them contain a card that was used to complete a set. If it does, I delete the entire partial set. This ensures that we will not be harboring any duplicate partial sets as we retain all partial sets as well. For example, if the card 3 needed to be deleted, we would delete the entire partial set [0,1,2,3]. We can delete the entire set as we retain the partial set [0,1,2] by the above step of the algorithm.

I also iterate over the quick complete dictionary and if any of the almost completed partial sets contain a card we are removing, I delete that appropriate entry from the dictionary as it is now missing more than one card which was the criteria to be included in the quick complete dictionary. Once we have deleted the cards from the appropriate data structures, we can iteratively continue these steps until *n* sets have been found.

**4.4.5. Finding a Set** When the dynamic algorithm does find a set, either through quick complete or while appending all satisfying cards to a partial set, it checks whether it has found *n* sets. If it has, it will immediately return and therefore will not have to waste time constructing all of the partial sets for subsequent iterations if it had found the threshold of *n* sets. This will speed up the final iteration as we will not build the complementary data structures unnecessarily.

## 4.5. Testing

For both of these implementations, I created testing infrastructure to ensure that the sets found by the solvers are indeed sets, and that the algorithm correctly removed the sets from the board.

**4.5.1. Testing Satisfying Set** To confirm that for all properties the cards from the set have either the same value or all different value, I iterated over all properties and checked whether when I put the values from each property into a set data structure that the length of the set was either 1, meaning that all the values were the same, or of length *v*, meaning all values were distinct elements.

To check that all the cards must be from the board, I had to store all iterations of the board, as new cards could be added and sets would be removed. Therefore, I used a set data structure to store all the cards, and updated this set as new cards were added to the board. Finally, I iterated over each set found and checked that it was included in the board at some point and therefore meant it was taken from the board.

To confirm that the last constraint of all distinct cards was satisfied for each possible set, I asserted that the length of the set was equivalent to the length of the set when converted into a set data structure. Therefore, if they are equal, all elements in the set are distinct.

Finally, the solver has to handle finding many sets, and to ensure that the solver does not find the same set $n$ times, I assert that when converted to a long list of cards that make up all $n$ sets found, that the length of this list converted to a set data structure should be the same as the length of the long list. Therefore, it would show that all $n*v$ cards are all distinct.

**4.5.2. Testing Correct Deletion** The constraints created for the SMT solver also rely on the fact that the cards from a satisfying set must be deleted from the board at some point. Therefore, as a check to ensure that the corresponding cards were deleted from the board, I iterated over all cards and asserted that all the cards found as part of sets were no longer part of the board at termination of the program. This is a good double check to ensure that the SMT solver is functioning properly, because the SMT hinges on the fact that cards that can no longer be used are deleted from the board before rebuilding the constraints. This is also important for the dynamic algorithm and brute force implementation because they must output the correct board as well after finding $n$ sets.

## 5. Results and Evaluation

### 5.1. Timing Tests

To test the relative effectiveness of the three implementations, I divided the testing into three cases to see how the implementations functioned on a variety of test cases. Since there are three variables to change: number of values, properties, and sets to find, I vary one of them and keep the other two constant. This creates a total of three overarching categories that can be broken down further as the two constant variables can take on different values.

To accurately test the timing of each solver, I utilized Python's time library to accurately test CPU time used for each of the processes. Therefore, if other processes are running in the background, they will not factor into the timing tests of these solvers. Also, only the process of creating the solver data structure and finding $n$ sets is accounted in CPU time, as creating the randomizer and verification of correct sets following finding the sets is irrelevant to the speed of the solver itself.

To give more accurate results, the times for a given set of values, properties, and sets to find are averaged over ten trials to give one data point. For each of the ten trials, I test each solver with the same beginning board so that we can see how they match up directly against each other on the same board.

I again used matplotlib to graph the time trials. These graphs are composed of connected dotted lines that are color coded by solver.

### 5.2. Brute Force Inefficiency

The brute force implementation is the solution that many solvers online employ. The brute force solution is fast in the actual Game of Set in which there are only 3 values and 4 properties, even being able to beat the SMT solver and dynamic algorithm. However, as $v$, $p$, and $n$ are increased to add complexity to the problem, the brute force solution begins to take exponential time. Therefore, the brute force solution must be deleted in real timing tests between the SMT solver and dynamic algorithm to provide fair competition and not ruin the scale of the graphs. To see the inefficiency of the brute force implementation, see Figure 7 as we vary the value and 8 as we change the number of properties.

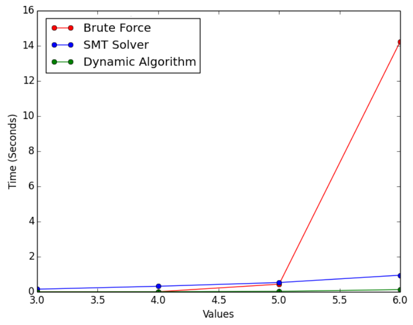


**Figure 7: 3 Properties and 5 Sets Found Varying the Number of Values**

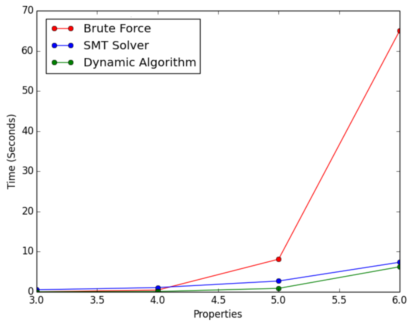


**Figure 8: 4 Values and 10 Sets Found Varying the Number of Properties**

### 5.3. Comparing Symmetry Breaking Constraints

The first symmetry breaking constraint used dictated that for the first property in which they differed, they needed to be sorted by that property. However, this was a relative constraint and did not condense the search tree as much as possible, labelled as constraint 13 above. Therefore, I also created an alternate constraint that is more specific and rigidly dictates that for the first property in which the differ the first card's value has to be 0 and so forth, labelled as constraint 15 above.

We can see the effect of symmetry breaking in Figure 9 and 10, in which I graph the speed of the three variations of the SMT solver against each other. As the number of values and properties of the game is increased, the no symmetry breaking SMT solver's run time blows up exponentially.

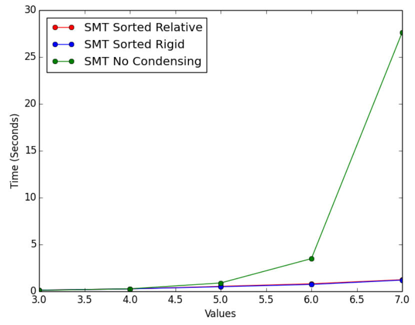


**Figure 9: 3 Properties and 5 Sets Found Varying the Number of Values**

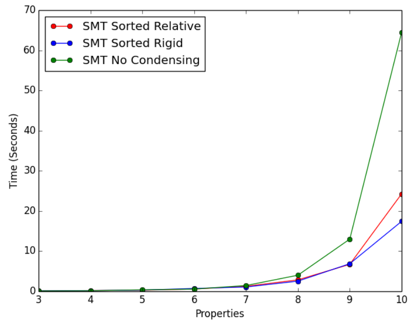


**Figure 10: 3 Values and 10 Sets Found Varying the Number of Properties**

To determine which symmetry breaking constraint yields the greatest speedup, I run the two implementations against each other. The rigidly sorted constraint condenses the search space more than the relative sorted constraint causing it to be faster as can be seen in Figure 11 and 12.

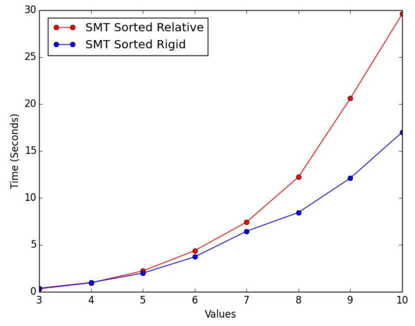


**Figure 11: 4 Properties and 10 Sets Found Varying the Number of Values**

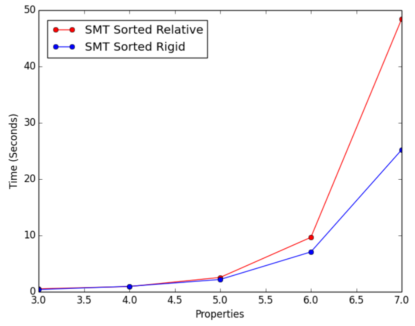


**Figure 12: 4 Values and 10 Sets Found Varying the Number of Properties**

In all cases, the rigid sort is marginally faster and this difference increases as the parameter increases. For the final implementation of the SMT solver, I settled on using the rigidly sorted constraint. This is the fastest version of the SMT solver and would provide the best competitor against the dynamic algorithm.

## 5.4. Comparing the Final Implementations of the Dynamic Algorithm and SMT Solver

## 5.5. Varying the Number of Values

There exists a threshold for which the dynamic algorithm performs better than the SMT solver. (See Figure 13). Up until there are 5 values, the dynamic algorithm is marginally faster than the SMT solver, but once the number of values increases past 5, the time grows exponentially while the SMT solver is relatively linear. At this threshold, the dynamic algorithm takes much too long and it becomes only feasible to run the SMT solver by itself.

The SMT solver is able to handle values as large as 10, even with 4 properties instead of 3 properties from the previous figure, while still retaining its linear growth. (See Figure 14). The dynamic algorithm performs much worse because of how many more partial sets it has to store. As we increase the number of values, the dynamic algorithm will store larger partial sets because satisfying sets will be larger (of size $v$ where $v$ is the number of values). Memory usage will be higher and iterating over these sets will take more time. This explains why the dynamic algorithm grows exponentially while the SMT solver is relatively linear with respect to the number of values.

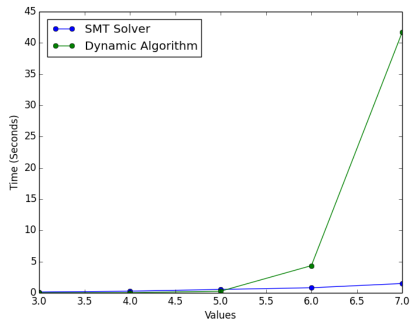


**Figure 13: 3 Properties and 5 Sets Found Varying the Number of Values**

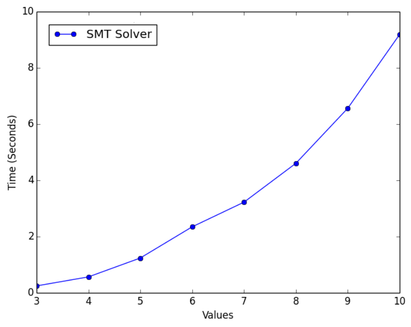


**Figure 14: 4 Properties and 5 Sets Found Varying the Number of Values**

## 5.6. Varying the Number of Properties

The SMT solver is able to finish larger instances of the problem faster than the dynamic algorithm past a certain threshold. (See Figure 15). Once the number of properties increases past 5, the SMT solver is significantly faster than the dynamic algorithm. At this threshold, the dynamic algorithm takes much too long and it becomes only feasible to run the SMT solver by itself.

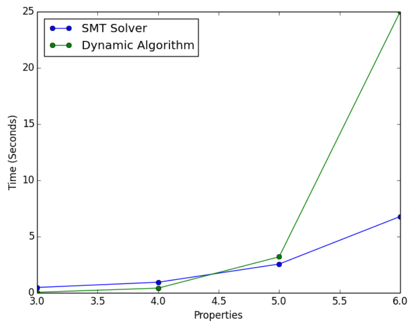


**Figure 15: 4 Values and 10 Sets Found Varying the Number of Properties**

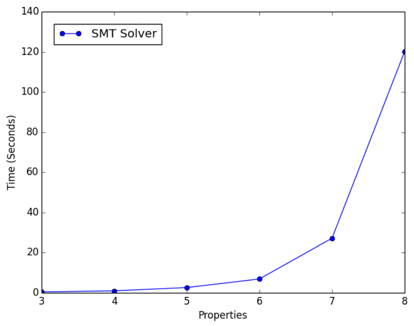


**Figure 16: 4 Values and 10 Sets Found Varying the Number of Properties**

The SMT solver is able to handle much larger number of properties than the dynamic algorithm without growing exponentially till a later degree. Once the number of properties is increased to 7, the SMT solver also begins to grow exponentially. (See Figure 16).

## 5.7. Varying the Number of Sets to be Found

As the number of sets to be found increases, if the number of values and properties is high enough, the dynamic algorithm is much slower than the SMT solver. (See Figure 17.) Even though the dynamic algorithm creates a speed up by retaining all partial sets especially for multiple draws, the speed up does not always outweigh the cost of storing so many partial sets.

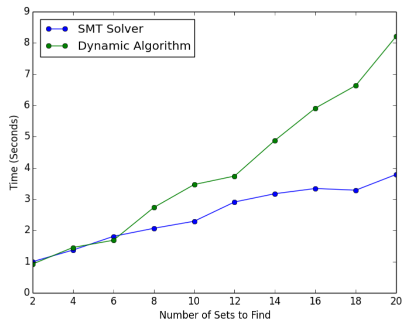


**Figure 17: 4 Values and 5 Properties Varying the Number of Sets to Find**

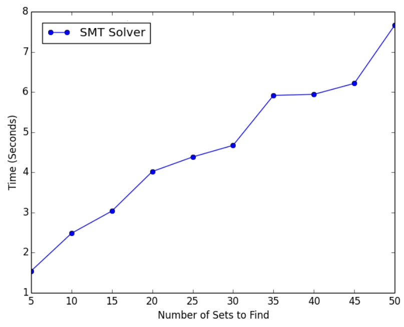


**Figure 18: 4 Values and 5 Properties Varying the Number of Sets to Find**

As the number of sets increases, the growth of both the SMT solver and dynamic algorithm are roughly identical. However, the SMT solver is marginally faster. The runtime continues to grow linearly in Figure 18 as it finds more sets than in the previous figure with the same constant parameters.

## 6. Conclusion and Future Work

This paper outlines the creation of three different solvers: brute force, dynamic algorithm, and SMT based, to solve the generalized Game of Set. Through timing tests, I found that as expected, the brute force implementation blows up exponentially as any of the parameters are increased. It does not lend itself to be an efficient solver for the more general case. The dynamic algorithm and SMT based solver both perform much better than the brute force and are superior in their own ways. The dynamic algorithm is faster on smaller cases than the SMT solver, but as the parameters increase, the memory usage of the dynamic algorithm become too unwieldy and causes exponential growth in its runtime. Interestingly, only when the SMT solver is used on higher number of properties does its growth become exponential. Otherwise, in the case of higher number of values and sets to find, the SMT solver can continue its linear growth.

For future research into this problem, it would be interesting and potentially much faster if there is a way to combine the SMT solver and dynamic algorithm. This way, we could benefit from the dynamic algorithm's speed on smaller cases but also the SMT solver's efficiency for larger cases. If a solution combining the two can be found and implemented, this line of thinking could be very useful in creating efficient solutions for other problems that grow exponentially and need an efficient solver for a wide range of parameters.

## 7. Acknowledgments

Thank you to Professor Kincaid for guidance and support through this project.

## 8. Ethics

I pledge my honor that this project represents my own work in accordance with University regulations.

Steven Takeshita

## References

[1] K. Chaudhuri *et al.*, “On the complexity of the game of set,” 2003.

[2] L. De Moura and N. Bjørner, “Satisfiability modulo theories: Introduction and applications,” *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011. Available: http://doi.acm.org/10.1145/1995376.1995394

[3] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research*, 6th ed. Edinburgh: Oliver and Boyd, 1963.

[4] F. S. M. Jorquera and A. Legge, “Set® card game solver using image processing techniques on smart-phone photos,” 2013.

[5] S. Nolte, “Javascript set game solver.”

[6] P. Norvig, “The odds of finding a set in the card game set®,” 2017.

[7] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2002.

[8] Samantha, “Game, set, match,” 2016. Available: https://socialmathematics.net/2016/06/01/game-set-match/

## 9. Appendix

All code is hosted on `https://github.com/steventakeshita/IW_Spring_2017`.