– 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, brute force, SMT solver, and dynamic algorithm, to efficiently find sets in the generalized version of the Game of Set. The brute force solution is ruled out as its growth is exponential. 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 program follows a relatively linear progression, whereas the dynamic algorithm exponentially grows, most likely due to its heavy memory usage, meaning at very high 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 of the game. 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 collected 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 fast can a computer find sets, and what are the different ways that we minimize the time. The game itself is relatively limited


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

with only three values and four properties, and therefore generalizing the number of values and properties will make the game much more difficult for a computer to solve. Therefore, the goal of my project is to create a solver that locates a set efficiently in practice for a game with $p$ properties and $v$ values. 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.

### 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 $v$ more cards will be immediately added. Therefore, there are 2 update functions that the algorithms must handle. One to add $v$ new cards to the board and the other to remove $v$ cards that formed a set.

By removing the first set of cards seen, this will simulate real gameplay, in which the goal is to find sets as fast as possible. By finding a generalized $n$ number of sets, as cards are added and removed, the dynamic algorithm will hopefully see some speed up. This further simulates gameplay in which the goal is to find a collection of sets and it is likely that there might not exists sets or sets

will be found causing new cards to be added.

## 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. Chaudhuri et al (2003) proved that the generalized version of Set is in fact NP-Complete through a reduction from perfect-Dimensional Matching, a known NP-Hard problem [**?**]. This paper proves computational complexity bounds, but does not solve the problem in practice. On the internet, solvers do exist and can be readily found. For example, this online JavaScript solver created by Steve Nolte, takes in the exact cards that appear on the board and outputs the total number of sets that exist on the board and the configurations [**?**]. 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).

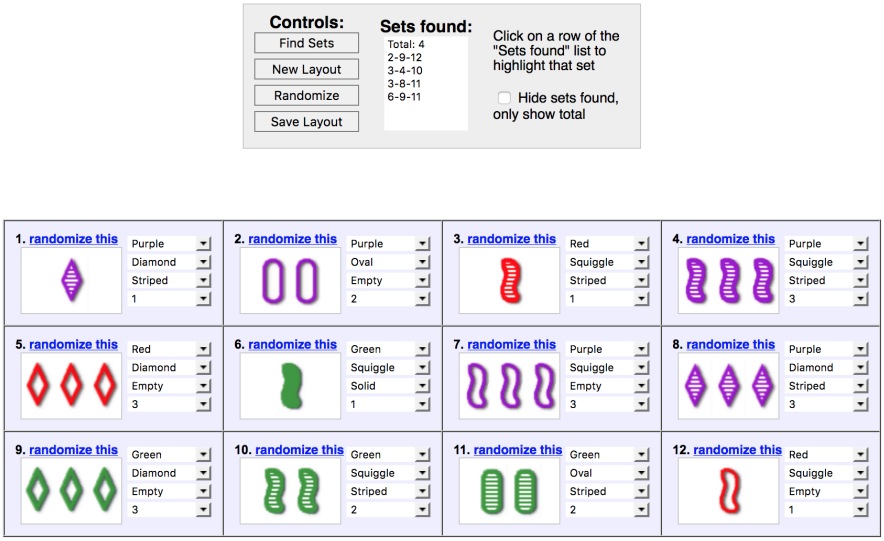


**Figure 2: Javascript SET Game Solver by Steve Nolte**

Some solvers even employ image processing to identify sets [**?**]. These more advanced programs are able to parse the cards from an image using computer vision and determine sets without a user inputting the cards manually. Again, this solver only supports use for one iteration of the game and will need to be updated manually to load in the new cards when the sets have been removed.

All these solvers are constrained to have four properties and three values as they only apply to the original Game of Set and cannot be used for a generalized version. Moreover, all these solvers restrict their mechanism to the brute force solution. They iteratively check all possible combinations of cards to determine satisfying sets. The solvers I created solve the generic case of Set in which it has $p$ properties, $v$ values, and we are searching for $n$ 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 decides 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 [**?**].

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 theoretically an efficient solver for an NP-Complete problem and can find sets due to the optimizations that the creators have implemented to create the fastest speedup possible.

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 $b_{i,j} \in \{0, 1, ..., v-1\}$:

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

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

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

**3.2.2. All Different or All Same Constraint** To correctly identify satisfying, distinct sets that exist from the given cards, we need to create three sets of constraints. The first set of constraints will only be satisfied when the cards found $K$ 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:**

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

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

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

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

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

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 $B$:

$$((k_{i,1} = b_{1,1}) \wedge (k_{i,2} = b_{1,2}) \wedge ... \wedge (k_{i,p} = b_{1,p})) \vee$$
$$((k_{i,1} = b_{2,1}) \wedge (k_{i,2} = b_{2,2}) \wedge ... \wedge (k_{i,p} = b_{2,p})) \vee ... \vee$$
$$((k_{i,1} = b_{n,1}) \wedge (k_{i,2} = b_{n,2}) \wedge ... \wedge (k_{i,p} = b_{n,p})) \quad (8)$$

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

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

**3.2.4. All Distinct Cards** We constrain the possible cards in the set to be from $B$, but this includes duplicates. A possible set could be three of the exact same cards and would satisfy the above constraints but does not represent a real set in the game. Therefore, the last constraint is that 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 coming after $i$ :

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

Written more succinctly:

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

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

$$\begin{aligned}(k_{1,1} \leq k_{2,1} \leq ... \leq k_{v,1}) \wedge ((k_{1,2} \leq k_{2,2} \leq ... \leq k_{v,2}) \vee \neg(k_{1,1} = k_{2,1} = ... = k_{v,1})) \\ \wedge ((k_{1,3} \leq k_{2,3} \leq ... \leq k_{v,3}) \vee \neg(k_{1,2} = k_{2,2} = ... = k_{v,2}) \vee \neg(k_{1,1} = k_{2,1} = ... = k_{v,1})) \\ \wedge ... \wedge ((k_{1,p} \leq k_{2,p} \leq ... \leq k_{v,p}) \vee \neg(k_{1,p-1} = k_{2,p-1} = ... = k_{v,p-1}) \vee ... \vee \neg(k_{1,1} = k_{2,1} = ... = k_{v,1}))\end{aligned} \tag{12}$$

Written more compactly:

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

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 determined by cards [0,1,2], the solver could potentially first check [1,2,0], which would waste time as we know that in the first position, must be lowest value for the first property in which the cards differ. Therefore, we can rigidly set that we know 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 and can be encoded as follows:

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

Written more compactly:

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

These two strategies are different and I will explore the relative effectiveness of both 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. To remove the set from the board, 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. This can be written as:

$$\forall k\in K,\forall v\in V\,(k_1\neq v_1\vee k_2\neq v_2\vee...\vee k_p\neq v_p) \tag{16}$$

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

$$((k_{i,1} \neq v_{1,1}) \vee (k_{i,2} \neq v_{1,2}) \vee \ldots \vee (k_{i,p} \neq v_{1,p})) \wedge$$
$$((k_{i,1} \neq v_{2,1}) \vee (k_{i,2} \neq v_{2,2}) \vee \ldots \vee (k_{i,p} \neq v_{2,p})) \wedge \ldots \wedge$$
$$((k_{i,1} \neq v_{v,1}) \vee (k_{i,2} \neq v_{v,2}) \vee \ldots \vee (k_{i,p} \neq v_{v,p})) \quad (17)$$

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

$$\bigwedge_{i=1}^{v} \left( \bigwedge_{j=1}^{v} \left( \bigvee_{m=1}^{p} k_{i,m} \neq v_{j,m} \right) \right) \quad (18)$$

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) from above. Therefore, this is represents a full reduction to SMT from the Game of Set.

However, in this generalized game for some values of $p$ and $v$, it 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, not using information from previous rounds to verify it faster.

### 3.3. Dynamic Algorithm

I will also implement a dynamic algorithm approach that will use previous knowledge of the board between draws to quickly discover sets. This approach will be useful because of the nature of the games changing probabilities as it is played. Norvig (2017) showed 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 [**?**]. Therefore, as the game is iteratively played in which sets are taken out, the game becomes immensely harder. Finding a high number of sets $n$ becomes harder as we must draw more and more cards. Therefore, remembering

partial sets across draws can greatly speed up the algorithm as it determines whether it can find a set. I will also employ a similar symmetry breaking constraint from the SMT solver for the dynamic algorithm to create a faster algorithm, as instead of storing all possible permutations of a partial set, I will only store the one set that a permutation will condense to by keeping each partial set sorted. 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 sets are removed from the board.

## 4. Implementation

The simulations for the randomizer and solvers were all written in Python. Considering Python's lightweight feel and ease of iterating over all variables in a list, makes it natural to easily represent a deck and board of cards. Therefore, Python made the code easier to write.

### 4.1. Randomizer

The randomization algorithm for creating a randomized deck is very important in terms of creating a real life scenario gameplay. Therefore, the algorithm must uniformly distribute all of the cards ($v^p$) initially so that no solver can have an unfair advantage knowing what cards are more likely to come out. I used the Random library in Python to generate cards. I assumed the Random library to be perfectly random. 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 [**?**]. To convert the algorithm for use on a deck, I use rejection sampling to determine new cards to add to the board. I create a randomized 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 is not in the board already and not in the removed cards set (ie. part of a set that was already found), I add it. If it is a duplicate of a card I have already seen, I generate another card until I have created $v * p$ 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, this is known as a Las Vegas algorithm. To support the update function of removing cards, I append those cards to a set of removed cards that cannot be used for new cards.

Therefore, when $v$ new cards are to be added, I generate $v$ new random cards through the above algorithm to be added to the board. This yields a randomized $v * p$ starting boards and unbiased drawing of $v$ cards from the deck.

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

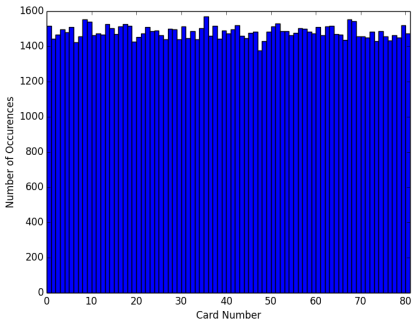


**Figure 3: 3 properties, 4 values, 10,000 trials Initial Board Distribution**

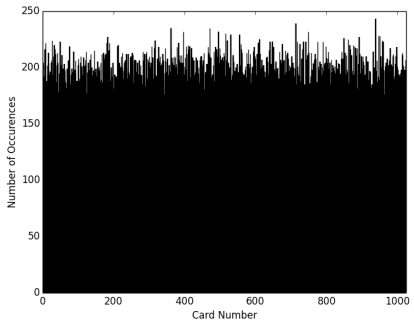


**Figure 4: 4 properties, 5 values, 10,000 trials Initial Board Distribution**

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 Figure 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.

### 4.2. Brute Force Implementation

As a very 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 total number of $\binom{v*p}{3}$ 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 anyway, the brute

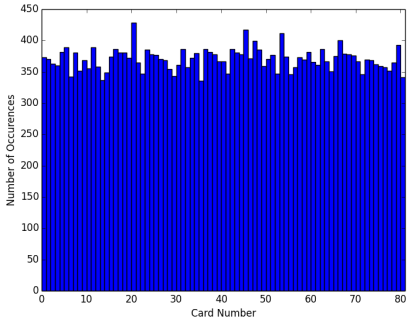


**Figure 5: 3 properties, 4 values, 10,000 trials Distribution of 4 New Cards Drawn**

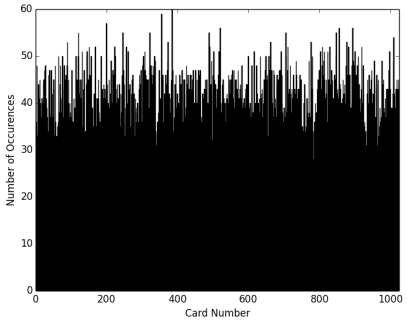


**Figure 6: 4 properties, 5 values, 10,000 trials Distribution of 5 New Cards Drawn**

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 creates an easy way to 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 a satisfying set of assignments for the variables that is satisfied under the constraints. Therefore, this satisfying assignments represents the $v$ cards that compose a set. If the SMT solver is "unsat," then there exists no satisfying assignment and more cards must be drawn to find a set.

**4.3.2. Delayed Deletion Optimization:** To optimize the SMT Solver implementation, I queue up the set to be removed and add constraint 14 (a satisfying set cannot include any card that was found the be part of a set) to run through the SMT Solver again to find another set to be removed in case there existed many sets already on the starting board. Once the SMT Solver no longer says a set exists, I then add $v$ new cards (or more if the board had a lot of sets and therefore the SMT solver was able to find many sets without having to update the constraints) 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.Once now sets exist on the board and I draw the corresponding number of cards, I rebuild the constraints and run the SMT solver to hopefully find another set of cards.

**4.3.3. Constraining the Domain Optimization:** The SMT solver iteratively searches for a solution through a search tree, attempting to constrain the domain for each variable until a satisfying assignment has been discovered, a process called Forward Checking, or all the possible assignments have been exhausted and 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 enforces 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) and constrains the domain of each variable. This will break the symmetry in the encoding and speed up the process of the SMT solver, as the SMT solver will not need to search branches of its search tree that are duplicates of another branch.

### 4.4. Dynamic Algorithm Implementation

The dynamic algorithm approach hinges on the tradeoff between memory and speed. The brute force solution does not need to keep track of partial sets and can spend time iterating over all combos as fast possible. The dynamic algorithm, is on the other side of the spectrum and utilizes much more space, recording down 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 so 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 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 properties 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 for completing a set. I also include a symmetry breaking constraint, identical to the optimal SMT solver, as adding in cards to a partial set will create many duplicate partial sets. As an example of how this condenses the space of partial lists consider this example: 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. Therefore, I keep the partial sets sorted at all times (by saying that the first property for which they differ, they must be in sorted order), and when I add a new partial set (add a new card to the partial set) to the partial set list I check whether it already exists thereby not creating duplicates in this list of partial sets.

**4.4.2. Partial Set List and Quick Complete Dictionary Creation** Next, 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.

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] just in case a card I draw later will determine a whole new set), creating a whole 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 that were added before (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). I must keep the original set because depending on how new cards are drawn, this stub set can be satisfied differently. By adding the cards incrementally, one card at a time to each partial set, if it does so 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 and then delete one of the partial sets as I keep them sorted.

When I add cards to a partial set that satisfy its constraints, 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 there 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. Therefore, to create the quick complete data structure, I add to the dictionary the missing card mapped to the partial set.

**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 the dictionary, append it to the partial set it completes, and add it to the queue of found cards. As we iterate through, it is possible that the cards used in one of these quick completed sets could be used for another completed set and therefore 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 stub partial sets from these new cards. I create combinations over all these cards and all the 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 stub 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 runs.

**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 it the 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. Therefore, we can remove all the partial sets that contain cards that were used to create a completed set.

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.

### 4.5. Testing

For both of these implementation, 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 check 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 the same length as before, meaning all values were distinct elements.

To check that all the boards 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 therefore distinct and make up all different cards.

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 relies 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 set, 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 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. At termination, we iterate over all cards found as being part of sets and assert that they are no longer part of the board.

# 5. Results and Evaluation

## 5.1. Timing Tests

To test the relative effectiveness of the three implementations, I divided the testing into three different 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 into multiple categories as the two constant variables can take on different values and have the third value vary.

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 and take real time, this 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. Therefore, this would account for instances in which a set can be found instantly or if all combinations need to be examined to find a set, and would give a more accurate time for a test case. Also, for each of the ten trials, I test each solver with the same beginning board such 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 efficient enough and fast in the case of 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 in some cases. However, as $v$, $p$, and $n$ are increased to add complexity to the problem, the brute force solution time 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 where red is the brute force solution, blue is the SMT solver, and green is the dynamic algorithm.

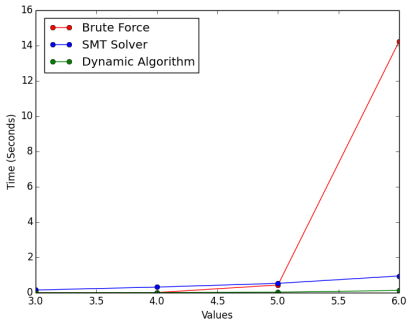


**Figure 7: 3 Properties and 5 Sets Found Varying the Number of Values**

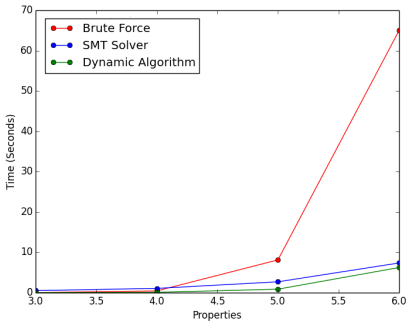


**Figure 8: 4 Properties and 10 Sets Found Varying the Number of Properties**

### 5.3. Comparing Symmetry Breaking Constraints

The first symmetry breaking constraint I used dictated that for the first property in which they differed, they needed to be sorted by the property. However, this was a relative constraint and did not condense the search tree as much as possible. 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.

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 (no condensing of the search tree is green, the relative sort is red, and the rigid sort is blue) against each other. As the number of values of the game is increased, the no condensing SMT solver's run time blows up exponentially. Therefore, we can see that the symmetry breaking constraints do provide considerable speed up for all different values.

Now, to determine which symmetry breaking constraint yields the greatest speedup, I run the

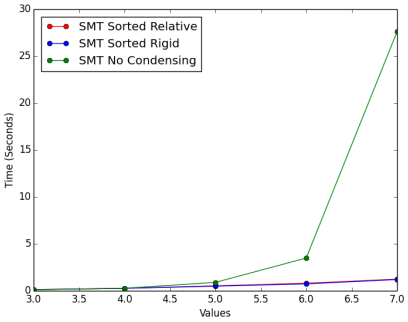


**Figure 9: 3 Properties and 5 Sets Found Varying the Number of Values**

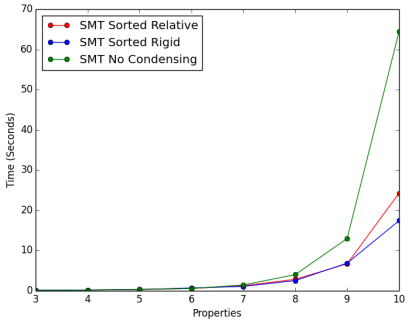


**Figure 10: 3 Values and 10 Sets Found Varying the Number of Properties**

two implementations against each other. As the rigidly sorted constraint condenses the search space more than the relative sorted, the speed up of adding this constraint can be seen in Figure 11 and 12 where the relative sort is in red and the rigid sort is in blue. In all cases, the rigid sort is marginally faster and this difference increases as the parameter increases. Therefore, for the final implementation of the SMT solver, I settled on using the rigidly sorted constraint. This is the fastest one and would provide the best competitor against the dynamic algorithm.

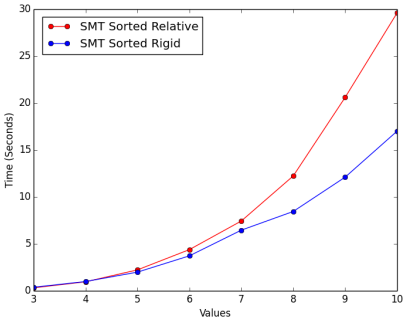


**Figure 11: 4 Properties and 10 Sets Found Varying the Number of Values**

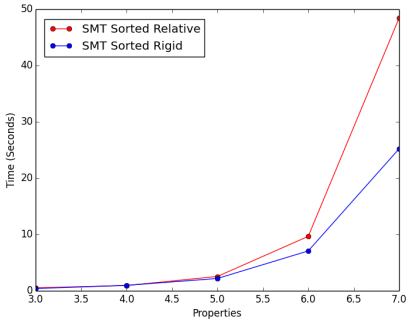


**Figure 12: 4 Values and 10 Sets Found Varying the Number of Properties**

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

To test the dynamic algorithm and SMT solver solutions against each other, I used a variety of different parameters to see how they perform against each other.

### 5.5. Varying the Number of Values

In the case of changing the value of the deck to be used, the value has to be restricted to be greater than two. If the number of values was either one or two, then a set could be immediately found as any card and any two cards are a valid set, in each case respectively. Therefore, I only test values in which the are larger than two for accurate results. As you can see, there exists a threshold for which the SMT solver performs better than the dynamic algorithm. See Figure 13 (dynamic algorithm is in green and the SMT solver is in blue). Up until there are 5 values, the dynamic algorithm is able to beat the SMT solver, but once the number of values increases past 5, the time grows exponentially while the SMT solver is relatively linear.

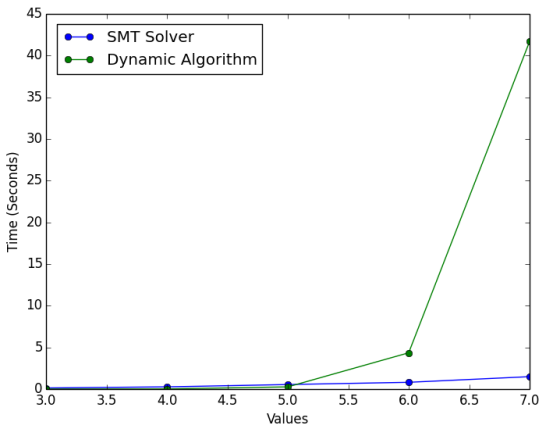


**Figure 13: 3 Properties and 5 Sets Found Varying the Number of Values**

### 5.6. Varying the Number of Properties

Again, the SMT solver is able to finish larger instances of the problem faster than the dynamic algorithm past a certain threshold. See Figure 14 (dynamic algorithm is in green and the SMT solver is in blue). Only once the number of properties increases past 5 is the SMT solver significantly faster than the dynamic algorithm.

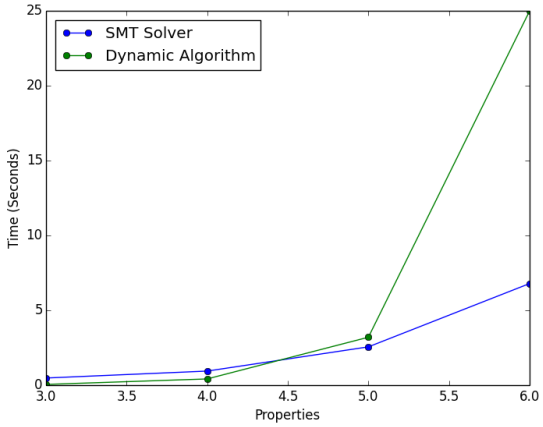


**Figure 14: 4 Values and 10 Sets Found Varying the Number of Properties**

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

Though I had assumed that the dynamic algorithm would expect the highest speed up when the number of sets to be found increases, as you can see from Figure 15 (dynamic algorithm is in green and the SMT solver is in blue), once the number of values and properties increases high enough, the dynamic algorithm cannot compete with the SMT solver and is much slower as the number of sets to be found. Even though the dynamic algorithm creates a speed up by retaining all partial sets especially for multiple draws, the speed up probably does not outweigh the cost of storing so many partial sets.

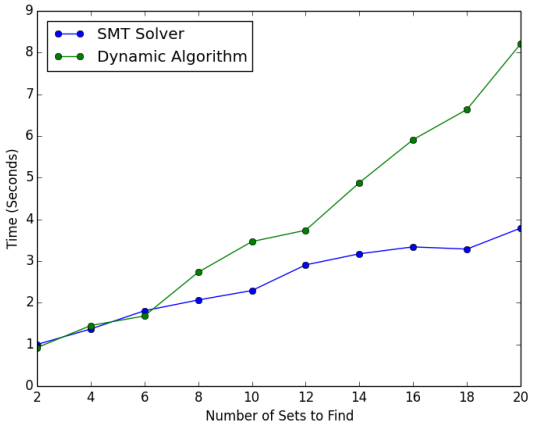


**Figure 15: 4 Values and 3 Properties Varying the Number of Sets to Find**

## 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. Though the brute force solver should be used for the original game which has very few possibilities and can be easily calculated, 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 superior on smaller cases than the SMT solver, but as the parameters increase, the memory usage of the dynamic algorithm become too unwieldy and causes it be much slower, exponentially increasing. On the other hand, the SMT solver, though slower on smaller cases, follows a roughly linear growth and is an efficient solver as the parameters increase to larger values.

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's efficiency for larger cases. One easy way to do this would be to run both algorithms in parallel and whichever one returns first will be a set. The only issue would be the bookkeeping would be much harder as we would constantly need to change the constraints for the SMT or the partial sets in the dynamic algorithm. 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

## 9. Appendix

These are the data tables for which the above graphs were created from.

### 9.1. Brute Force Inefficiency

| Value | SMT Time (sec) | Dynamic Time (sec) | Brute Force Time (sec) |
|---|---|---|---|
| 3 | 0.151558 | 0.0011556 | 0.0005788 |
| 4 | 0.3181856 | 0.0076392 | 0.0088644 |
| 5 | 0.5297282 | 0.0303382 | 0.4365208 |
| 6 | 0.9475802 | 0.1281706 | 14.238836 |

**Table 1: 3 Properties and 5 Sets Found Varying Values Showing Inefficiency of Brute Force**

| Properties | SMT Time (sec) | Dynamic Time (sec) | Brute Force Time (sec) |
|---|---|---|---|
| 3 | 0.515003 | 0.0131808 | 0.0244682 |
| 4 | 1.0666256 | 0.0920564 | 0.4419958 |
| 5 | 2.699265 | 0.8713774 | 8.1313608 |
| 6 | 7.3704448 | 6.2403746 | 65.0100778 |

**Table 2: 4 Values 10 Sets Found Varying Properties Showing Inefficiency of Brute Force**

### 9.2. No Condensing for SMT versus Condensing

| Value | SMT Rigid Time (sec) | SMT No Condense Time (sec) | SMT Relative Time (sec) |
|---|---|---|---|
| 3 | 0.1315286 | 0.1119434 | 0.1313658 |
| 4 | 0.262593 | 0.2711514 | 0.2629172 |
| 5 | 0.4915598 | 0.893835 | 0.5346276 |
| 6 | 0.7359456 | 3.492051 | 0.8089124 |
| 7 | 1.1994746 | 27.6221938 | 1.243199 |

**Table 3: 3 Properties and 5 Sets Found Varying Values Showing Inefficiency of Brute Force**

| Properties | SMT Rigid Time (sec) | SMT No Condense Time (sec) | SMT Relative Time (sec) |
|---|---|---|---|
| 3 | 0.1227916 | 0.11586 | 0.114809 |
| 4 | 0.1911524 | 0.1624124 | 0.2034166 |
| 5 | 0.3240426 | 0.3860674 | 0.297322 |
| 6 | 0.716594 | 0.5641488 | 0.5785234 |
| 7 | 1.0865544 | 1.4534238 | 1.252956 |
| 8 | 2.542946 | 4.0203912 | 2.8551718 |
| 9 | 6.8818012 | 13.0063166 | 6.7265682 |
| 10 | 17.436174 | 64.4995744 | 24.2810026 |

**Table 4: 3 values 5 sets to find varying properties**

### 9.3. SMT Condensing Runoff

| Value | SMT Rigid Time (sec) | SMT Relative Time (sec) |
|---|---|---|
| 3 | 0.3780628 | 0.3163608 |
| 4 | 0.9876946 | 0.9584652 |
| 5 | 1.9884428 | 2.2310528 |
| 6 | 3.7249208 | 4.3889242 |
| 7 | 6.460034 | 7.4369284 |
| 8 | 8.4522018 | 12.236469 |
| 9 | 12.1094048 | 20.610751 |
| 10 | 17.0089256 | 29.6069972 |

**Table 5: 4 properties 10 sets to find varying values**

| Properties | SMT Rigid Time (sec) | SMT Relative Time (sec) |
|---|---|---|
| 3 | 0.4043404 | 0.5289736 |
| 4 | 0.967702 | 0.9353798 |
| 5 | 2.1844898 | 2.548027 |
| 6 | 7.0779262 | 9.6672632 |
| 7 | 25.211324 | 48.43503 |

**Table 6: 4 values 10 sets to find varying properties**

### 9.4. SMT vs Dynamic Algorithm Final Runoff

| Value | SMT Solver (sec) | Dynamic Algorithm (sec) |
|---|---|---|
| 3 | 0.1379438 | 0.0020102 |
| 4 | 0.2696062 | 0.024966 |
| 5 | 0.5482628 | 0.265697 |
| 6 | 0.815219 | 4.354691 |
| 7 | 1.4929692 | 41.6958392 |

**Table 7: 3 properties 5 sets to find varying values**

| Properties | SMT Solver (sec) | Dynamic Algorithm (sec) |
|---|---|---|
| 3 | 0.4797718 | 0.0525568 |
| 4 | 0.9429812 | 0.4190292 |
| 5 | 2.5603214 | 3.2035058 |
| 6 | 6.7876418 | 24.9906024 |

**Table 8: 4 values 10 sets to find varying properties**

| Sets to Find | SMT Solver (sec) | Dynamic Algorithm (sec) |
|---|---|---|
| 2 | 0.998714 | 0.9229568 |
| 4 | 1.3699552 | 1.4530574 |
| 6 | 1.8076996 | 1.6831626 |
| 8 | 2.0658776 | 2.7318328 |
| 10 | 2.2921722 | 3.4669206 |
| 12 | 2.9084806 | 3.7383826 |
| 14 | 3.1736078 | 4.8806272 |
| 16 | 3.3400666 | 5.9122122 |
| 18 | 3.2863072 | 6.6352644 |
| 20 | 3.7891036 | 8.2078884 |

**Table 9: 4 values 5 properties to find varying number of sets**