Adv Alg Lab 1 Report

Cassandra Overney, Jane Sieving

March 2020

Part 1: Game Description

We chose to explore the card game, SET, which was invented by population geneticist Marsha Jean Falco in 1974. A deck consists of 81 unique cards that vary in 4 properties with 3 possible values for each property. The features are typically: number of shapes (1, 2, or 3), shape (diamond, wave, or oval), shading (solid, striped, or outline), and color (red, green, or purple). Each possible combination of features appears once in the deck, resulting in $3 \times 3 \times 3 \times 3 = 3^4 = 81$ cards.

In the game, players compete to quickly find valid SETs within some subset of the 81 cards. A valid SET is defined as 3 cards (the same as the number of values) where in each property, every card is either the same as, or distinct from, all other cards. Examples of valid SETs and an invalid SET are given below:

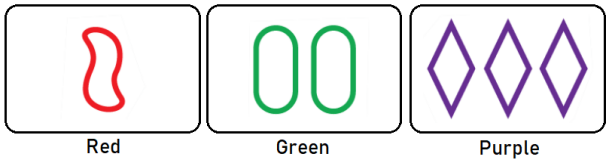

Figure 1: *A valid SET. Shape, number and color are all distinct, fill is consistent.*

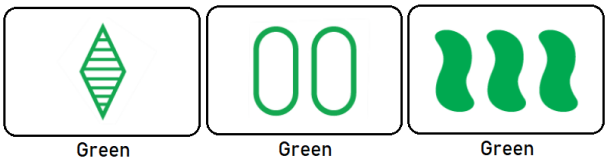

Figure 2: *A valid SET. Shape, number and fill are all distinct, color is consistent.*

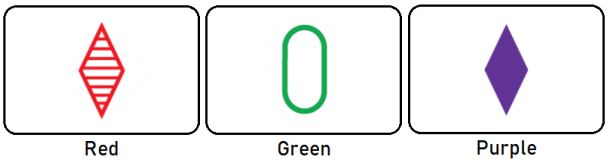

Figure 3: *An invalid SET. Fill and color are all distinct, number is consistent, but exactly 2 are diamonds.*

It is possible for all cards to be distinct in every property, but they must differ in at least one property as a deck contains no duplicates. In some cases, it is impossible to find a valid SET among the current subset of cards under consideration. Because of this, it is interesting to create an algorithm which can "solve" SET: that is, find a valid SET among a subset of cards, or determine if no valid SET exists in the subset.

From this understanding of the game, we can formally define it as follows:

- p: number of properties (shape, color, etc.). $p = 4$ in standard SET.
- v: number of values (i.e., 1, 2, 3). $v = 3$ in standard SET.
- D: the deck. $D = \{1...v\}^p$; that is, the deck contains all cards having one value for each property, for all permutations of different values. The deck has size v^p.
- C: the dealt cards, a subset of all possible cards D. $C \subseteq D = \{1...v\}^p$. The number of cards dealt (the size of the subset) will vary, for reasons explained shortly.
- c: a card in D. A given card c has values $c[0]...c[p]$, or a value for each property.
- S: a SET, consisting of v different cards in D: $S = \{c_1...c_v\} \in D$
- Valid SET: a SET of cards where for all properties i from 1 to p, the values $c_1[i]...c_v[i]$ are either all the same or all unique.
- The problem: Given a subset of cards $C \in D$, with some values of v and $p > 0$, does there exist a valid SET $S \subseteq C$?

Given the many parameters that can be changed in the game of SET (v, p, number of dealt cards), there are multiple ways to analyze the time complexity of the problem. k-card SET has a time complexity of k^v, because v cards must be selected and there are k cards in C. To solve the problem, all combinations of v cards can be checked for validity. This results in polynomial time with respect to k, meaning that k-card SET is in P.

On the other hand, k-value SET and k-property SET are more unwieldy. In k-value SET, k is a number with respect to which we calculate the complexity, and p may vary (in other words, it is an arbitrary constant.) In k-property SET, k represents the number of properties instead, while v is varied. There is an algorithm for k-value set which runs in $\Theta(n^{k-1}p)$ time. This is exponential, and while a better algorithm may exist, it shows the increasing complexity of the problem when viewed with respect to v.

We will focus on k-property SET, which can be shown to be NP-complete by reducing k-dimensional matching to k-property SET. In this k-dimensional matching problem, a matching M will be a subset of configurations $\{1...v\}^k$. M is a perfect matching if for all properties (or dimensions) i from 1 to k, the values of that property/dimension are unique across M.

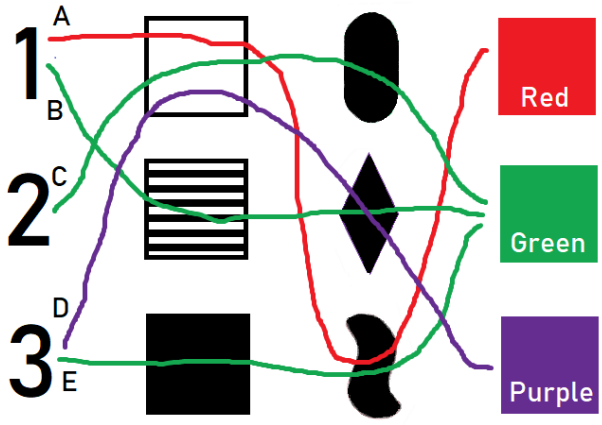

Figure 4: *The cards from figures 1 and 2 as a 4-dimensional matching problem. Cards A, C and D or cards B, C and E make valid SETs; however, we can see that 4-D matching does not fully solve the SET problem because nodes are re-used when a SET has a consistent property.*

Fig. 4 shows a representation of 4-property, 3-value SET as a 4-dimensional matching problem. The cards from figures 1 and 2 are represented as lines. Lines represent "configurations", or choices of 1 value in each dimension. A perfect matching M would be a selection of v configurations so that no 2 or 3 configurations shared a value on any dimension. It is important to note that to fully model k-property SET in this way, it would have to be valid for all v configurations to share a value in any (but not all) dimension(s).

Part 2: Proof of NP-completeness

In this section, we will prove that the k-property SET problem is NP complete by reducing a known NP complete problem, k-dimensional matching, into the k-property SET problem such that the k-property SET problem is solvable if and only if the k-dimensional matching problem is solvable. The decision problem for k-dimensional matching is "given a set $C \subseteq \{1, ..., v\}^k$ with $k > 0$, does there exist a perfect matching $M \subseteq C$?" M is a perfect matching if the number of elements in M is equal to the number of values per dimension, $|M| = v$, and for all dimensions $i \in \{1, ...k\}$, the values of these dimensions for all elements in M are distinct from each other. On a conceptual level, this is very similar to finding a SET with no shared attributes where dimensions correspond to properties. Thus, every perfect matching is a valid SET.

We begin the formal reduction by defining $C \subseteq \{1, ..., v\}^k$ and $C' \subseteq \{1, ..., v+1\}^k$ such that C' contains a valid SET if and only if C contains a perfect matching. C' can be viewed as a deck of SET cards with one extra possible value for each property (note that we still have k properties). To go from C to C', we can add a special card, c, which has the $v+1$ value for all k properties, $c = (v+1, ..., v+1)$. Since we are just adding one card to the C set in the k-dimensional matching problem, we can construct C' in polynomial time. With this construction, a perfect matching in C corresponds to a valid SET in C', and a valid SET in C' corresponds to a perfect matching in C.

First suppose that there is a perfect matching $M \subseteq C$. Let $M' = M \cup c$ where we add the special card, c to M. Since none of the elements in M share a value with c and $|M'| = v + 1$, M' is a perfect matching and is also a valid SET. $M' \subseteq C'$, so C' contains a valid SET.

Now suppose C' contains a valid SET, M', with cards $c_1, ..., c_{v+1}$. There must be some property in k for which all the cards are distinct or else the cards would be all the same, which isn't possible in a SET deck. Since we have $v + 1$ cards, one of the cards must have a value of $v + 1$ for one of its properties. $v + 1$ only exists in c, so $c \in M'$. Also, since no other elements in M' agree with c in any dimension, the elements of M' must be distinct in all properties/ dimensions, which means M' is a perfect matching. If we remove c from M', we still get a perfect matching, solving the k-dimensional matching problem.

Since we can reduce a k-dimensional problem into a k-property SET problem in polynomial time, the k-property SET problem is NP-complete.

It might be more difficult to reduce the k-property SET problem into the k-dimensional matching problem because the cards in a valid SET don't have to be all distinct from each other as long as any property shared between a pair of cards is shared by all the cards in a SET. Due to this, k-property SET might be harder to solve than k-dimensional matching.

Part 3: Implementation

For the implementation part of this lab, we decided to implement an optimum pair checking algorithm for the 3-value SET problem. Optimum pair checking is very similar to the brute force approach but instead of potentially iterating through all pairs of SET cards, the n cards are partitioned into two groups, and only pairs within the two separate groups are examined.

We created three classes, Deck, Hand, and Card. Each Card is represented by a list of values where the index of the list corresponds to a property. A Deck object holds all possible v^p combinations of cards and

chooses n random cards to form a Hand. A Hand is associated with a set of cards and is where we look for SETs. Our source code can be found here.

The pseudo-code for the brute force algorithm is:

1. Determine all pairs of cards in a Hand object
2. While a SET is not found, iterate through all pairs of cards. For each pair of cards c_i, c_j
   (a) Compute the third card c_k that forms a SET with c_i, c_j
   (b) Check if c_k is in the Hand, if so return the valid SET
3. Return no valid SET

The time complexity for this algorithm is $O(n^2p)$. Basically, there are $\binom{n}{2} = O(n^2)$ pairs of cards to iterate through. Computing c_k takes $O(p)$ time because we need to iterate through all properties of the card. If the number of values equals 4, then we would need to iterate through every triplet of cards. There are $\binom{n}{3} = O(n^3)$ triplets of cards to iterate through. Following this pattern, if the number of values equals k, then our run time would be $O(n^{k-1}p)$. This run time is exponential with respect to k, which supports the idea that if we don't define v or p, then we wouldn't be able to find a SET in polynomial time.

For the specific case where $v = 3$, an optimal pair-checking algorithm can be used to achieve a better run time than the brute-force method. The pseudo-code for the optimum pair checking algorithm is:

1. Split the cards in a Hand object into two lists C_1 and C_2
2. Determine all pairs of cards in C_1 and C_2 separately
3. While a SET is not found, iterate through all pairs of cards. For each pair of cards c_i, c_j:
   (a) Compute the third card c_k that forms a SET with c_i, c_j
   (b) Check if c_k is in the Hand, if so return the valid SET
4. Return no valid SET

The time complexity for this is still exponential, but it is better than brute force. If there are n cards in the Hand, then there are $n/2$ cards in each of C_1 and C_2. For each of these subsets, $(n/2)^2$ checks must be done to check for a c_k in the Hand for every pair in either subset. Each of the checks will take $O(p)$ time as before, since every property must be computed. So the overall run time will be $O(2 * (n/2)^2 * p)$ or simply $O(n^2p/2)$. Compared to the brute force algorithm on $v = 3$, the optimal pair checking can be expected to run twice as fast on a given Hand of cards that contains a valid SET. This algorithm is visualized in Fig. 5.

To get a better idea of how the brute force and optimum pair checking algorithms differ from each other in terms of run time, we decided to generate two figures. Fig. 6 compares the two algorithms when we gradually increase the number of cards associated with a Hand object. Fig. 7 compares the two algorithms when we gradually increase the number of properties in the SET deck.

From Fig. 6, we can see that the optimum pair checking algorithm actually takes longer than brute force as the number of cards dealt increases. The number of cards dealt refers to n in our $O(n^2p)$ run time. The two algorithms are harder to compare at lower x-values. Considering how drastically the optimum pair checking curve oscillates, it is hard to draw conclusions from the time difference between the two algorithms. The slowness of the optimum pair checking algorithm may be due to our implementation in code, which involves converting between set-like and list-like objects and concatenating lists. These steps could add on the order of $O(n^2/2)$ operations, which might slow the algorithm down beyond the pair checking process alone. This would be especially evident with increased n, as there would be more cards and pairs to deal with. Another possible factor is that the n values depicted in the graph may be too small to actually observe the long-term run time behaviors of the 2 algorithms.

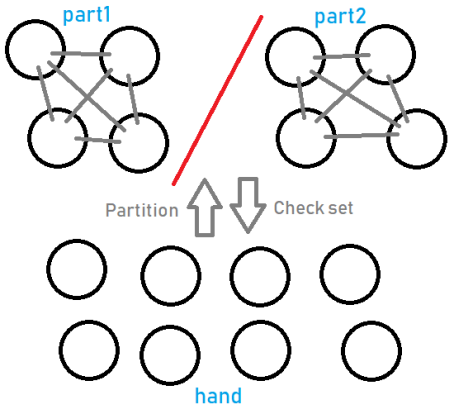

Figure 5: *How pair checking reduces the number of checks required when v=3*

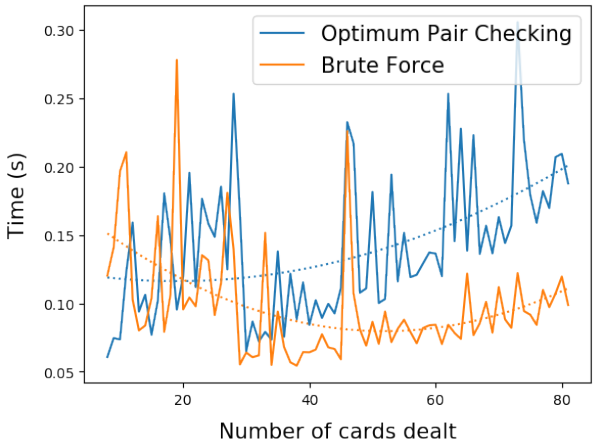

Figure 6: *Comparing Optimum Pair Checking vs. Brute Force for varying deck size.*

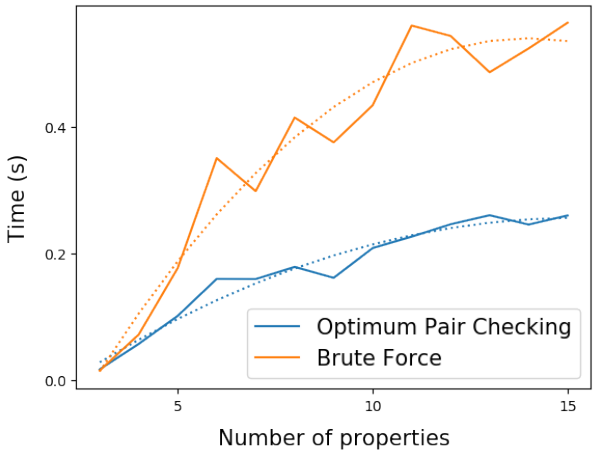

Figure 7: *Comparing Optimum Pair Checking vs. Brute Force for varying property number.*

From Fig. 7, a difference between the two algorithms is more apparent. After the number of properties surpasses 5, brute force tends to take around 0.2 more seconds to complete compared to the optimum pair checking algorithm. More generally, the brute force algorithm appears to take about twice as long as the optimum pair checking algorithm, which is consistent with our earlier run time analysis. The number of cards dealt in every trial was 12.

Test Cases

We wrote a few test cases using *pytest* to make sure that our SET solver finds valid SETs. We tested object representations of the valid and invalid SETs depicted in Figures 1, 2, and 3. Our test for Fig. 1 is the following:

```
card1 = Card([0, 1, 2, 0]) # 1 red outlined wave
card2 = Card([1, 2, 2, 1]) # 2 green outlined ovals
card3 = Card([2, 0, 2, 2]) # 3 purple outlined diamonds
hand = Hand([card1, card2, card3])
# test optimum pair checking algorithm
set1 = hand.find_set() # returns a tuple
assert is_valid_SET(set1) == True # should be a valid SET
# test brute force algorithm
set2 = hand.find_set_brute_force() # returns a tuple
assert is_valid_SET(set2) == True # should be a valid SET
```

The is_valid_SET() function is given a set of Card objects and determines whether they form a valid SET. The function basically iterates through all properties, and if there is any where the cards don't either all share a value or have different values then it returns False. If it is able to reach the end of the function, then it returns True. More details regarding our tests can be found in *test.py*.

References

[1] Chaudhuri, Kamalika, et al. "On the complexity of the game of set." (2003). http://pbg.cs.illinois.edu/papers/set.pdf.

We used this paper as inspiration for our NP-complete proof and optimum pair checking implementation. After carefully reading through this resource, we attempted to re-explain the concepts using our own words and knowledge on SET and NP-completeness.

[2] Lampis, M., Mitsou, V. "The Computational Complexity of the Game of Set and its Theoretical Applications" (2014). https://arxiv.org/pdf/1309.6504.pdf.

We referenced this paper when trying to find a way to prove that SET was NP-complete.