University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Conference and Workshop Papers | Computer Science and Engineering, Department of

2011

A Reformulation Strategy for Multi-Dimensional CSPs: The Case Study of the SET Game

Amanda Swearngin
University of Nebraska-Lincoln, aswearng@cse.unl.edu

Berthe Y. Choueiry
University of Nebraska-Lincoln, choueiry@cse.unl.edu

Eugene C. Freuder
University College Cork, e.freuder@cs.ucc.ie

Follow this and additional works at: https://digitalcommons.unl.edu/cseconfwork

 Part of the Computer Sciences Commons

Swearngin, Amanda; Choueiry, Berthe Y.; and Freuder, Eugene C., "A Reformulation Strategy for Multi-Dimensional CSPs: The Case Study of the SET Game" (2011). *CSE Conference and Workshop Papers*. 178.
https://digitalcommons.unl.edu/cseconfwork/178

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Conference and Workshop Papers by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

Swearngin, Choueiry & Freuder in Ninth International Symposium on Abstraction, Reformulation and Approximation (SARA) 2011.
Pages 107-116. Copyright 2011, AAAI Press. Used by permission.

A Reformulation Strategy for Multi-Dimensional CSPs: The Case Study of the SET Game

Amanda Swearngin[1] **Berthe Y. Choueiry**[2] **Eugene C. Freuder**[3]

[1]ESQuaReD Laboratory [2]Constraint Systems Laboratory
University of Nebraska-Lincoln, USA
{aswearng|choueiry}@cse.unl.edu

[3]Cork Constraint Computation Centre, University College Cork, Ireland
e.freuder@4c.ucc.ie

Abstract

In this paper we describe a reformulation strategy for solving multi-dimensional Constraint Satisfaction Problems (CSPs). This strategy operates by iteratively considering, in isolation, each one of the uni-dimensional constraints in the problem. It exploits the approximate symmetries identified on the domain values in order to enforce the selected constraint on the simplified problem. This paper uses the game of SET, a combinatorial card game, to motivate and illustrate our strategy. We propose a multi-dimensional constraint model for SET, and describe a basic constraint solver for finding all solutions of an instance of the game. Then, we introduce an algorithm that implements our reformulation strategy, and show that it yields a dramatic reduction of the search effort. Our approach sheds a new light on the dynamic reformulation of CSPs, leading the way to new strategies for effective problem solving. We use the game of SET as a toy problem to illustrate our strategy and explain its operation. We believe that our approach is applicable to more complex domains of scientific and industrial importance, and deserves thorough investigations in the future.

1 Introduction

Multi-dimensional Constraint Satisfaction Problems (CSPs) were introduced in (Yoshikawa and Wada 1992) and have been shown to model many applications of practical importance such as resource allocation and configuration. In a multi-dimensional CSP, the domains of all variables are identical and the domain values are specified according to a set of domain dimensions (i.e., attributes). The constraints in the problem that apply to a single domain dimension are said to be uni-dimensional, otherwise they are multi-dimensional. In this paper, we propose a general reformulation strategy for solving multi-dimensional CSPs that reduces the cost of problem solving by facilitating the discovery of approximate symmetries. Our strategy, shown in Figure 1, operates on a multi-dimensional CSP by iteratively enforcing each uni-dimensional constraint on the corresponding dimension of the domain while ignoring all other domain dimensions and constraints. Ignoring all but one constraint allows one to identify, in the relaxed problem, symmetries that do not hold in the original problem (Freuder and Sabin 1995; 1997). Such approximate symmetries can be exploited to reduce the computational cost of enforcing the constraint on the relaxed problem. Each step in Figure 1 may discover unsolvability or produce one or more simplified subproblems where the considered constraint holds. At the end of the process, any constraint solver can be used to solve the resulting problem(s) by enforcing the remaining constraints. In (1995; 1997), Freuder and Sabin propose a similar approach that uses symmetries based on neighborhood value interchangeability (Freuder 1991). However, their strategy differs from ours as follows. The first simplified CSP is solved and its solution is used to solve the original CSP. In our approach, we foresee a *sequence of reformulation steps*, each enforcing a single uni-dimensional constraint, where as Freuder and Sabin described a single abstraction step (ref. '2. *Reduce*' in (Freuder and Sabin 1997)).

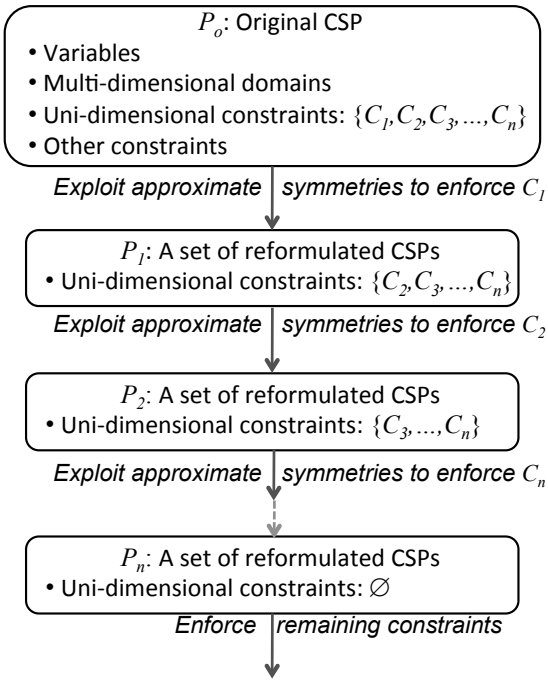


Figure 1: A reformulation strategy for multi-dimensional CSPs.

In this paper, we introduce a reformulation algorithm that 'instantiates' the general strategy of Figure 1 to solve the

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

game of SET,[1] a combinatorial card game. This game was invented in 1974 by Marsha Jean Falco,[2] a population geneticist. She was inspired to create the game by her work on determining whether epilepsy in German shepherd dogs is inherited (Davis and McLagan 2003). In this paper, we propose a multi-dimensional constraint model for SET that has four uni-dimensional constraints. We describe a basic constraint solver for solving an instance of the game. Our solver is a simple backtrack search procedure with symmetry breaking to find all the solutions of the instance. Our reformulation algorithm for SET features the following components: (1) Heuristics for selecting the uni-dimensional constraint to consider at each step; (2) The use of meta-interchangeability (Freuder 1991) as an approximate symmetry; and (3) A disjunctive decomposition of an intermediate CSP into subproblems with non-overlapping solutions as a result of enforcing the selected uni-dimensional constraint. Our reformulation significantly reduces the search effort.

We have implemented our approach in an online interactive system for a single player and for two players. Naturally, the value of the system is *not* in solving the game, which, given its size, can be played by humans without the help of a computer and is widely enjoyed by children and mathematicians alike. However, we believe that our system, when completed,[3]o will be useful to teach and demonstrate problem-solving strategies to students in Computer Science and to the general public. Beyond SET, our approach sheds a new light on the dynamic reformulation of CSPs, leading the way to new strategies for effective problem solving. We believe that our approach is applicable to more complex domains of practical industrial importance beyond the toy problem considered in this paper, which calls for future investigations.

This paper is structured as follows. Section 2 gives background information about the game, Constraint Satisfaction, and multi-dimensional CSPs. Section 3 describes our model and the search procedure for solving it. Section 4 introduces our reformulation of the constraint model, and Section 5 discusses our reformulation algorithm for SET. Section 6 discusses our results. Section 7 presents our interactive interface for the game, available online on `http://gameofset.unl.edu`. Finally, Section 8 relates our work to previous research, and Section 9 concludes this paper drawing directions for future research.

2 Background

Below, we introduce the game and provide some background information about the modeling techniques.

2.1 The Game of SET

SET is a combinatorial card game consisting of a deck of 81 playing cards. Each card is uniquely determined by the values of four *attributes*, namely, the *number* of objects drawn on the card and their *color*, *filling*, and *shape*. We denote these attributes by N, C, F, and S, respectively. Each attribute takes one of three possible values as follows: {1,2,3} for the dimension *number*, {*red*,*green*,*purple*} for *color*, {*striped*,*full*,*empty*} for *filling*, and {*squiggle*,*oval*,*diamond*} for *shape*, see as shown in Figure 2. To play the game, twelve cards are dealt and

Number	1	2	3
Color	red	green	purple
Filling			
Shape			

Figure 2: The four attributes and their values.

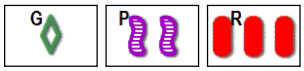


Figure 3: A solution set.

placed, face up, on the table, visible to all players. The players compete to find a collection of exactly three cards that constitute what we call a *solution set*. For each of the above listed attributes, the three cards of a solution set must all have either the same value or different values for the attribute. Figure 3 shows three cards forming a solution set. In this example, the three cards differ on all four attributes. The first player to identify a solution set picks it up, and the cards are replaced with new ones from the deck. If all players agree that a solution set cannot be found among the twelve cards, three more are dealt and the game resumes. The operation is repeated until a set is found. Usually, the number of cards on the table quickly returns to twelve. The maximum number of cards on the table at any one time is 21, because any combination of 21 cards is guaranteed to have at least a solution set. The proof was performed by exhaustive computation (Davis and McLagan 2003). The game continues until all cards have been picked up or there are no more sets among the remaining cards. The winner is the player with the largest number of sets at the end of the game. For the sake of space, our examples will show game instances with only nine cards although the original game considers twelve cards.

Obviously, a simple nested *for*-loop can 'easily' generate all combinations of three cards. Each combination can then be tested to check whether or not it satisfies the constraint. Such an approach is shortsighted. Indeed, human beings are unlikely to play the game by examining $\binom{12}{3} = 220$ combinations. Instead, it is fair to assume that they use various modeling and reformulation strategies. It would be equally ridiculous to use the simple nested *for*-loop to solve industrial-size problems because the number of combinations grows exponentially with the size of the problem and few of the generated combinations satisfy the constraints. For example, $\binom{12}{3} = 220$ combinations have on average 2.7 solutions, and $\binom{81}{3} = 85320$ combinations have only 1080 solutions. Thus, the simple nested *for*-loop is a strategy that is neither interesting to teach a human player nor to use for automation in an industrial setting.

2.2 Constraint Satisfaction Problems (CSPs)

A Constraint Satisfaction Problem (CSP) is defined by $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ a set of domains,

[1] `http://en.wikipedia.org/wiki/Set_(game)`

[2] `http://www.setgame.com/set/history.htm`

[3] A tool for displaying and explaining the reformulation steps has yet to be developed.

and $\mathcal{C}$ a set of constraints. Each variable $V_i \in \mathcal{V}$ has a finite domain $D_i \in \mathcal{D}$. Each constraint in $\mathcal{C}$ applies to a subset of the variables (the scope of the constraint), and restricts the combination of values that those variables can take at the same time. A solution to a CSP is an assignment of a value to each variable such that all the constraints are satisfied. In general, solving a CSP requires finding one or all solutions.

2.3 Multi-Dimensional (CSPs) & Related Work

In (1992), Yoshikawa and Wada defined of a *multi-dimensional* CSP as a CSP problem where:

1. All variables have the same domains,[4] and
2. The domain can be specified by a multi-dimensional array of values, where each value is described by a combination of domain dimensions.

Further, they called a constraint defined over only one domain dimension a *one-dimensional* constraint, otherwise the constraint is called a *multi-dimensional* constraint. As an example, they discussed a classroom-scheduling application where the classrooms are the variables and the domains of the variables are combinations of teachers and time slots.[5]

3 Solving SET as a CSP

Below, we describe our constraint model and a search procedure for finding all solutions.

3.1 A Constraint Model for SET

Our constraint model has (only!) three variables corresponding to the three cards of a solution set: $\mathcal{V} = \{V_1, V_2, V_3\}$. All three variables have the same domain, which are the cards c_i placed on the table:

$$D_1 = D_2 = D_3 = \{c_1, c_2, \ldots, c_i\}, \text{where } i \in [3, 21].$$

We model the domain of a SET variable as a multi-dimensional array indexed by the following attributes of the playing cards: *number*, *color*, *filling*, and *shape* (see Figure 3). We also include the unique identifier *id* of a playing card as a fifth pseudo-attribute: It allows our program to uniquely identify each card. For the sake of clarity, we flatten the multi-dimensional representation of a domain in the two-dimensional array shown in Figure 4. We call this array the *domain table*.

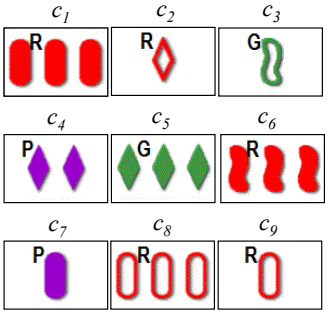
To represent the game of SET, we use a set of five ternary uni-dimensional constraints on $\{V_1, V_2, V_3\}$. The constraints, for an attribute A where $A \in \{N, C, F, S\}$ and the card *id*, are defined according to the following templates:

$A^{=}$: This constraint for the attribute A mandates that all three variables take the same value.

$A^{\neq}$: The all-different constraint for the attribute A requires that no two variables take the same domain value.

[4] We will relax this condition, sometimes, during reformulation.

[5] We find rather contrived their artificially lumping together two distinct objects, teachers and time, only to split them when the constraints are being enforced. We find that the game of SET is a more natural illustration of a multi-dimensional CSP than the examples provided in the original paper.



Attribute/val		C_1	c_2	C_3	C_4	c_5	c_6	c_7	c_8	c_9
Number	*1*	0	1	1	0	0	0	1	0	1
	2	0	0	0	1	0	0	0	0	0
	3	1	0	0	0	1	1	0	1	0
Color	*r*	1	1	0	0	0	1	0	1	1
	g	0	0	1	0	1	0	0	0	0
	p	0	0	0	1	0	0	1	0	0
Filling	*f*	1	0	0	1	1	1	1	0	0
	e	0	1	1	0	0	0	0	1	1
	s	0	0	0	0	0	0	0	0	0
Shape	*s*	0	0	1	0	0	1	0	0	0
	o	1	0	0	0	0	0	1	1	1
	d	0	1	0	1	1	0	0	0	0

Figure 4: The domain table of a nine-cards example.

The constraints are:

1. $N^{=} \oplus N^{\neq}$: All three cards have the same number or they have three different numbers.
2. $C^{=} \oplus C^{\neq}$: All three cards have the same color or they have three different colors.
3. $F^{=} \oplus F^{\neq}$: All three cards have the same filling or they have three different fillings.
4. $S^{=} \oplus S^{\neq}$: All three cards have the same shape or they have three different shapes.
5. $id^{\neq}$: A solution set consists of three (distinct) cards.

The set of constraints is thus:

$$\mathcal{C}_o = \{(N^{\neq} \oplus N^{=}), (C^{\neq} \oplus C^{=}), (F^{\neq} \oplus F^{=}), (S^{\neq} \oplus S^{=}), id^{\neq}\} \quad (1)$$

Figure 5 shows the constraint network of our model.

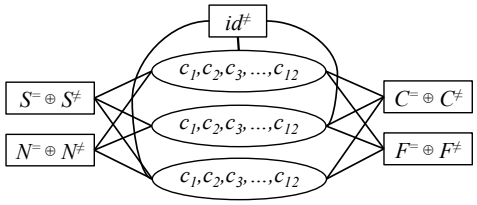


Figure 5: The constraint network of the CSP of a SET instance.

Generally speaking, constraints can be specified either in extension or in intension (i.e., as a predicate function on the scope of the constraint). In extension, they are a list of either acceptable tuples (supports) or forbidden tuples (conflicts). Given the number of variables of the constraint model (i.e., three), implementing the constraint in intension seems to be the simplest choice. Indeed, if the constraints were to be implemented in extension, the constraints can be built once and for all (domains size equal to 81) or dynamically every time a CSP is formed to be solved (domains size in [12,21] for a regular game). The number of supports to generate and store for a domain of 81 cards is $P(81, 3) = \frac{81!}{(81-3)!} = 511920$. Otherwise, it is between: $P(12, 3) = \frac{12!}{(12-3)!} = 1320$ and $P(21, 3) = \frac{21!}{(21-3)!} = 7980$. Although the number of tuples of the remaining constraints are significantly smaller (less

than 3^3), it is obviously more cost effective to implement all constraints in intension.

3.2 Solving the Constraint Model

We implemented a simple backtrack search to find all the solution sets in a given configuration of any number of playing cards. In order to prevent search from generating solutions that can be obtained from other solutions by simple permutation of the cards over the three variables, we have added to the model a symmetry-breaking constraint based on lexicographical ordering of the cards unique identifiers. Given that the constraints on the attributes have two mutually exclusive components (see Expression (1)), our backtrack search implements a convenient combination of forward checking and back-checking (Prosser 1993).

- The symmetry breaking constraint, the $id^{\neq}$ constraint, and all equality constraints ($A^{=}$) are enforced by forward checking.
- The four all-different constraints ($A^{\neq}$) other than $id^{\neq}$ are enforced by back-checking.

The depth of the search tree is at most three, one for each variable. When the first variable is instantiated, the only checkable constraints are $id^{\neq}$ and the symmetry breaking one. They are enforced by forward checking. After instantiating the second variable, we can determine by back-checking, for each of the remaining four attributes, which of the two constraints (equality $A^{=}$ or inequality $A^{\neq}$) holds between the first two variables. The constraint that applies is selected and the other one is "switched off." At this point, if any equality constraint on the current variable is "switched on," it is enforced by forward checking. The domains of the third variable is consistent with all applicable equality constraints. If any all-different constraint $A^{\neq}$ is applicable (i.e., switched on), then back-checking is applied to consider only consistent instantiations. Generally speaking, and depending on the applicable constraints, one may be able to improve the performance of search by enforcing higher consistency levels during search.

4 Constraint Model Reformulation

In this section, we reformulate our constraint model for SET by exploiting the multi-dimensionality of the domains. First, we describe how we reformulate the CSP by partitioning the domains of the variables based on the values of a given domain dimension. We show that this process yields, in the case of the game of SET to a disjunctive decomposition of the CSP producing a set of CSPs at each reformulation step, which we illustrate with two examples.

4.1 Model Reformulation

Each reformulation in Figure 1 step consists in partitioning the domains of the variables according to one of the four domain dimensions and enforcing the constraint relative to that dimension. We describe the process via an example.

Consider the example of six-card game of Figure 6. In this example, all cards are red and have an empty filling. The only constraints left enforce are thus $N^{=} \oplus N^{\neq}$, $S^{=} \oplus S^{\neq}$, and $id^{\neq}$. Considering first the dimension 'number,' we partition the variables' domains into equivalences classes based on the values of the chosen dimension, *number*, as shown in Figure 7. This operation corresponds to domain partition-

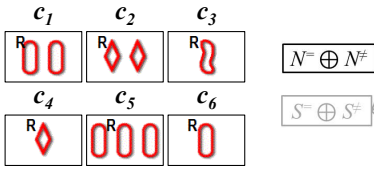


Figure 6: Simple example.

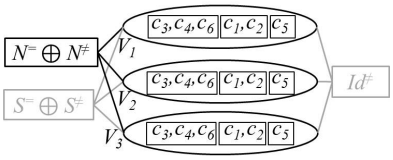


Figure 7: CSP model.

ing by value meta-interchangeability (Freuder 1991). All subproblems can be generated by a Cartesian product of the domain partitions. However, enforcing the constraint $N^{=} \oplus N^{\neq}$ and ignoring symmetrical subproblems[6] yield a decomposition of the CSP into only four CSPs: one for each domain partition enforcing $N^{=}$ (Figure 8) and one where the domains have all-different partitions enforcing $N^{\neq}$ (Figure 9). The solutions of the CSP in Figure 7 are partitioned

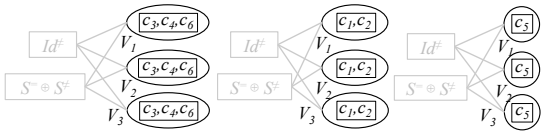


Figure 8: Domain dimension: *number*; values: 1, 2, and 3.

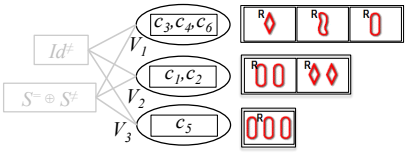


Figure 9: Variables have all different numbers.

over the four problems generated in this manner. The above process can be applied for each of the four domain dimensions.

4.2 Reformulation strategy for SET

The strategy of Figure 1 exploits the one-dimensionality of the constraints of the CSP model where each one-dimensional constraint is applied to the CSP in sequence, one at a time. For SET, we showed how exploiting value meta-interchangeability for a given dimension and enforcing the constraint relative to that dimension decomposes the CSP into four subproblems whose solution sets do not overlap. Combining the reformulation strategy of Figure 1 and the above domain partitioning by value interchangeability yields the reformulation tree of Figure 10. In this figure, A_i denotes a domain dimension, a, b, c domain partitions, and

[6] Subproblems that can be obtained by permutation of the domains are ignored.

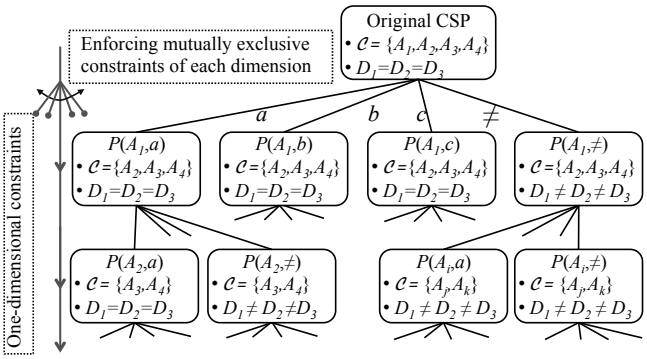


Figure 10: One-dimensional constraints are considered in sequence; mutually exclusive constraints are considered in parallel.

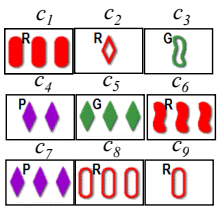
$P(A_i, a)$ the subproblem resulting from enforcing the constraint relative to A_i and a. Two important questions arise in general:

1. *Which dimension (i.e., attribute) to choose at each step?* Naturally, one should reduce the branching factor by choosing, for example, the dimension that yields the smallest problem, the most symmetries, the largest domain partitions, etc.
2. *Which subproblems are generated at each branching step?* The decomposition of Section 4.1 depends on the interchangeability and constraint types that hold for the considered dimension. Generalizing this decomposition for all types of symmetries and constraints requires further investigation.

In Section 5.2, we answer the above two questions for the game of SET. Our approach is based on the analysis of the domain table shown in Figure 4. Below, we motivate that approach with two examples, see Sections 4.3 and 4.4. All generated subproblems have the same set of variables. They differ in the constraints and the variables' domains. To generate the children of a given problem in the tree of Figure 10, we need to specify the set of constraints and the domain set of each child. The constraints set of a child subproblem is that of the parent minus the constraints of the attribute used in the branching. The domains of a child is smaller than those of its parent, which is the main incentive for the decomposition. As for the set of domains of a child, we distinguish the case where the enforced constraint at the branching step is an equality or an inequality constraint. The former keeps all domains equal whereas the latter yields a new problem where variables have different domains. Below, we illustrate the above on two examples.

4.3 Branching on equality constraints

In Figure 11, we consider the example of Figure 4 but report the domain table in a more compact form. Focusing on the dimension *filling*, we notice that *filling* takes only two values e and f. We conclude that we should form only two CSPs: one for the cards with an *empty* filling (i.e., $\{c_2, c_3, c_8, c_9\}$) and the other for the cards with a *full* filling (i.e., $\{c_1, c_4, c_5, c_6, c_7\}$). Because there is no card



$D_1 = D_2 = D_3$	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
Number	3	1	1	2	3	3	3	3	1
Color	r	r	g	p	g	r	p	r	r
Filling	f	e	e	f	f	f	f	e	e
Shape	o	d	s	d	d	s	d	o	o

Figure 11: The compacted domain table of the example of Fig. 4.

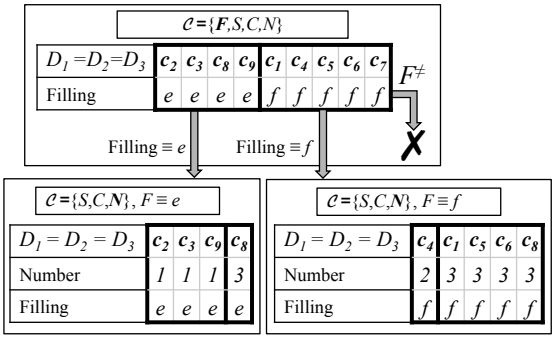


Figure 12: Branching on equality constraints $F^{=}$.

with a *striped* filling, we should not generate (1) a subproblem for this third possible value or (2) a subproblem where $F^{\neq}$ holds. The two generated subproblems have one fewer constraint than their parent and the domains of the variables in each subproblem are the same. The tree of Figure 12 illustrates the partitioning of the cards $c_1, \ldots, c_9$ as described above, then the partitioning of $\{c_2, c_3, c_8, c_9\}$ and $\{c_1, c_4, c_5, c_6, c_7\}$ given the domain dimension *number*. To detect the above-described situation, our algorithm examines the (detailed) domain table shown in Table 1. Column l

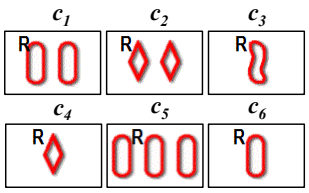
Table 1: Cell 10l shows that no card with a *striped* filling exists.

	a	b	c	d	e	f	g	h	i	J	k	l
1	Attribute/val		c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	Σ
2		1	0	1	1	0	0	0	1	0	1	4
3	Number	2	0	0	0	1	0	0	0	0	0	1
4		3	1	0	0	0	1	1	0	1	0	4
5		r	1	1	0	0	0	1	0	1	1	5
6	Color	g	0	0	1	0	1	0	0	0	0	2
7		p	0	0	0	1	0	0	1	0	0	2
8		f	1	0	0	1	1	1	1	0	0	5
9	**Filling**	e	0	1	1	0	0	0	0	1	1	4
10		**s**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
11		s	0	0	1	0	0	1	0	0	0	2
12	Shape	o	1	0	0	0	0	0	1	1	1	4
13		d	0	1	0	1	1	0	0	0	0	3

sums up the number of cards in the domain for a given attribute value (represented as a row in the table). The null entry in Cell 10l indicates that there is no card with a *striped* filling (Row 10) in the domain, and that selecting *filling* as a dimension for reformulation would yield only two subproblems and not four.

4.4 Branching on an inequality constraint

The example of Figure 13 shows the compacted domain table of the six-cards example of Figure 6. The remaining one-



$D_1 = D_2 = D_3$	c_1	c_2	c_3	c_4	c_5	c_6
Number	2	2	1	1	3	1
Color	r	r	r	r	r	r
Filling	e	e	e	e	e	e
Shape	o	d	s	d	o	o

Figure 13: The compacted domain table of the example of Fig. 6.

dimensional constraints are over *shape* and *number*. Here, we can choose either dimension *shape* or *number* for the reformulation step. Choosing the dimension *number* partitions the variables' domains into three sets of cards: $\{c_3, c_4, c_6\}$, $\{c_1, c_2\}$, and $\{c_5\}$ with *number* values 1, 2, and 3, respectively, see Figure 14. The subproblems with the domains

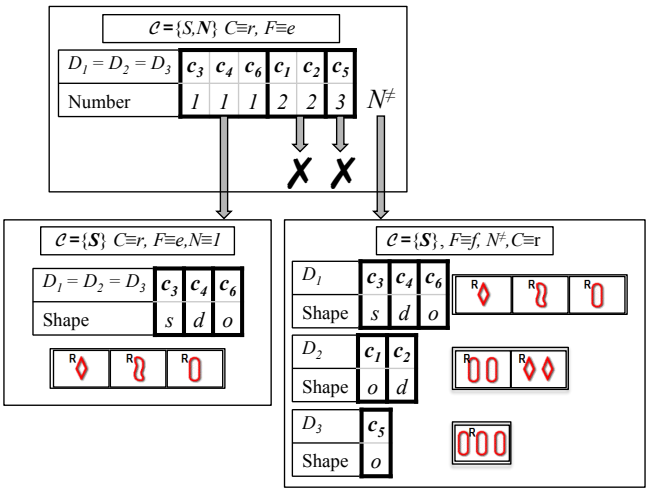


Figure 14: $D_1 \neq D_2 \neq D_3$ when branching on $N^{\neq}$.

$\{c_1, c_2\}$, and $\{c_5\}$ do not have the three required cards for a solution set, and are unsolvable. Our algorithm realizes that fact by checking on the values in cells $3l$ and $4l$ of Table 2.[7] Now, in the subproblem where $N^{\neq}$ holds, the three CSP variables have all-different domains, one for each value of the attribute *number* (i.e., 1, 2, and 3). None of these domains is empty in this subproblem. Importantly, note that when the domains of a CSP in some node of the tree in Figure 10 are all different (i.e., $D_1 \neq D_2 \neq D_3$), the same also holds for all its children in the tree. In summary, because the entries in $3l$ and $4l$ of Table 2 are in $[1, 3)$ we do not generate subproblems with the domains $\{c_1, c_2\}$ or $\{c_5\}$ but we generate the subproblem for $N^{\neq}$.

5 Reformulation Algorithm

Below we describe our reformulation algorithm for the game of SET. It is motivated by the examples of Figures 12 and 13

[7] Only two domain dimensions are shown in Table 2, reflecting the constraints applicable on the problem on top of Figure 13.

Table 2: Cells $3l$ and $4l$ indicate that the subproblems for values 2 and 3 need not be generated.

	a	b	c	d	e	f	g	h	l
1	Attribute/val		c_1	c_2	c_3	c_4	c_5	c_6	Σ
2	Number	1	0	0	1	1	0	1	3
3		2	1	1	0	0	0	0	2
4		3	0	0	0	0	1	0	1
11	Shape	s	0	0	1	0	0	0	1
12		o	1	0	0	0	1	1	3
13		d	0	1	0	1	0	0	2

and implementing the general strategy of Figure 1.

5.1 Decomposition Tree

The nodes of the tree Figure 10 maintain the following information about the subproblem at the node:

1. *Set of variables*: The set of variables and their domains.
2. *Set of constraints* $\mathcal{C}$: The set of applicable constraints. The one-dimensional constraints are reduced by one from a parent to a child.
3. *Domain table*: The table describing the multi-dimensional domain such as the one shown in Figure 4 to which we add the column indexed l as shown in Tables 1, 2, and 3. The l column sums up the values of the c_i's in each row in the corresponding table. It indicates the number of cards with the corresponding attribute value. This domain table is useful for choosing the attribute to branch on as illustrated in Sections 4.3 and 4.4. Note that only the

Table 3: Domain table.

	a	b	c	d	e	f	g	h	i	j	k	l
1	Attribute/val		c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	Σ
2	Number	1	1	0	0	0	0	1	0	0	0	2
3		2	0	1	0	0	1	0	1	0	1	4
4		3	0	0	1	1	0	0	0	1	0	3
5	Color	r	1	0	0	0	0	1	0	0	0	2
6		g	0	1	0	1	1	0	0	0	1	4
7		p	0	0	1	0	0	0	1	1	0	3
8	Filling	f	1	0	0	0	0	0	0	1	1	3
9		e	0	1	0	0	0	0	0	0	0	1
10		s	0	0	1	1	1	1	1	0	0	5
11	Shape	s	1	0	0	0	0	0	0	0	1	2
12		o	0	1	0	1	1	1	0	0	0	4
13		d	0	0	1	0	0	0	1	1	0	3

Table 4: Summary of domain table.

	a	b	c	d	e	f	g	h
1	Attribute/val		Σ_{D1}	Σ_{D2}	Σ_{D3}	Π_{Di}	Σ_{Di}	$\Pi\Sigma_{Di}$
2	Number	1	1	1	0	0	2	24
3		2	1	1	2	2	4	
4		3	1	1	1	1	3	
5	Color	r	1	0	0	0	1	16
6		g	1	2	1	2	4	
7		p	1	1	2	2	4	
8	Filling	f	1	0	2	0	3	12
9		e	1	0	0	0	1	
10		s	1	3	1	3	4	
11	Shape	s	1	0	1	0	2	24
12		o	1	3	0	0	4	
13		d	1	0	2	0	3	

entries corresponding to the applicable constraints need to be represented and updated. Entries corresponding to constraints enforced in an ancestor node in the decomposition tree are omitted as in Table 2.

4. *Summary of domain table*: When the variables' domains are all different (i.e., $D_1 \neq D_2 \neq D_3$), we generate an additional table that is the *summary of the domain table* as shown in Table 4 Here again, we keep and maintain

only entries (i.e., rows) corresponding to applicable constraints. Columns c, d, and e sum up the number of cards with a given attribute value in the corresponding domain. Columns f and g are the product and sum, respectively, of the entries in columns c, d, and e in the same row. Finally, column h is the product of the values in column g for the same attribute. The heuristics introduced below justify the use of this summary information.

5.2 Flow Chart

The flow chart of the algorithm is shown in Figure 15 and operates under the following assumptions:

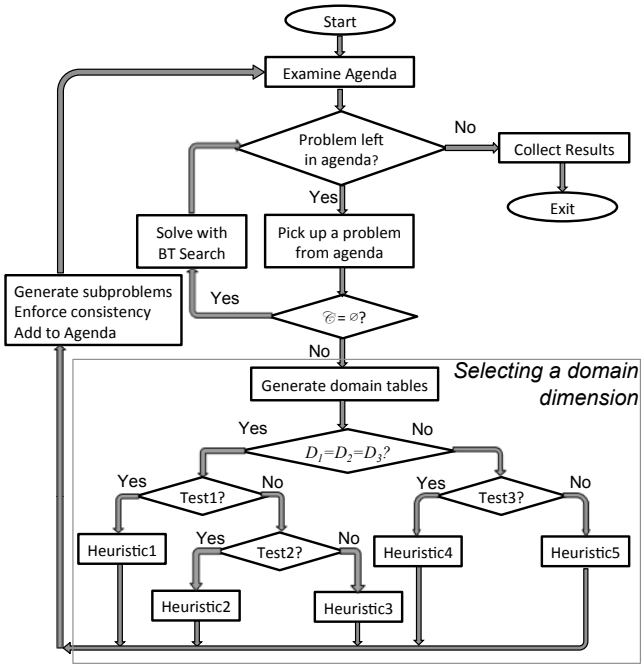


Figure 15: Our reformulation algorithm.

- An agenda is used to keep track of all 'open problems.' At the start, the initial problem is placed on the agenda. The agenda can be managed in according to any strategy.
- Solving with BT search indicates finding all the solutions to a subproblem using the search procedure described in Section 3.2. The solution sets found are stored in some unspecified data structure.
- One may choose to enforce some level of consistency on each generated subproblem before placing it the agenda. Subproblems that are deemed unsolvable are not added to the agenda.

The algorithm uses the following test conditions and heuristics. Below, we distinguish problems with identical domains (i.e., $D_1 = D_2 = D_3$) and the others (i.e., $D_1 \neq D_2 \neq D_3$). When $D_1 = D_2 = D_3$, we examine only the domain table:

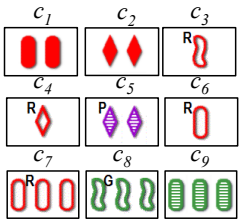
- *Test1*: When the column l in the domain table of the subproblem has a *null* entry, the two implications are enforced:
  1. No card has that dimension value, the corresponding subproblem need not be generated. For example, *striped* for dimension filling in Figure 12.
  2. For the same reason, we cannot form a subproblem where domains are all different. For example, $F^{\neq}$ in Figure 12 and $N^{\neq}$ Figure 14.
- *Heuristic1*: We branch on the attribute with the largest number of zeros in the column l because branching on this attribute yields the smallest number of new subproblems, and will not require creating an $A^{\neq}$ subproblem.
- *Test2*: When the value in column l for any value i for a dimension A is in $[1, 3)$, the A^i subproblem need not be generated. For example, values 2 and 3 for the dimension *number* in Figure 14.
- *Heuristic2*: We branch on the dimension that has the largest number of entries in column l that are less than three. When there are no null entries in column l for any dimension, all four subproblems must be generated.
- *Heuristic3*: We randomly choose any of the 'active' dimensions. This step requires generating all four subproblems.

When $D_1 \neq D_2 \neq D_3$, we consider only the table called 'summary of domain table' (see example in Figure 4). The columns c, d, and e indicate the size of domain D_1, D_2, and D_3. Their product is available in column$f$ of the table: it is zero if some domain is empty. Column g computes the number of cards that have the same value for the same dimension, the product for all the products for the same dimension is recorded in column h. A null value in column h indicates that the dimension needs to be chosen in priority because it would yield the generation of at most two subproblems.

- *Test3*: The entry in column h for a attribute A of the summary of domain table is equal to zero. In this situation, there is at least one value i for A that does not appear in any domain.
- *Heuristic4*: We branch on the attribute where column h is null, breaking ties in favor of the attribute with the largest number of null entries in column f. Note that when an entry in column f is null (for an attribute value i), the subproblem A^i and that where $A^{\neq}$ holds will have at least one empty domain, and need not be generated (or will be pruned).
- *Heuristic5*: We branch on the attribute where column f has the largest number of null entries. We generate all four subproblems, those for the attribute value i with a null entry in column f will have an empty domain and will be pruned.

The above three tests cover all cases, and yield a complete and sound algorithm for selecting a dimension for reformulation. Generating and storing the various tables is linear in the number of attribute values. Also, once a problem is decomposed, the corresponding tables can be discarded. The number of generated subproblems is $\mathcal{O}(|A|^{i+1})$ where $|A|$ is the number of attributes and i is the maximum number of attribute values.

Figure 17 shows the trace of a short execution of our algorithm on the nine-cards problem of Figure 16. This problem



$D_1=D_2=D_3$	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
Number	2	2	1	1	2	1	3	3	3
Color	r	r	r	r	p	r	r	g	g
Filling	f	f	e	e	s	e	e	e	s
Shape	o	d	s	d	d	o	o	s	o

Figure 16: An example with nine cards and one solution set.

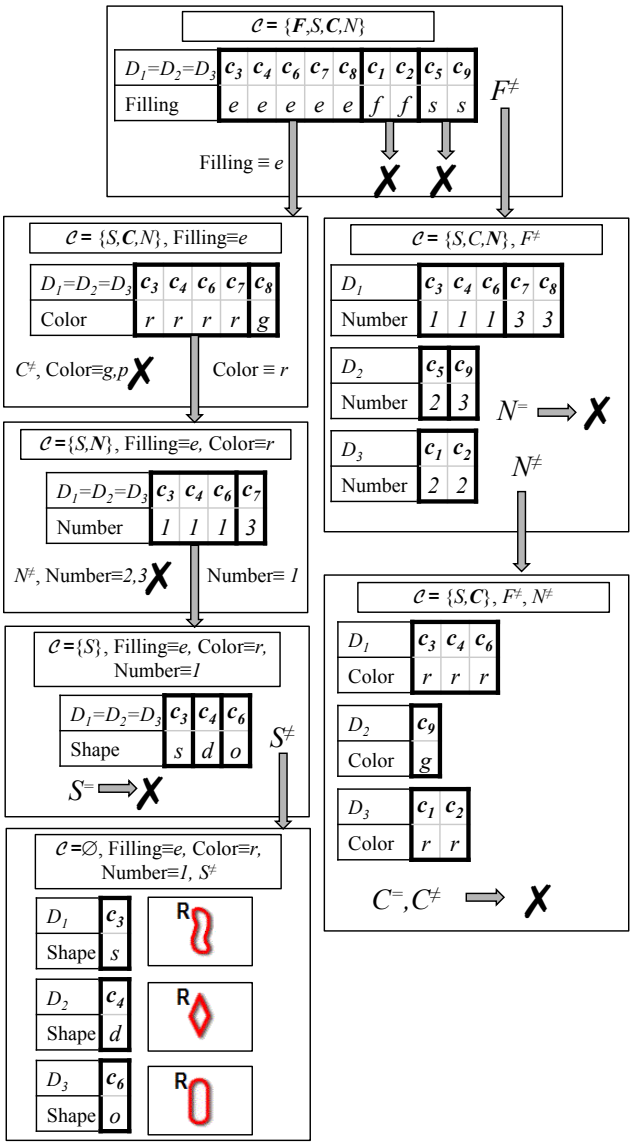


Figure 17: Illustrating the reformulation strategy.

instance has one solution. The algorithm proceeds by selecting sequentially the dimensions *filling*, *number*, and *shape*.

6 Results

To evaluate the effectiveness of our reformulation strategy, we run the following three algorithms on a set of 1000 random SET instances of domain size {3,4, ...,81}: Brute force, search (Section 3.2), and reformulation followed by search (Section 5). 'Brute force' is the algorithm with the three nested *for*-loops that generates all combinations of three cards then tests whether or not they satisfy the constraints. Figure 18 shows the number of constraint checks for increasing domain size. Figure 18 shows the number of constraint checks (#CC) for increasing domain size; Figure 19 shows the number of nodes visited (#NV) for increasing domain size; and Table 5 compares the performance of all three algorithms on two domain sizes (12 and 81 cards) displaying the average number of solutions, #CC, #NV, and CPU time.

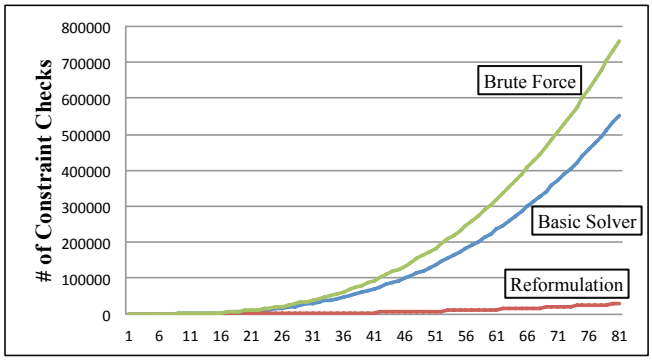


Figure 18: Comparing the numbers of constraint checks for increasing problem size.

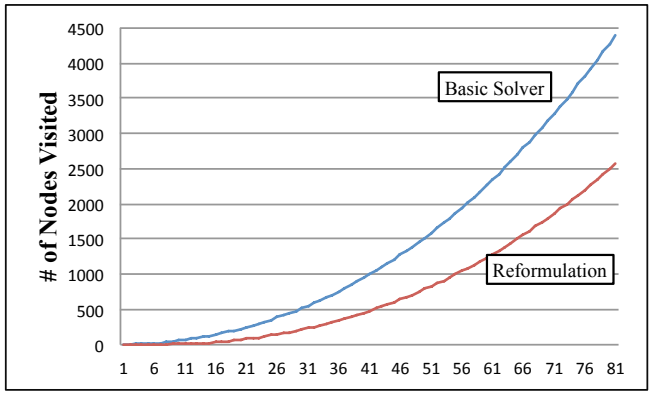


Figure 19: Comparing the number of nodes visited for increasing problem size. Brute force was not included as it overwhelmed the others.

The goal is more to compare the trends than the raw numbers. From those results, it is apparent that reformulation dramatically reduces the rapid growth, with increasing the domain size, of the number of constraint checks (Figure 18) and significantly that of nodes visited (Figure 19), thus demonstrating the benefits of our reformulations. The time measurements are not significant given the size of the

Table 5: Comparing the performance of three algorithms.

Algorithm	#Cards	#Sol	#CC	#NV	Time [msec]
Brute force	12	2.77	1956.80	220	**0**
Search			1726.653	80.77	62.46
Reformulation			**85.08**	**12.65**	5.85
Brute force	81	1080	758808	85320	**0**
Search			553365.00	4401.00	101.04
Reformulation			**31158.00**	**2565.00**	**39.44**

problem, the precision of the clock (i.e., 10 ms), and the time necessary for setting up the data structures for search. While it is true that the game of SET is a simple problem, we do believe that our techniques are widely applicable and will have significant impact on industrial size applications.

7 An Interactive Interface for SET

We have built a graphical user interface for SET (see Figure 20) that uses the two approaches for finding all solution sets for a given set of cards on the board. The interface allows us to compare our solvers performance in the context of a real game of SET.

The game can be played in two modes: "Single Player" and "Two Players." The game also features two automated solvers: by Backtrack Search (Section 3.2) and by Reformulation (Section 5.2). The users can play the game in CSP mode or non-CSP mode. If they are in CSP mode, they can see statistics about the current board (number of constraint check, CPU time, etc.) as well as switch between the two solvers. We have also provided a "2 Card Hint" button, which highlights two cards appearing in the same solution set, letting the user find the third. We also have a "1 Card Hint" button, which highlights one card that appears in a solution set, letting the user find the remaining two cards. If at any time there is no solution found for the twelve cards on the board, three more cards will flip open creating a board of up to 21 cards. If there are no solutions found in the cards displayed, the game is over, and the user can start again. Our applet is available online on `http://gameofset.unl.edu`.

8 Discussion & Related Work

Like most popular puzzles, the rules of SET are rather simple yet the game is quite addictive. We have found combinatorial puzzles to be effective vehicles to introduce the general public to the area of Constraint Processing (CP). They are also amazingly suitable tools to attract Computer Science students to study CP and train them in modeling, search, and constraint propagation techniques. For example, past students have developed constraint models and various propagation algorithms for Minesweeper[8] (Bayer, Snyder, and Choueiry 2006) and Sudoku[9] (Reeson et al. 2007). In addition to the educational benefits, our initiatives have inspired new research directions as documented in (Karakashian et al. 2010b; Woodward et al. 2011). Combinatorial puzzles have thus allowed us to serve all three tenants of the academic mission, namely, research, education, and outreach.

In (1995; 1997), Freuder and Sabin described abstraction and reformulation procedures for solving multi-dimensional CSPs, considering both multi-dimensional constraints and multi-dimensional domains. They evaluated them on n-queens problems in (1995; 1997), and on randomly generated problems with a controlled level of interchangeability in (1997). Our motivation and procedures significantly overlaps with theirs:

1. Their reformulation considers a single reformulation step, while ours is designed to accommodate any number of one-dimensional constraints for reformulation.
2. After reformulation and search, their technique includes a refinement step, which is not needed in our approach.
3. Finally, their experiments, conducted on random problems, do not provide the strikingly convincing results that our approach does.

As future work, we propose to investigate how to automate the selection of types of interchangeability exploited.

9 Conclusions & Future Work

In conclusion, we have created a reformulation strategy for multi-dimensional CSPs that shows a significant reduction in the search effort. We believe that the techniques we have proposed here can be applied to problems with more complex domains and with more of a real-world significance. In addition to building a graphical tool to explain problem solving by reformulation to students and the general public, there are several directions for future work that we would like to explore. The first is to build a general theory of reformulation for multi-dimensional CSPs that unifies the one proposed in (Freuder and Sabin 1997) and the one we have proposed above. The second is to investigate the applicability and usefulness of such a theory for general CSPs especially in light of the advances in the study of interchangeability and symmetry in CSPs (Freuder 1991; Gent, Petrie, and Puget 2006; Karakashian et al. 2010a). Finally, we would like to investigate whether the definition of a multi-dimensional CSP provided in (Yoshikawa and Wada 1992) needs to be revised to allow variable domains to be different.

Acknowledgments This material is based in part upon works supported by the National Science Foundation under Grant No. CCF-0747009, Grant No. CNS-0855139, and Grant No. RI-1117956, and by Science Foundation Ireland under Grant No. 05/IN/1886.

References

Bayer, K.; Snyder, J.; and Choueiry, B. Y. 2006. An Interactive Constraint-Based Approach to Minesweeper. In *Proc. of AAAI-2006*, 1933–1934.

Davis, B. L., and McLagan, D. 2003. The Card Game SET. *The Mathematical Intelligencer* 25 (3):33–40.

[8] `http://minesweeper.unl.edu`

[9] `http://sudoku.unl.edu`

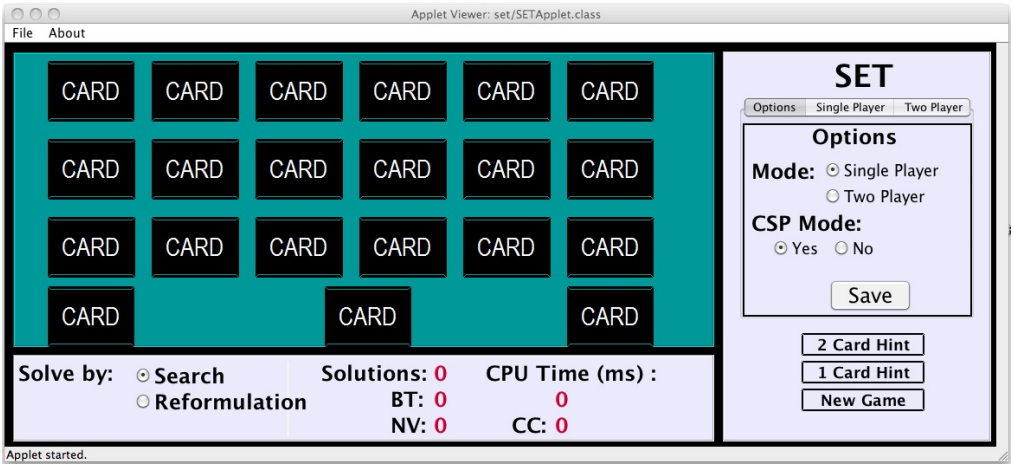


Figure 20: The graphical user interface.

Freuder, E. C., and Sabin, D. 1995. Interchangeability Supports Abstraction and Reformulation for Constraint Satisfaction. In *Symposium on Abstraction, Reformulation and Approximation, SARA'95*, 62–68.

Freuder, E. C., and Sabin, D. 1997. Interchangeability Supports Abstraction and Reformulation for Multi-Dimensional Constraint Satisfaction. In *Proc. of AAAI-97*, 191–196.

Freuder, E. C. 1991. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI-91*, 227–233.

Gent, I.; Petrie, K.; and Puget, J.-F. 2006. *Handbook of Constraint Programming*. Elsevier. chapter 10, 329–376.

Karakashian, S.; Woodward, R.; Choueiry, B. Y.; Prestwhich, S.; and Freuder, E. C. 2010a. A Partial Taxonomy of Substitutability and Interchangeability. In *International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon 10)*, 1–18.

Karakashian, S.; Woodward, R.; Reeson, C.; Choueiry, B. Y.; and Bessiere, C. 2010b. A First Practical Algorithm for High Levels of Relational Consistency. In *Proc. of AAAI-2010*, 101–107.

Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9 (3):268–299.

Reeson, C. G.; Huang, K.-C.; Bayer, K. M.; and Choueiry, B. Y. 2007. An Interactive Constraint-Based Approach to Sudoku. In *Proc. of AAAI-2007*, 1976–1977.

Woodward, R.; Karakashian, S.; Choueiry, B. Y.; and Bessiere, C. 2011. Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In *Proc. of AAAI-2010*, 112–119.

Yoshikawa, M., and Wada, S. 1992. Constraint Satisfaction in A Multi-Dimensional Domain. In *First International Conference on Artificial Intelligence Planning Systems (AIPS 92)*, 252–259.