PuzzleJAR: Automated Constraint-based Generation of Puzzles of Varying Complexity

September 30, 2014

Abstract

Engaging students in practicing a wide range of problems facilitates their learning. However, generating fresh problems that have specific characteristics, such as using a certain set of concepts or being of a given difficulty level, is a tedious task for a teacher. In this paper, we present PuzzleJAR, a system that is based on an iterative constraint-based technique for automatically generating problems. The PuzzleJAR system takes as parameters the problem definition, the complexity function, and domain-specific semantics-preserving transformations. We present an instantiation of our technique with automated generation of Sudoku and Fillomino puzzles, and we are currently extending our technique to generate Python programming problems. Since defining complexities of Sudoku and Fillomino puzzles is still an open research question, we developed our own mechanism to define complexity, using machine learning to generate a function for difficulty from puzzles with already known difficulties. Using this technique, PuzzleJAR generated over 200,000 Sudoku puzzles of different sizes (9x9, 16x16, 25x25) and over 10,000 Fillomino puzzles of sizes ranging from 2x2 to 16x16.

1   Introduction

Students learn by practicing many problems, but generating fresh problems that have specific characteristics, such as using a certain set of concepts or being of a given difficulty level, is a tedious task for a teacher. Our goal is to automatically generate programming problems that are parameterized by complexity and by the set of concepts a student wants to learn. In this work, we present PuzzleJAR, a system that solves a simpler task of automatically generating Sudoku and Fillomino puzzles of different complexity levels. Our technique and algorithms are general enough to generate programming problems as well as problems from other domains including algebra and trigonometry problems.

We present a generic iterative constraint-based algorithm for generating problems of different complexity levels. Most previous approaches for automatically generating puzzle problems have been specific to a given puzzle and are based on a set of heuristic rules. PuzzleJAR, on the other hand, lets one specify the puzzle definition and puzzle complexity in a declarative fashion using constraints and then uses efficient constraint-solving to incrementally solve constraints generated from different iterations. The system first generates a completely random problem that satisfies the constraints. It then removes elements from the complete problem using a user-defined probabilistic function such that certain validity constraints are satisfied. We use the z3 SMT solver [2] and its theory of linear arithmetic for representing and solving the constraints.

We have successfully used PuzzleJAR to automatically generate more than 200,000 9x9 Sudoku puzzles and more than 10,000 Fillomino puzzles. A random 9x9 Sudoku and Fillomino puzzle automatically generated by our system is shown in Figure. 1 and Figure. 2 respectively. Our puzzles vary across a number of features: number of empty spaces, number of solutions, distribution of empty spaces, repetition of digits etc. The declarative nature of PuzzleJAR lets us easily parameterize the algorithm to also generate puzzles of different sizes, such as 16x16 and 25x25 Sudoku puzzles. Since computing the difficulty level of a Sudoku puzzle or a Fillomino puzzle is still an open research problem, we resort to machine

learning techniques to learn a function over the puzzle features from a set of labelled Sudoku problems obtained from popular Sudoku websites and newspapers. We then use this learnt function to characterize the Sudoku problems generated by our system into different complexity levels. We are currently extending our system to support generation of Python programming problems and other puzzles.

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

(a)

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

(b)

Figure 1: (a) A 9x9 Sudoku problem automatically generated by our system, and (b) its solution.

5			5		7		4	
6		1	2	2			1	4
	6	7		6	7		3	1
6	7		1					3
	7		3	6				
1				3	6		4	4
5			1	7		5		5
4	4	5	4	4	7			2
					1			7

(a)

5	5	5	5	5	7	4	4	4
6	6	1	2	2	7	7	1	4
6	6	7	6	6	7	7	3	1
6	7	7	1	6	7	7	3	3
6	7	7	3	6	6	5	4	4
1	7	7	3	3	6	5	4	4
5	5	5	1	7	7	5	5	5
4	4	5	4	4	7	7	2	2
4	4	5	4	4	1	7	7	7

(b)

Figure 2: (a) A 9x9 Fillomino problem automatically generated by our system, and (b) its solution.

This paper makes the following major contributions:

- We present a general constraint-based system, PUZZLEJAR, to automatically generate puzzles of varying complexity.

- We present a machine learning approach to learn the complexity function of puzzles.

- We successfully evaluate the PUZZLEJAR system to generate more than $200,000$ Sudoku puzzles and $10,000$ Fillomino puzzles of different complexity.

- We have created an interactive website to let users solve these puzzles. To the best of our knowledge, the number of Sudoku and Fillomino puzzles are at least an order of magnitude more than the puzzles present on any website/book.

2 Materials and Methods

2.1 Overview of PuzzleJAR

We first present an overview of the PUZZLEJAR system to synthesize puzzles of varying complexity levels. PUZZLEJAR takes three components as input: (i) a declarative definition D of the puzzle $\mathcal{P}$, (ii) a complexity function C, and (iii) a set of transformation functions $\widetilde{T}$. In this section, we formally define these components and present our general synthesis algorithm to automatically generate puzzles of varying complexity.

2.1.1 Definitions

Definition 1 (Puzzle). *A two dimensional puzzle board of size $n \times m$ is defined using a valuation function $\mathcal{P} : \mathbb{N} \times \mathbb{N} \to \mathcal{D}$, which assigns values to the squares on the puzzle board. The value of a square (i, j) on the puzzle board is denoted by $\mathcal{P}(i, j)$, where $1 \leq i \leq n$, $1 \leq j \leq m$, and $\mathcal{D}$ denotes the set of possible values the puzzle squares can take.*

Definition 2 (Declarative Definition). *The declarative definition of puzzle D defines constraints over the set of valid values $\mathcal{P}(i,j)$ that the puzzle squares can take.*

Definition 3 (Complexity). *The complexity function $C : \mathcal{P} \to H$ takes a puzzle board $\mathcal{P}$ as input and maps it to a finite class of hardness levels denoted by H.*

Definition 4 (Transformations). *A transformation function $T : \mathcal{P} \to \mathcal{P}$ takes a puzzle board as input and transforms it to another puzzle board such that the new puzzle board also satisfies the puzzle constraints D. The set of all transformation functions are denoted by $\widetilde{T}$.*

Algorithm 1 GenPuzzles(D, C, $\widetilde{T}$)

1: $\mathcal{P}_I := $ GetRandomPuzzle(D)
2: $\mathcal{P}_C := \mathcal{P}_I$
3: SquaresTested $:= \emptyset$
4: PuzzComplDict $:= \{\}$
5: **while** $|$SquaresTested$| \neq |\mathcal{P}_I|$ **do**
6:    $(i,j) = $ ChooseSquare($\mathcal{P}_C$)
7:    SquaresTested $= $ SquaresTested $\cup\, (i,j)$
8:    **if** isRemoveValid($\mathcal{P}_C, i, j$) **then**
9:

$$
\mathcal{P}'_C(k,l) = \begin{cases} \mathcal{P}_C(k,l) & \text{if } (k,l) \neq (i,j) \\ \phi & \text{if } (k,l) = (i,j) \end{cases}
$$

10:       $R := \emptyset$
11:       **for** $T \in \widetilde{T}$ **do**
12:          $R := R \cup T(\mathcal{P}'_C)$
13:       **end for**
14:       **for** $\mathcal{P} \in R$ **do**
15:          $h := C(\mathcal{P})$
16:          PuzzComplDict$[h] := $ PuzzComplDict$[h] \cup \mathcal{P}$
17:       **end for**
18:       $\mathcal{P}_C := \mathcal{P}'_C$
19:    **end if**
20: **end while**
21: **return** PuzzComplDict

2.1.2 Constraint-based Synthesis Algorithm

The algorithm first uses the `GetRandomPuzzle` function to generate an initial random puzzle board configuration $\mathcal{P}_I$ by solving the puzzle constraints D using an off-the-shelf constraint solver. It then starts emptying squares on the board one at a time until all the squares have been tested, i.e. the size of the set $|\texttt{SquaresTested}|$ becomes equal to the size of the puzzle board $|\mathcal{P}_I|$. The algorithm chooses the squares to be emptied using the `ChooseSquare` function, which takes the current puzzle board configuration $\mathcal{P}_C$ as an input. The `ChooseSquare` function uses a user-defined strategy to select a square that can vary from being completely random to a strategy that selects squares based on the distribution of the values of the current puzzle board. After a square is selected to be removed, the algorithm checks whether certain puzzle constraints are met after removing the chosen square using the `isRemoveValid` function. A common `isRemoveValid` function is to check if the current puzzle $\mathcal{P}_C$ has a unique solution, but PuzzleJAR allows for any general `isRemoveValid` function.

If the `isRemoveValid` function returns `True`, i.e. if we still get a valid puzzle after removing the square (i, j) from the puzzle $\mathcal{P}_C$, the algorithm creates a new puzzle $\mathcal{P}_C'$ that has the same square values as the puzzle $\mathcal{P}_C$ except the square (i, j) whose value is now set to ϕ (denoting an empty square). Often times, we can apply puzzle semanticss-preserving transformations to the puzzle boards to get new puzzle board configurations. The algorithm applies the set of transformations $\widetilde{T}$ to the new puzzle board $\mathcal{P}_C'$ to obtain a set of puzzle boards R. Finally, the algorithm computes the complexity of each puzzle board h using the complexity function C and assigns it to appropriate complexity level in the dictionary `PuzzComplDict`. This `PuzzComplDict` is the resulting dictionary that is returned by the `GenPuzzles` algorithm.

In general, it is difficult to provide a puzzle complexity function C that can assign a hardness level to a puzzle board. Even for relatively simpler puzzles such as the Sudoku, the complexity function is an open research question [3]. In PuzzleJAR, we try to approximate this complexity function using machine learning techniques. We obtain a set of labelled

training data that consists of a set of puzzle configurations each labelled with a hardness label h. Currently we use the puzzle data available in books/web/newspapers, but we plan to obtain such labelled training data from human subjects in future. For a puzzle, we define a feature vector consisting of a set of features that may be useful to capture its complexity. We then use Support Vector Machines to learn a function C that can map the feature vectors of the puzzles in the training set to their corresponding hardness levels.

PUZZLEJAR allows for any `isRemoveValid` function such as a function that checks whether the number of current solutions is less than a constant k. A general strategy to perform this check is to use an off-the-shelf constraint solver to first find a solution S to the puzzle, and then solve for another solution S' by adding an extra constraint that the new solution should not be the previous solution $S \neq S'$. For a value k, we get the constraint $S' \neq S_1 \wedge S' \neq S_2 \wedge \cdots \wedge S' \neq S_k$. This strategy needs $k + 1$ (potentially exponential) solver calls to check whether a square can be emptied from the puzzle board, which can make the overall algorithm quite expensive. For the common case of $k = 1$ (the constraint that the puzzle should always have a unique solution), we can perform this check efficiently using just a single solver call by adding a constraint that $S' \neq \mathcal{P}_I$, i.e. check whether there exists a solution S' that is different from the original puzzle board.

2.2   Case Studies

We now instantiate our general synthesis technique in PUZZLEJAR on two puzzles: Sudoku and Fillomino. For each puzzle, we present the three components: its declarative definition, the features for defining the complexity function, and the set of semantics-preserving transformations.

2.2.1   Sudoku

An example Sudoku problem generated by PUZZLEJAR together with its solution is shown in Figure. 1. The 9x9 Sudoku puzzle constraints are that each row, each column, and each

3x3 square should on the puzzle board should take distinct values from 1 to 9. A more formal definition for the Sudoku puzzle can be found in [13].

Declarative Definition

We use the python frontend of the z3 constraint solver in combination with list comprehension to specify the 9x9 Sudoku puzzle declaratively. As can be noticed from the encoding, it can be easily generalized to other Sudoku sizes, such as 16x16 or 25x25.

We first define 81 different integer variables (`X[0][0]`, `X[0][1]`, `...`, `X[8][8]`), where `X[i][j]` denotes the value of the Sudoku cell (i, j). We also define the valid set of values each element can take: $1 \leq$ `X[i][j]` ≤ 9 (valid values).

```
X = [[Int('x%d%d' % (i,j)) for i in range(9)] for j in range(9)]
valid_values = [And ( X[i][j] >= 1, X[i][j] <= 9) for i in range(9)
for j in range(9)]
```

We now add the Sudoku constraints that the values in each row should be distinct (`rows_distinct`), values in each column should be distinct (`cols_distinct`), and that values each 3x3 square should be distinct (`three_by_three_distinct`).

```
row_distinct = [Distinct(X[i]) for i in range(9)]
cols_distinct = [Distinct([X[i][j] for i in range(9)]) for j in
range(9)]
three_by_three_distinct = [ Distinct([X[3*k + i][3*l + j] for i in
range(3) for j in range(3)]) for k in range(3) for l in range(3)]
```

To encode partially filled Sudoku board (where a 0 value denotes an empty space), we simply add the constraint `X[i][j]` == `board[i][j]` when `board[i][j]` $!= 0$.

```
 already_set = [X[i][j] == board[i][j] if board[i][j] != 0 for i in
 range(9) for j in range(9)]
```

The complete set of Sudoku constraints is obtained by combining all previous constraints:

```
Sudoku_constraint = valid_values + row_distinct + cols_distinct +
three_by_three_distinct + already_set
```

Creating the Initial Puzzle

We use the z3 constraint solver to generate a Sudoku board in a 2 dimensional list such as the example shown in Figure. 1(b)

Emptying Squares

The next step in our algorithm is to start removing values from the board. For Sudoku, our method for selecting the next square to remove includes the following sub-steps:

1. **Select a square to empty in a probabilistic fashion.** For row N, we calculate the percentage of cells that have not been removed, P_N. We then randomly select a row i, where $0 \leq i \leq 8$, and generate a random number between 0 and 1. If this number is less than P_i, then we keep i as our selected row. If this number is greater than P_i, then we discard i and select a new random row and a random number. Once we have a number that is less than the percentage calculated we keep this row. We then follow the same process to find which column j in the row we should empty. We choose (i, j) as the square we wish to empty. The complete process allows for rows with more un-emptied cells to have a greater change of being chosen.

2. **Temporarily set the selected square as emptied.** We create a temporary board and a new set of z3 constraints with the changed value.

3. **Check whether the temporary board is valid.** If the temporary board has fewer than k solutions, we keep the board. Otherwise, the selected square is added to `vals_tried`, the list of squares that do not work, and we repeat steps 1 - 3 until we have a valid selected square.

4. **Permanently remove a valid square.**

5. **Repeat steps 1 through 4** until the target number of emptied squares is reached or the sum of the number of squares in `vals_tried` and the number that are already

Table 1: Symmetrical Transformations	
9x9 Sudoku Board	12x12 Sudoku Board
1. Relabeling the nine digits	1. Relabeling the twelve digits
2. Permuting the three 3x9 stacks	2. Permuting the four 3x12 stacks
3. Permuting the three 9x3 bands	3. Permuting the three 12x4 bands
4. Permuting the three rows within a stack	4. Permuting the three rows within a stack
5. Permuting the three columns within a band	5. Permuting the four columns within a band
6. Reflecting about the axes of symmetry in a square	6. Reflecting about the horizontal and vertical axes in a square
7. Rotation by 90 degrees	

emptied reaches 81.

Transformations

We are able to quickly generate more full Sudoku boards for emptying without using the SMT solver. To do this, we repeatedly apply mathematically symmetrical transformations to an already-created Sudoku board. Out of about 6×10^{21} total unique 9x9 Sudoku boards, these transformations can generate about 3×10^{6} new unique Sudoku boards from an existing one [1]. Furthermore, we can apply most or all of these transformations to larger boards. Table 1 shows the symmetrical transformations that can be applied to 9x9 and 12x12 boards.

As the last step of our generation algorithm, we perform these transformations again on an already emptied board to generate more emptied puzzles that possibly have a different number of solutions.

Defining Complexity

After a Sudoku puzzle is generated, we determine its difficulty using machine learning. For each puzzle we use, a 15-component vector is generated to describe the unsolved board. These components are (1) number of solutions; (2) number of empty squares; (3) number of rows with at least seven blank squares; (4) number of columns with at least seven blank squares; (5) number of 3x3 grids with at least seven blank squares; (6 - 14) number of occurrences of each digit; (15) standard deviation of number of occurrences of each digit from the mean number of occurrences.

The SVM library by scikit-learn [7] then uses the vector to categorize the puzzle into one of four difficulties: (1) Easy, (2) Medium, (3) Hard, and (4) Evil.

2.2.2 Fillomino

An example Fillomino problem generated by PUZZLEJAR together with its solution is shown in Figure. 2. The Fillomino puzzle constraints are that the puzzle board should be divided into regions such that the square value of the cells in a region should all have the same value, where the value is equal to the size of the region. A more formal definition for the Sudoku puzzle can be found in [12].

Declarative Definition

Using the z3 Python frontend, we declaratively define a Fillomino puzzle. First, we define an NxN board and assert that only the values between 1 and N can be on the board:

```python
cells = [[Int("x%d%d" % (i,j)) for i in range(1,N+1)] for j in
         range(1,N+1)]
valid_cells = [And(cells[i][j] <= N, cells[i][j] >=1) for i in
         range(N) for j in range(N)]
```

Now we must assert the definition of a Fillomino puzzle, that the value of all of the squares in a specified region must be the same as the number of squares in that region. We

do this using the concept of spanning trees in the graphs, with each vertex being a cell on the board and each edge representing some relationship between the cells.

We define each `edge_val` to be a directed edge between a cell and one of its adjacent cells. If `edge_val == 1`, there exists an outgoing edge from that cell to an adjacent square, and if `edge_val == 0`, there exists no such edge. We constrain it so that outgoing edges exist only between two cells that are in the same region.

```
for i in range(N):
    for j in range(N):
        for (k,l) in getAdjacent1(i,j):
            edge_var[(i,j,k,l)] = Int("e%d%d%d%d" % (i,j,k,l))
edge_val_constraints = [Or(edge_val==0,edge_val==1) for edge_val in
    edge_var.values()]
```

In our construction of the directed graph, there can be at most one outgoing edge from every square, meaning that the sum of all `edge_val` for a cell will always be less than or equal to 1.

```
for i in range(N):
   for j in range(N):
      for (k,l) in getAdjacent1(i,j, N):
         if lessThan(i,j,k,l):
            sum_edges = edge_var[(i,j,k,l)] + edge_var[(k,l,i,j)]
                edge_val_constraints.append(sum_edges <= 1)
```

We then add the constraint that if there is an edge between two cells, those two cells are in the same regions; therefore, they should have the same value. Finally, we define a cell size as being the sum of the sizes of the adjacent cells + 1. In each region, we add the constraint such that there is exactly one cell with a size equal to its value.

Creating the Initial Puzzle

We have two ways of creating the initial full board. We generate a full board with z3 using the declarative definition of a Fillomino puzzle, but this code is slow and not random in z3. Our second option is to create a board only using a Python program. The Python program

works by selecting a starting square, randomly selecting a sequence length from a list of valid sequence lengths, and then setting a list of squares that are sequence length long and that all have the value sequence length. We continue with this selection process until the board is completely filled.

Emptying Squares

We adapt our emptying algorithm for Fillomino.

1. **Select a square to empty.** Choose a square from a region that has more than one cell that is not emptied. Unlike in Sudoku, we do not check probabilistically for squares in rows, columns, or three by three grids that have more un-emptied squares because the only constraint in Fillomino is that there must be at least one value in every region that is not emptied.

2. **Temporarily set the selected square as emptied, check whether the temporary board is valid, and permanently remove a valid square.** These steps are the exact same as steps 2 - 4 of Sudoku emptying.

3. **Repeat steps 1 and 2** until the target number of emptied squares is reached or the sum of the number of squares in `vals_tried` and the number that are already emptied reaches N^2.

Transformations

Because of the region constraints, there are less transformations that can be applied to Fillomino than Sudoku, but a number still exist. These are (1) Rotation, (2) Vertical reflection, and (3) Horizontal reflection. These transformations be applied to an emptied board to get 7 new Fillomino boards with different orientations form the original one.

Defining Complexity

Using machine learning, we determine the difficulty of an unsolved Fillomino puzzle based on its 4-component characterizing vector. The components are (1) number of cells; (2) number of empty squares; (3) number of regions; (4) optimality, whether the puzzle can be further emptied. The function generated by the SVM categorizes a Fillomino puzzles into four difficulties similar to those of Sudoku puzzles.

3 Results and Discussions

We tested PUZZLEJAR with two experiments on different components of the algorithm: (i) the scalability of the puzzle generation and emptying steps, and (ii) the reliability of our machine learning technique in determining difficulty of a puzzle

3.1 Scalability of PuzzleJAR

To test the scalability of our puzzle generation algorithm in PUZZLEJAR, we generated 16x16 and 25x25 Sudoku boards to compare with the standard 9x9 boards and all square Fillomino boards from sizes 2x2 to 16x16.

When we empty a puzzle with dimensions NxN, we set as a parameter the target number of cells that the program would aim to empty from an initially full board. When the program reaches a point where it cannot further empty any more cells (i.e., the emptying of any remaining cell would cause the board to have more than the maximum number of allowable solutions), the emptying process will be considered finished. Because of this decision, we observe stagnation in our run times as the target number of empty cells is increased beyond a certain threshold.

Figure. 3 shows program running times to create and empty 9x9, 16x16, and 25x25 Sudoku puzzles as the target number of empty squares is increased. There is an exponential increase in run time as the number of possible empty spaces increased. Even for 25x25 Sudoku

puzzles, we find that the time required to generate a Sudoku puzzle is quite reasonable (about 500 seconds).

We can see in Figure. 3 that the generation time for 9x9 puzzles did not continue to increase after the target number of empty cells was raised beyond 60 because the emptying had stopped before the target of 60+ empty cells had been reached.

A similar experiment with Fillomino puzzles demonstrates a similar pattern of stagnation after a threshold. Unlike the run times of Sudoku puzzles, however, the run times for Fillomino puzzles follow a linear trend, not an exponential trend, as the target number of empty spaces is increased. This linear trend could be a result of more constraints in the z3 declaration of Fillomino in the z3 declaration of Sudoku, which allows for fewer iterations.
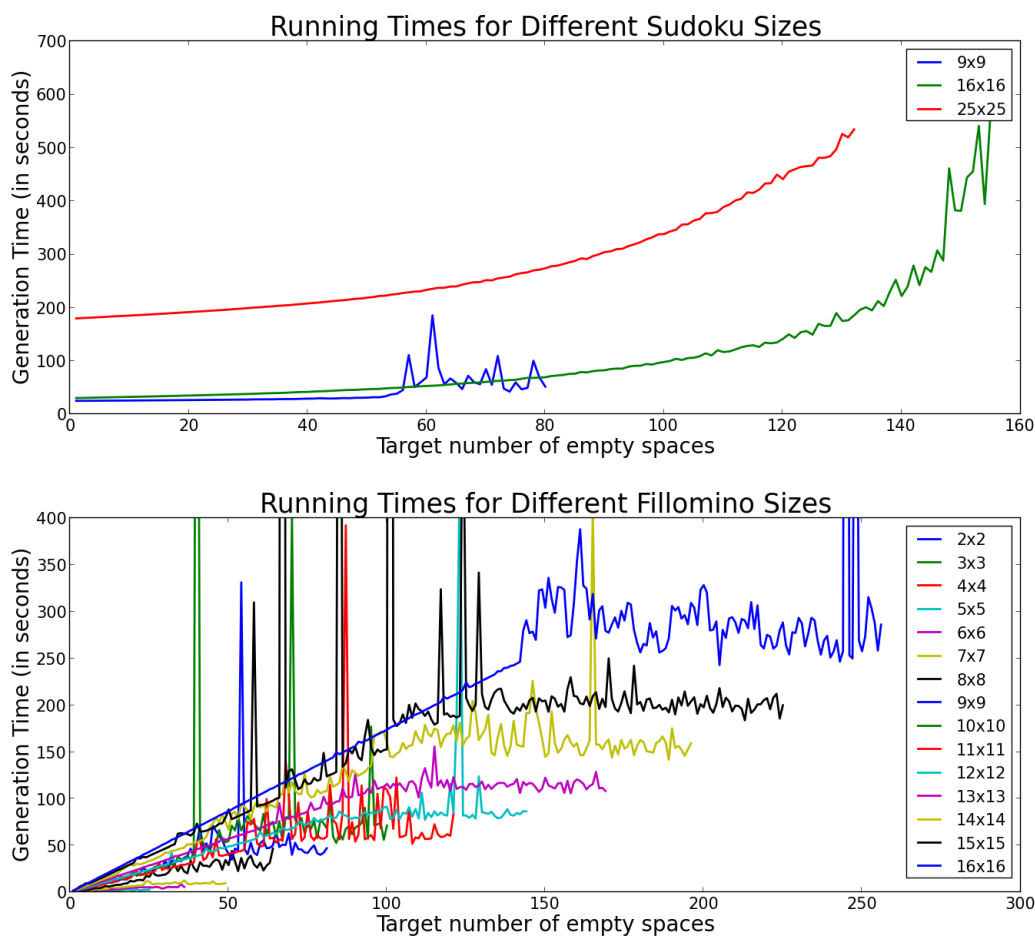

Figure 3: Running Times by Target Number of Empty Spaces (a) Sudoku (b) Fillomino.

3.2 Machine Learning for Puzzle Complexity Function

Our method of determining puzzle difficulty was to use machine learning to categorize a puzzle's characterizing vector. To test the reliability of this approach, we recorded published puzzles and their respective difficulty levels from online puzzle providers. This database of puzzles was randomly divided into two sets: a training set and a testing set. The training set was used to generate the SVM's categorization function, and the testing set was used to generate the "success rate": the percentage of puzzles in the testing set whose SVM-determined difficulty matched the difficulty assigned by the puzzle providers.

For our Sudoku database, we recorded 206 puzzles from Web Sudoku, the largest online Sudoku puzzle provider. For our Fillomino database, we recorded 40 puzzles from Math In English, a puzzle website that had categorized puzzles by difficulty levels similar to those of Web Sudoku: (1) Easy, (2) Moderate, (3) Challenging, and (4) Super Difficult. Figure. 4 shows average success rates over 500 trials as the percentage of puzzles in the training set was increased. Our results show that as we increased the number of puzzles in the training set, the success rate increased to 80%.
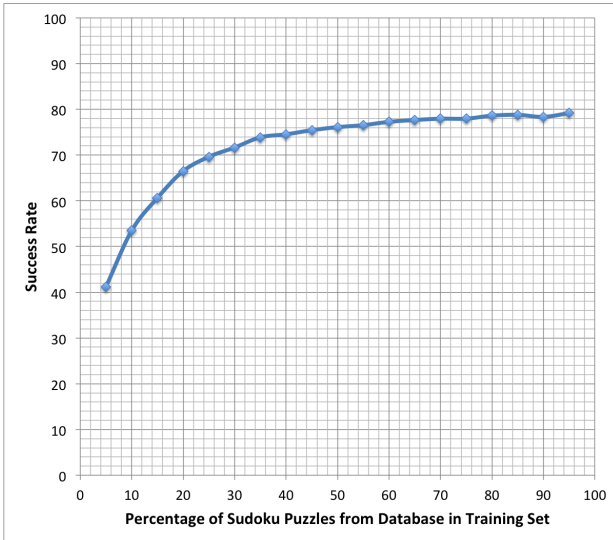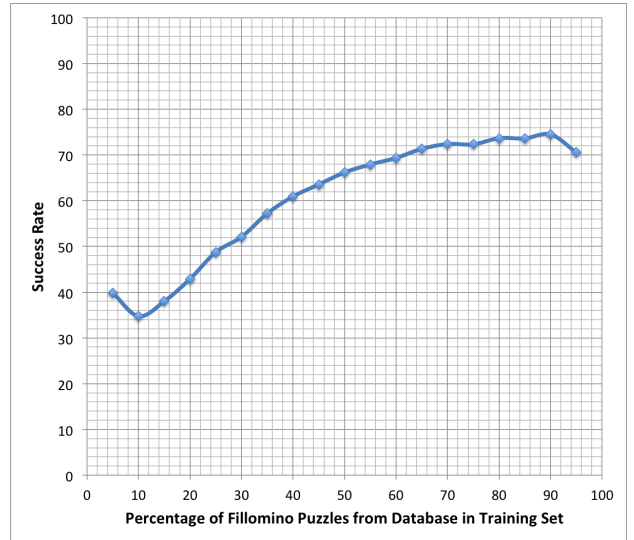
(a)   (b)

Figure 4: Success Rates (a) Sudoku (b) Fillomino.

4   Related Work

The large scale of students in popular courses have forced researches to develop new auto-
mated technologies to solve problems such as automated feedback generation [9] and solution
generation [5]. Another important problem resulting from this scale is that of automated
problem generation to cater to the practice needs of different students as well as for providing
different exams to students of a given difficulty level, which our work addresses.

There has been some previous work on generating new problems in various domains,
namely algebra and programming. The work on generating new algebra problems has mostly
been looked upon using two main approaches. In the first approach, a teacher is provided
a certain set of parameter values that are fixed for a given domain [6]. For example, for
generating a quadratic equation, the parameters can be the number of roots, difficulty of
factorization, whether there is an imaginary root, the range of coefficient values etc. Given
a set of feature valuations, the tool generates the corresponding quadratic equations. The
second approach takes a particular proof problem, and tries to learn a problem template from
the problem which is then instantiated with different concrete values [8]. The system first
tries to learn a general query from a given proof problem, which is then executed to generate
a set of proof problems. Since the query is only a syntactic generalization of the original
problem, only a subset of them are valid problems, which are identified using polynomial
identity testing. Our approach, however, creates different versions of the same problem
by introducing different number of holes in the original problem based on a parametric
complexity function.

More recently, a technique was proposed to generate fill-in-the-blank Java problems where
certain keywords, variables and control symbols are removed randomly from a correct solu-
tion [4]. The technique blanks variables using the condition that at least one occurrence of
each variable remains in the scope and blanks control symbols such that at least one occur-
rence of a paired symbol (such as brackets) remains. Our technique, on the other hand, is
more general since it is constrained-based and can check for more interesting constraints such

as unique solutions. It also can capture the notion of problem difficulty using complexity functions.

5   Future Work

We are currently working on extending PUZZLEJAR to support automated generation of Python programming problems. Since our generic algorithm is parametric with respect to problem definition, complexity function definition, and solving algorithm, we just need to instantiate these components for generating Python problems. We are using the Sketch [10, 11] solver to encode Python semantics inside a constraint solver. For example, consider the following python function everyOther that appends every alternate element of an input list l1 with every alternate element of another input list l2.

```python
def everyOther(l1,l2):
  x=l1[:2]
  y=l2[:2]
  z = x.append(y)
  return z
```

This python program is then converted into an equivalent Sketch program. The main challenge in this translation is that Sketch is a statically typed language whereas Python is dynamically typed, but we use a strategy similar to the strategy used in Autograder [9] to use union types to encode Python types in Sketch. The translated code looks as follows:

```
MultiType everyOther(MultiType l1,MultiType l2){
  MultiType x = listSlice(l1,0,2);
  MultiType y = listSlice(l2,0,2);
  MultiType z = append(x,y);
  return z;
}
```

We now introduce holes inside the translated program using the hole construct in Sketch. These hole values can take any constant integer values. We then use the Sketch solver to solve the constraints such that there still exists a unique solution to the problem while the

number of holes are maximized. After the end of the algorithm, we expect to get a new python programming problem as:

```python
def everyOther(l1,l2):
  x=l1[:__]
  y=l2[:__]
  z = __.append(y)
  return __
```

In addition to programming problems, we would also like to generate problems in Mathematics (algebra, trigonometry, geometry etc.). We need to define only new domain-specific languages to encode the corresponding semantics of these domains, and then we can plug them into our algorithm to generate new problems.

6 Conclusion

We present the PUZZLEJAR system, based on a constraint-based iterative algorithm, to automatically generate new fill-in-the-blank type problems. We used Sudoku and Fillomino puzzles as case studies, as their descriptions fit naturally with constraint solvers. The PUZZLEJAR system was able to generate hundreds of thousands of both types of puzzles over different sizes. We are currently extending PUZZLEJAR to also create Python programming problems, on which we already have some initial results. We believe a constraint-based approach provides a generic and flexible mechanism for teachers to specify different constraints that they would like a problem to have; we can then use efficient constraint solvers to automatically generate new problems.

References

[1] D. o. M. Cornell University. The math behind sudoku, June 2009.

[2] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] M. Ercsey-Ravasz and Z. Toroczkai. The chaos within sudoku. *CoRR*, abs/1208.0370, 2012.

[4] N. Funabiki, Y. Korenaga, T. Nakanishi, and K. Watanabe. An extension of fill-in-the-blank problem function in java programming learning assistant system. In *Humanitarian Technology Conference (R10-HTC), 2013 IEEE Region 10*, pages 85–90, Aug 2013.

[5] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 50–61, 2011.

[6] N. Jurkovic. Diagnosing and correcting student's misconceptions in an educational computer algebra system. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, ISSAC '01, pages 195–200, New York, NY, USA, 2001. ACM.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] R. Singh, S. Gulwani, and S. K. Rajamani. Automatically generating algebra problems. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[9] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 15–26, 2013.

[10] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6), 2013.

[11] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[12] Wikipedia. Fillomino, http://en.wikipedia.org/wiki/fillomino, 2014.

[13] Wikipedia. Sudoku, http://en.wikipedia.org/wiki/sudoku, 2014.