Automatically Generating Puzzles of Different Complexity

September 29, 2014

Abstract

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. In this paper, we present an iterative constraint-based technique for automatically generating problems. Our technique, which is general in many domains, 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, we 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 the characteristics most beneficial for a student to learn. In this paper, we present a system that solves a simpler task of automatically generating sudoku and fillomino puzzles of different complexity levels. We find our technique and algorithms general enough to generate programming problems as well as problems in other domains, such as algebra and trigonometry.

We present a generic, iterative, and 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. Our approach, on the other hand, lets one specify the puzzle definition and puzzle complexity in a declarative fashion, using constraints and then efficient constraint-solving to incrementally satisfy restraints generated from different iterations. The algorithm first generates a completely random problem that satisfies the constraints. It then removes elements from the complete problem in a user-defined probabilistic fashion. We use the z3 SMT solver [6] and its theory of linear arithmetic for representing and solving the constraints.

We have successfully used our system to automatically generate more than 200,000 9x9 sudoku puzzles and more than 10,000 fillomino puzzles. 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 our system 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 is still an open research problem, we resort to machine learning techniques to learn a function over the sudoku features from a set of labelled sudoku problems obtained from popular sudoku websites and newspapers. We then use this 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, math problems, and other puzzles.

2 Overview of our Framework

We present an overview of our general framework to synthesize puzzles of varying complexity levels. Our framework takes three components as input: a declarative definition of the puzzle D, a complexity function C, and a set of transformation functions $\tilde{T}$. We first define these components and then present our synthesis algorithm to automatically generate puzzles of varying complexity.

Definition 1. *A two dimensional puzzle board of size $n \times m$ is defined using a valuation function $\mathcal{P} : \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{D}$, 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. *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. *The complexity function $C : \mathcal{P} \rightarrow H$ takes a puzzle board $\mathcal{P}$ as input and maps it to a finite class of hardness levels denoted by H.*

Definition 4. *A transformation function $T : \mathcal{P} \rightarrow \mathcal{P}$ 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 $\tilde{T}$.*

The algorithm first uses the `GetRandomPuzzle` function to get 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 |`SquaresTested`| becomes equal to the size of the puzzle

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

```
 1: $\mathcal{P}_I :=$ GetRandomPuzzle$(D)$
 2: $\mathcal{P}_C := \mathcal{P}_I$
 3: SquaresTested $:= \emptyset$
 4: complDict $:= \{\}$
 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 $T \in \tilde{T}$ 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:             complDict$[h] :=$ complDict$[h] \cup \mathcal{P}$
17:         end for
18:         $\mathcal{P}_C := \mathcal{P}'_C$
19:     end if
20: end while
21: return complDict
```

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 our framework 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 set to ϕ (denoting an empty square). Often times, we can apply puzzle constraints-preserving transformations to the puzzle boards to get new puzzle board configurations. The algorithm applies the set of transformations $\tilde{T}$ 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 `complDict`. This `complDict` is the resulting dictionary that is returned by the `GenPuzzles` algorithm.

In general, it is hard 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. In our framework, we try to approximate this complexity function using machine learning techniques. We obtain a set of labelled tranining 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/news papers, but we plan to get such labelled training data from human subjects in near future. For a puzzle, we define a feature vector consisting of a set of features that are specific to the puzzle which 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.

Our framework allows for general `isRemoveValid` functions 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 solution can not be the original solution $S \neq S'$. For a value k, we get the constraint $S' \neq S_1 \vee S' \neq S_2 \cdots S' \neq S_k$. This strategy needs $k + 1$ solver calls to check whether a square can be emptied from the puzzle baord, which can make the overall algorithm quite expenesive. 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. there should not exist a solution S' different from the original puzzle board.

3 Case Studies

3.1 Sudoku

3.1.1 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 constraints sudoku constraint is obtained by combining all previous constraints:

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

3.1.2 Creating the Initial Puzzle

To accomplish this, we use a set of equations using the python z3 constraint solver. This would generate a sudoku board in a 2 dimensional list such as the one below.

```
[[4, 9, 7, 1, 8, 2, 5, 3, 6]
 [1, 5, 2, 3, 6, 4, 8, 9, 7]
 [8, 6, 3, 5, 7, 9, 4, 1, 2]
 [7, 3, 4, 6, 9, 1, 2, 5, 8]
 [2, 8, 9, 4, 3, 5, 7, 6, 1]
 [5, 1, 6, 7, 2, 8, 9, 4, 3]
 [3, 2, 5, 9, 1, 7, 6, 8, 4]
 [9, 7, 1, 8, 4, 6, 3, 2, 5]
 [6, 4, 8, 2, 5, 3, 1, 7, 9]]
```

3.1.3 Emptying Squares

The next step is to start removing values from the board. 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 calculate a new decimal between 0 and 1. Once we have a number between 0 and 1 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. The final selected square will be at the intersection of the selected row and column, at (i, j). The complete process allows for rows with more un-emptied cells to have a greater chance 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 emptied reaches 81.

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

3.1.4 Transformations

We are able to quickly generate more full sudoku boards for emptying without using SAT solvers. 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. [source] 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.

3.1.5 Defining Complexity

After a sudoku puzzle is generated, we determine its difficulty using machine learning. For each puzzle, 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 then uses the vector to categorize the puzzle into one of four difficulties: (1) Easy, (2) Medium, (3) Hard, and (4) Evil.

3.2 Fillomino

3.2.1 Declarative Definition

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

```
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 graphs, with each vertex being a cell on a board and each edge being some type of 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)
```

If there is an edge between two cells, those two cells are in the same regions; therefore, they should have the same value.

```
same_value_constraint = []
for i in range(N):
    for j in range(N):
        for (k,l) in getAdjacent1(i,j, N):
            if lessThan(i,j,k,l):
               same_value_constraint.append(Implies(Or(
               edge_var[(i,j,k,l)]==1,  edge_var[(k,l,i,j)]==1),
               cells[i][j] == cells[k][l]))
```

Next, we create a new variable for each cell, `in_cell`, that is equal to the number of directed edges from neighboring cells going into the cell.

```
in_cell = [[Int("in%d%d"%(i,j)) for i in range(N)] for j in
   range(N)]
in_cell_constraints=[]
for i in range(N):
   for j in range(N):
      in_cell_constraints.append(And(in_cell[i][j]>=0,
      in_cell[i][j]<=1))
      sum_incoming_edges = Sum([edge_var[(k,l,i,j)] for (k,l) in
      getAdjacent1(i,j, N)])
      in_cell_constraints.append(in_cell[i][j]==sum_incoming_edges)
```

Finally, we define a cell size as being the sum of the sizes of the adjacent cells + 1 as the constraint solver iterates over all cells. In each region, we add the constraint such that there is exactly one cell with a size equal to its value.

```
size_region_constraint=[]
size_cell = [[ Int("size%d%d" % (i,j)) for i in range(N)] for j in
                            range(N)]
    size_region_constraint=[And(size_cell[i][j] >=1,
    size_cell[i][j]<=N) for i in range(N) for j in range(N)]

    for i in range(N):
        for j in range(N):
            size_adjacent = Sum([If(edge_var[(i,j,k,l)]==1,
            size_cell[k][l],0) for (k,l) in getAdjacent1(i,j, N)])
            size_region_constraint.append(size_cell[i][j] ==
            (size_adjacent+1))
            size_region_constraint.append(Implies(in_cell[i][j]==0,
            size_cell[i][j] == cells[i][j]))
```

3.2.2 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. Our second option is to create a board using python only. The python code 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 that all have the value sequence length. We continue like this until the board is completely filled.

3.2.3 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.

3.2.4 Transformations

Because of the randomness of the different regions, 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.

3.2.5 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.

4 Experiments

4.1 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

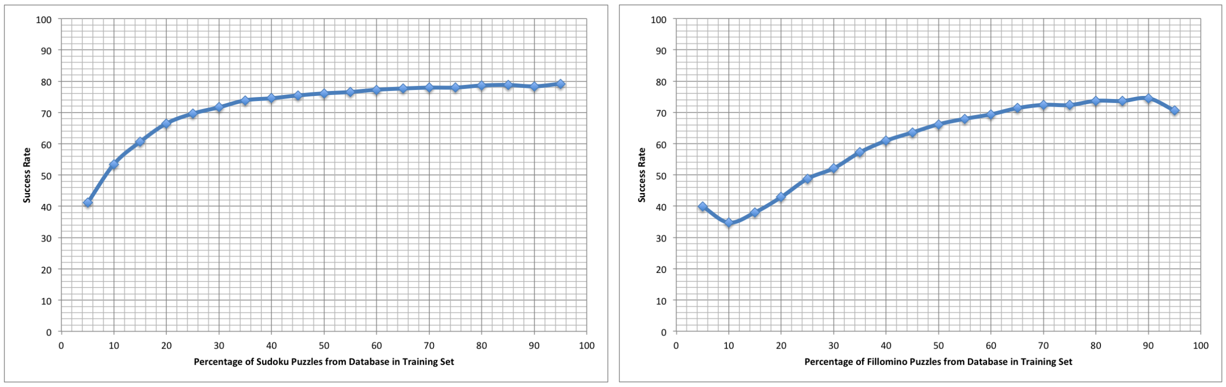


Figure 1: Success Rates (a) Sudoku (b) Fillomino

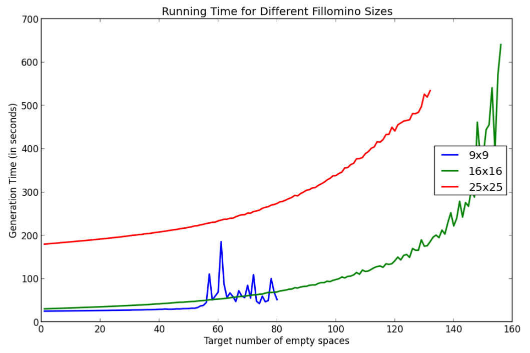
puzzles and their respective difficulty levels from online puzzle providers. This database of puzzles was randomly divided into two sets: a training set and a testing set. The training set was used to generate the SVM's categorization function, and the testing set was used to generate the "success rate": 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. Below are graphs of the average success rates of 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%.

4.2 Scalability of Our Approach

To test the scalability of our puzzle generation algorithm, we generated 16x16 and 25x25 sudoku boards to compare with the standard 9x9 boards and all square fillomino boards from 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



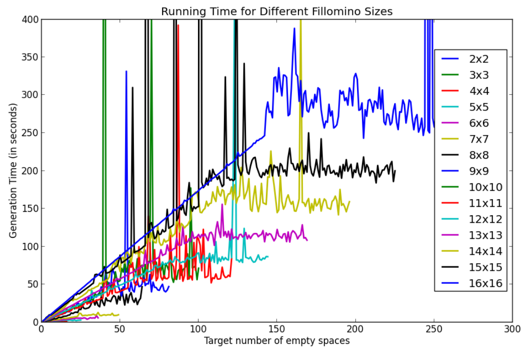


Figure 2: Running Times by Target Number of Empty Spaces (a) Sudoku (b) Fillomino

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, we observe stagnation in our run times as the target number of empty cells is increased beyond a certain threshold.

The graph below 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 (around 500 seconds).

We can see in the graph that the generation time for 9x9 puzzles does 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.

5 Related Work

With the advent of recent online education initiatives, we are seeing an ever increasing number of students enrolling in online classrooms. Some of the popular courses, such as Introduction to Programming and Introduction to Machine Learning, routinely report over 100,000 student enrollment. This large scale of students has forced us to develop new automated technologies to solve problems such as automated feedback generation [4] 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.

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 [2]. 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 proof, which is then instantiated with different concrete values [3]. The system first tries to learn a general query from a given proof problem, and then it executes to generate new 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 a 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 solution [1]. The technique blanks variables using the condition that at least one occurrence of each variable remains in the scope, and it blanks control symbols such that at least one occurrence 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 and an arbitrary complexity function.

6 Future Work

We are currently working on extending our algorithm 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 only need to instantiate these components for generating Python problems. We are using the Sketch[cite sketch] 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.

```
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 [4] that uses union types to encode Python types in Sketch. The translated code looks like this.

```
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:

```
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 only need to define 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.

7 Conclusion

We present 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. Our algorithm was able to generate hundreds of thousands of both types of puzzles over different sizes. We are currently extending our system 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.