Automatically Generating Puzzles of Different Complexity

September 27, 2014

1 Introduction

Students learn by practicing over lots of problems, but generating fresh problems that involve using the same set of concepts and are of a given difficulty level requires a lot of work for a teacher. Our main goal is to automatically generate programming problems that are of different complexity and that are parameterized by the set of concepts a student might want to learn. In this work, we present a system that solves a simpler task of automatically generating sudoku puzzles of different complexity levels, but we have found our technique and algorithms general enough to generate programming problems as well other domains such as algebra and trigonometry problems.

We present a generic iterative constraint-based algorithm to generate 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 uses efficient constraint-solving to incrementally solve constraints generated from different iterations. The algorithm first starts with a complete random problem that satisfies the constraints that it is well-formed and covers the desired set of concepts. It then starts removing elements from the complete problem in a user-defined probabilistic fashion such that after each removal some problem-specific properties (such as bounded number of solutions, uniform distribution of spaces etc.) are satisfied. 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 varying 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 16X16 and 25X25 sudoku problems. Since computing the hardness of a sudoku problem 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 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 as well as 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 $\mathcal{P} : \mathbb{N} \times \mathbb{N} \to \mathcal{D}$, 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. *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 $C : \mathcal{P} \to H$ takes a puzzle board $\mathcal{P}$ as input and maps it to a finite class of hardness levels denoted by H.*

Definition 4. *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 $\tilde{T}$.*

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

$\mathcal{P}_I :=$ `GetRandomPuzzle`(D)
$\mathcal{P}_C := \mathcal{P}_I$
`SquaresTested` $:= \emptyset$
`complDict` $:= \{\}$
while $|$`SquaresTested`$| \neq |\mathcal{P}_I|$ **do**
  $(i, j) =$ `ChooseSquare`$(\mathcal{P}_C)$
  `SquaresTested` $=$ `SquaresTested` $\cup (i, j)$
  if `isRemoveValid`$(\mathcal{P}_C, i, j)$ **then**

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

    $R := \emptyset$
    for $T \in \tilde{T}$ **do**
      $R := R \cup T(\mathcal{P}'_C)$
    end for
    for $\mathcal{P} \in R$ **do**
      $h := C(\mathcal{P})$
      `complDict`$[h] :=$ `complDict`$[h] \cup \mathcal{P}$
    end for
    $\mathcal{P}_C := \mathcal{P}'_C$
  end if
end while
return `complDict`

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 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 CaseStudies

3.1 Sudoku

3.1.1 Declarative Definition

We use the python frontend of z3 constraint solver 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, i.e. $1 \leq$ 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 sudoku constraints that the values in each row should be distinct (rows_distinct), values in each column should be distinct (cols_distinct), and that each 3X3 square should have distinct values (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 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 emptying this board. Our method for selecting the next square to remove includes a number of substeps. First, we calculate the percentage of squares filled in each row (number of squares in row that are full/9). We then randomly select one of these percentages and generate a random number between 0 and 1. If this number is greater than the percentage, we find a new percentage and 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 in the row we should empty. We set the square in the selected row and column equal to zero, signifying emptiness.

After we have found a square to empty, we generate a temporary board with the selected square having a value of 0. We also a new set of constraints to go along with this board using z3. If z3 can solve the new set of equations in less than K ways, we have a desired result and we keep this board. If z3 can generate a number of solutions that is greater than or equal to K, the chosen square is added to a list of squares that do not work and should not be tried again. This temporary board is discarded and we find another square using the board we had before the temporary board was generated. Once we find a square that, when removed, results in a desired board, we find another square to empty. If no other squares work, meaning that the number of squares emptied plus the number of squares in the list of squares that do not work equals 81, we stop looking for another square to empty. If the algorithm does not stop in the first case, it stops once we reach the number of squares we wish to empty, although this second case is less likely if the desired number of empty squares is greater than 60.

3.1.4 Defining Complexity

After a Sudoku puzzle is generated, we determine its difficulty using machine learning. For each puzzle, a vector is generated with components describing the unsolved board. These characteristics are:

- Number of solutions
- Number of empty squares
- Number of rows with at least seven blank squares
- Number of columns with at least seven blank squares
- Number of 3x3 grids with at least seven blank squares
- Number of occurrences of each digit (9 components)
- 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.

3.1.5 Transformations

We are able to quickly generate more full Sudoku boards for emptying without using SAT solvers. To do this, we apply the following mathematically symmetrical transformations on an already-created 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

We chose a random transformation and applied it to an already existing board 1000 times to create a new Sudoku board satisfying the original constraints. Out of about 6 x 1021 total unique 9x9 Sudoku boards, these transformations can generate about 3 x 106 new unique Sudoku boards from an existing one.

We can apply most or all of these transformations to already-created 12x12, 15x15, 16x16, and 25x25 boards. On a 12x12 board, for instance, the following transformations are analogous:

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

This method of quickly generating new boards generalizes to any higher-numbered Sudoku board.

The last step of our algorithm is quickly generating more sudoku puzzles from the board that was emptied. This task is accomplished by performing transformation on the existing board as described above.

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 that, for each region on the board, the value of all of the squares in a specified region must be the same as the number of squares in that region.

To do this, we first create a variable edgeval that can either be one, representing that an outgoing edge to an adjacent square exists or 0, showing that the edge does not exist. An outgoing side should only exist between two cells that are in the same region. The variable for an edge is edgevar(i,j,k,l), where (i,j) is the location of the original square and (k,l) is the location of the adjacent square.

```
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()]
```

For our method to work, we only want one outgoing edge from every square, meaning that the value of edgevar will always be ¡= 0.

```
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)
```

Now we create a new variable for each cell, incell, that is equal to the number of edges 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
```

Next we say that if there is an edge between two cells, those two cells are in the same regions, and 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
                    edge_var[(k,l,i,j)]==1), cells[i][j] ==
                    cells[k][l]))
```

Finally, we define a cells size as being the sum of the sizes of the adjacent cells + 1. We want there to be exactly one cell whose size is the same as 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 option to create the initial puzzle. We can run the declarative definition of a Fillomino puzzle, but this code is slow and will only get as puzzle because z3 does not come up with a different solution each time the code is run, it is the same one every time. Our second option is to randomly create a completed Fillomino puzzle using python without any z3. 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 Choosing a Valid Square

Choosing a Valid Square Our first step is define what we mean by a valid square. For most cases, we want to find a square that, when removed, results in a board that has exactly one solution, but in general, we want it to have less than K solutions. Our first step is to randomly choose a square on the board, with the only restriction being that every region must have at least one cell that is not emptied, as in the rules of Fillomino.

In Sudoku, a square in a row, column, or three by three grid that had less emptied squares in it already was more likely to be chosen. This is not the case in Fillomino because the number of squares in a row or column do not matter, the only constraint is that there must be at least one value in every region that is not emptied.

As in Sudoku, we create a temporary board that has the selected square removed (the square has a value of 0). We then apply the z3 declarative definition on this temporary board, and if z3 can find K or more ways to solve the puzzle, we know that we should not remove the square. We resort back to the old board, add the square that we tried to a list of squares that we know do not work, and try again with a new random square. This process continues until we have a desired number of squares emptied.

3.2.4 Defining Complexity

Using machine learning, we determine the difficulty of an unsolved fillomino puzzle based on the following characteristics:

- Number of cells
- Number of empty squares
- Number of regions (connected areas of the same value)
- Optimality (whether the puzzle can be emptied further)

3.2.5 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
3. Horizontal reflection

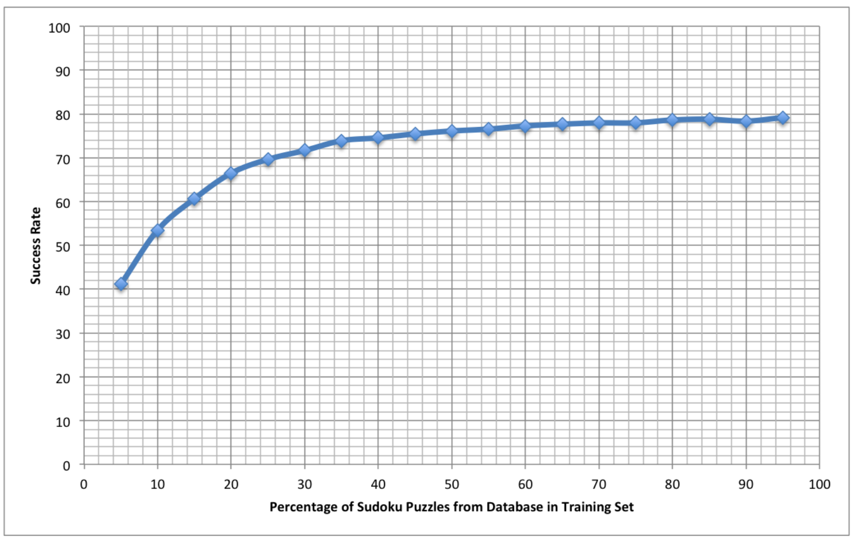
Applying these transformation to the original z3 declarative definition puzzles as well as the randomly generating python generated puzzles result in a fast way to create many more boards. The transformation can also be applied to an emptied board to get more solutions.

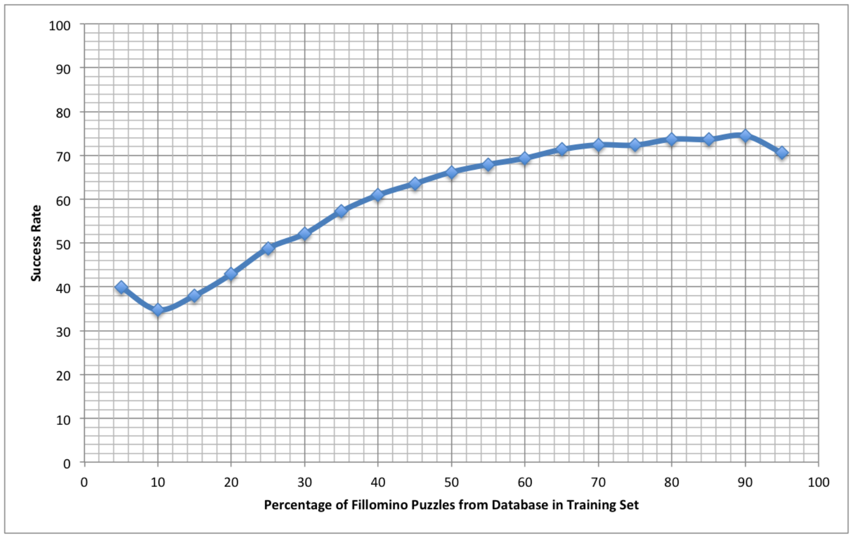
4 Experiments

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 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%.

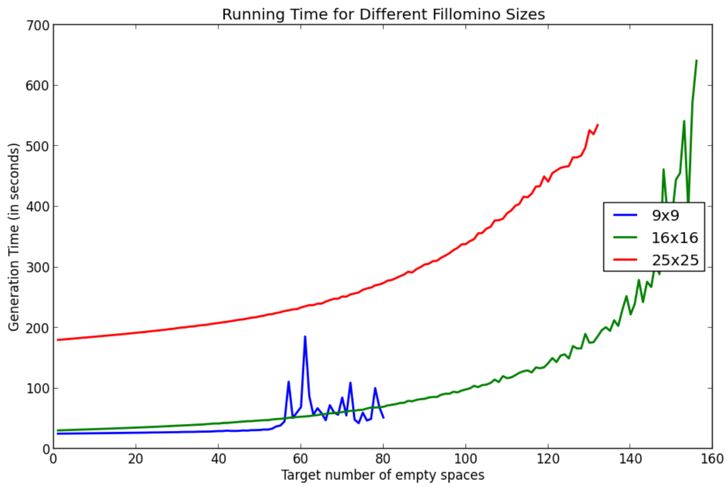
4.2 Scalability of Our Approach

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 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.

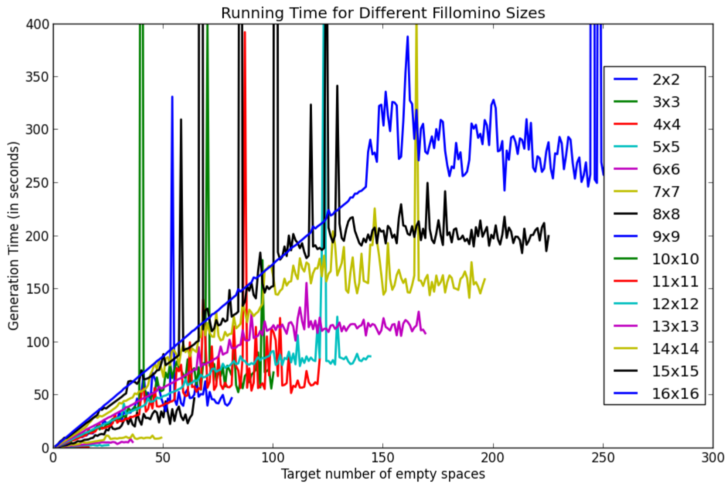
The graph below shows program running times to create and empty 9x9, 16x16, and 25x25 Sudoku puzzles as the target number of empty squares is increased. There is an exponential increase in run time as the number of possible empty spaces increased. Even for 25x25 sudoku puzzles, we find that the time 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 reports more than 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 that comes up because of 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 problem 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 problem which is then instantiated with different concrete values [3]. 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 solution [1]. 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 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 that it takes as an additional parameter.

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 just 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] to use 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.) as well. 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.