Automatically Generating Puzzles of Different Complexity

September 25, 2014

1 Introduction

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 different puzzles.

Definition 1. *A two dimensional puzzle board of size $n \times m$ is defined using a valuation function $\mathcal{P} : \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{D}$ that assigns values to the squares on the puzzle board. The value of the square (i, j) 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 all 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 $\tilde{T}$.*

Algorithm 2 The `GenPuzzles` algorithm for synthesizing puzzles of varying complexity levles.

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

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

9: **for** $T \in \tilde{T}$ **do**
10: $R := R \cup T(\mathcal{P}_C)$
11: **end for**
12: `complDict` $:= \{\}$
13: **for** $\mathcal{P} \in R$ **do**
14: $h := C(\mathcal{P})$
15: `complDict`$[h]$:= `complDict`$[h] \cup \mathcal{P}$
16: **end for**
17: **end if**
18: **end while**
19: **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. Note that sometimes for this first step, we also use custom random puzzle generators for scalability. It then starts removing the value at a square (i,j) until there are no more square values remaining to be removed. The algorithm uses the isValidSquare function to check if certain puzzle constraints hold after removing the square value (i,j). An example isRemoveOK function is that the Number of solutions to the puzzle after removing the square value is only 1. Let P_c denote the puzzle obtained after removing a valid square. We then apply a set of transformation functions T_P to get a set of new puzzles.

The difficulty of a generated puzzle is determined by characteristics of the unsolved board.

3 CaseStudies

3.1 Sudoku

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.

3.2 Fillomino

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

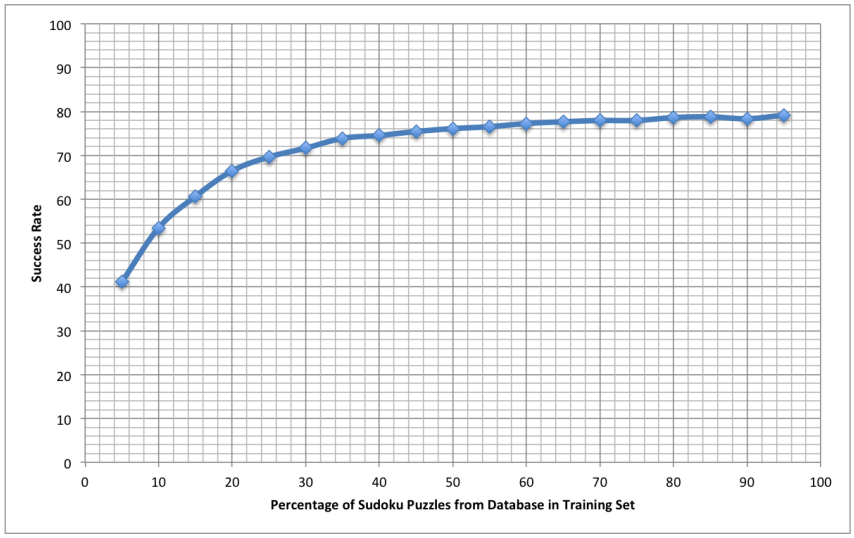
- Number of cells
- Number of empty squares
- Number of regions (connected areas of the same value)
- Optimality (whether the puzzle can be emptied further)

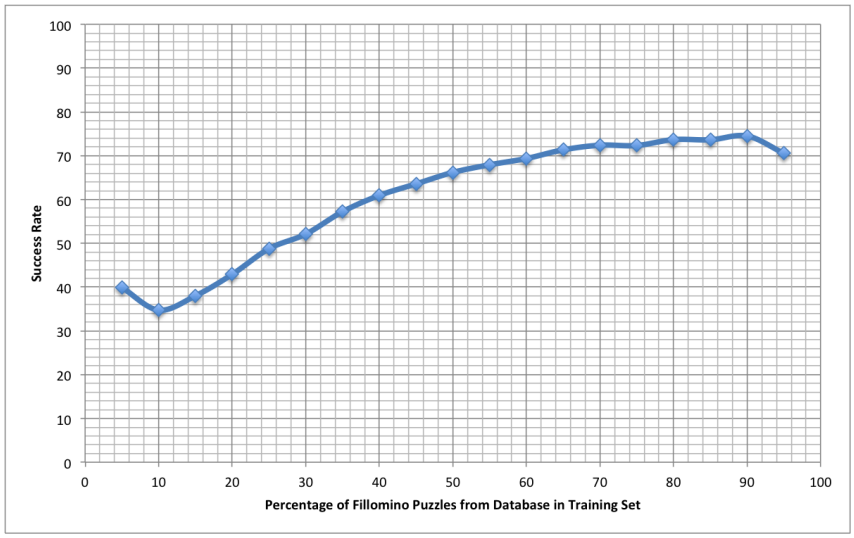
4 Experiments

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.

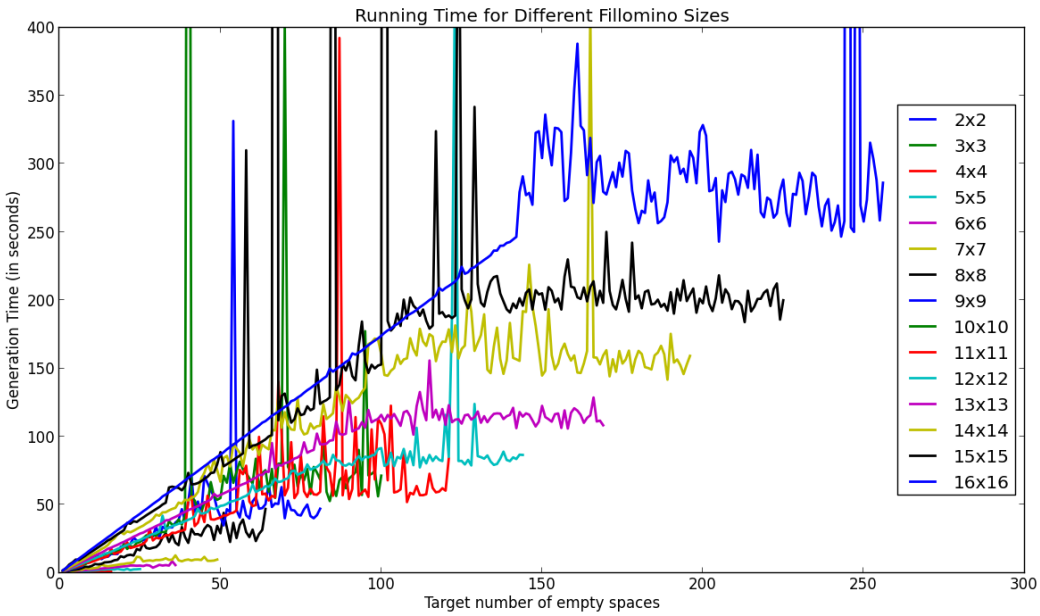
When we empty a puzzle with dimensions NxN, we set as a parameter the target number of cells that the program would aim to empty from an initially full board. When the program reaches a point where it cannot further empty any more cells (i.e. the emptying of any remaining cell would cause the board to 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).

[insert Sudoku Running Time Graph here]

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 val-

ues 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 Conclusion