Experimental Mathematics in Word Puzzles

Yonah Biers-Ariel, Matt Hohertz, Jingze Li, and Lauren Squillace

May 13, 2019

We wrote code to randomly generate complete crossword puzzles; i.e. $m \times n$ matrices where the entries in each row and column form a word. In this writeup, we describe our algorithms to generate these puzzles.

Let VOC1 be a set of words of length n, VOC2 be a set of words of length m, and let `Trunc`(VOC,k) be the set of length$-k$ prefixes of words in VOC. Let `RS`(VOC) denote a random sample from VOC. Our most basic algorithm inputs VOC1 and VOC2 (across clues will all come from VOC1 and down clues will all come from VOC2) as well as a constant GIVEUP which measures how many times the algorithm will try to add a word into a partial grid before giving and starting the grid over. It works as shown in Algorithm 1.

Algorithm 1: `GenPuzzle1`

```
output:=[]
count:=0
while Length (output) < m do
    v := RS (VOC1)
    k:=Length (output)
    viable:=true for i = 1 : n do
        if [output[1][i] ... output[k][i]v[i]] ∉ Trunc(VOC2, k + 1) then
            viable:=false
    if viable then
        Append v to output
    count++
    if count = GIVEUP then
        output:=[]
        count:=0:
return (output)
```

As written, `GenPuzzle1` looks at random potential across entries, determines whether all the partial down words that they form are valid prefixes, and, if so, adds that across entry to the puzzle. A better algorithm, though, might be one that is more likely to choose across entries when their partial down words are prefixes for many words, and would be less likely to choose across entries when their partial down words are prefixes for only a few words.

Thus, we will randomly sample many potential across entries, and assign to each one a score based on how many words their partial down words are prefixes for. The probability that we choose any potential across entry is then proportional to this score. This idea is implemented in Algorithm 2. Note that `TruncN`(VOC,pref) is the number of words in VOC with prefix pref.

As it turns out, Algorithm 2 is significantly faster than Algorithm 1, and we can use it to generate grids of size up to 5x6.

In addition to the changes seen in Algorithm 2, we also tried to change the restart condition. As it is, both algorithms restart completely with a blank grid once they have tried going through some number of putative across entries without forming a complete grid. One could imagine, though, an algorithm in which we instead simply deleted the most recent across entry after some number of failures, but kept the ones before it. When we implemented this change, though, we found that it ran significantly slower than either Algorithms 1 or 2.

Finally, we add the warning that, while these algorithms do terminate with probability 1 as long as a viable grid with the necessary dimensions exists, both algorithms can run for arbitrarily long periods of time.

Algorithm 2: `GenPuzzle2`

```
output:=[]
count:=0
while Length (output) < m do
    Vs := [RS (VOC1),..., RS (VOC1)]
    k:=Length (output)
    scores:=[1,...,1]
    for j = 1 : Length(Vs) do
        v:=Vs[j]
        for i = 1 : n do
            if [output[1][i] ... output[k][i]v[i]] ∉ Trunc(VOC2, k + 1) then
                score[j] = 0
            else
                score[j]× = TruncN(VOC2, [output[1][i] ... output[k][i]v[i]])
    if scores ≠ [0, 0, ..., 0] then
        Choose j ∈ {1, 2, ..., Length(scores)} such that each entry is
        chosen with probability scores[i]/ ∑_j scores[j]
        Append Vs[j] to output
    count++
    if count = GIVEUP then
        output:=[]
        count:=0:
return (output)
```