AI - 28 Oct 2020

Judicious brute force

There is a class of puzzles where each position of the puzzle is to be filled in with a symbol. This could be as simple as a 0/1 symbol (such as in the n-queens problem where a 0 means no queen at that position and a 1 means there is a queen at that position), or numbers (such as 1-9 in most Sudoku problems), or arbitrary labels (such as 0-9 and A-F in 16 x 16 sudokus). Often, as with a standard Sudoku puzzle, the numbers are acting as only labels, meaning that the numeric qualities of the number are not being used. In those cases, letters or other symbols could just as easily be used.

A possible (horrible) approach to solving this type of puzzle might be

```
def bruteForce(pzl):
  # returns a solved pzl or the empty string on failure
  if pzl is completely filled out:
    return "" if isInvalid(pzl) else pzl

  find a setOfChoices that is collectively exhaustive
  for each possibleChoice in the setOfChoices:
    subPzl = pzl with possibleChoice applied
    bF = bruteForce(subPzl)
    if bF: return bF
  return ""
```

The reason this is so horrible is that it fills each puzzle out completely before testing. For all but the most trivial of problems, the routine will hang because there are way too many possibilities to recurse through. However, we can gain considerable traction with only a slight modification. By testing the validity of a potential solution at each step of the process, it is the hope that large subtrees of invalid possibilities may be immediately pruned. For example, in the n-queens problem, if two queens attacked each other, there is no reason to continue filling in the puzzle since it's already unsolvable. The following is what was shown in class as our starting out point for the month:

```
def bruteForce(pzl):
  # returns a solved pzl or the empty string on failure
  if isInvalid(pzl): return ""
  if isSolved(pzl):  return pzl

  find a setOfChoices that is collectively exhaustive
  for each possibleChoice in the setOfChoices:
    subPzl = pzl with possibleChoice applied
    bF = bruteForce(subPzl)
    if bF: return bF
  return ""
```

Your homework is to use the above *judicious* `bruteForce` code (or small variation on it) to write the following code:

Sudoku basics

A standard Sudoku puzzle is a square $N \times N$ grid where $N = n^2$ and n is typically 3. There are N distinct symbols that may be affixed to empty grid positions so that each grid position gets exactly one symbol and no row nor any column has any duplicate symbols. In addition, none of the N sub-blocks of size $n \times n$ may have any duplicate symbols either.

This may be generalized so that if a Sudoku puzzle of size $N \times N$ is given, then the N subblocks will be of size $h \times w$ where h is the largest integer not greater than $\sqrt{N}$ that evenly divides N, and then $w = N/h$. None of the N rows, N columns, and N subblocks may have any duplicate symbol.

There are several basic variables and structures to set up:

1) Accept a file name from the command line, and read in a list of sudoku puzzles from this file.

1A) If instead of a file name, a sudoku puzzle is given, print out the Sudoku puzzle in 2D (and its solution, too). You could print this out like a large slider puzzle, but in the case of Sudoku the subblocks matter, so it is probably best to also have some kind of separator to indicate the subblocks. Simple ways to do this are to use the pipe and hyphen, or perhaps simply a space. How will you know whether it's a file name or a sudoku puzzle? Make reasonable assumptions.

2A) Loop through each puzzle. Per puzzle, call `setGlobals()` (or `setLookupTables()`). `setGlobals()` should first determine the size, `N`, of the puzzle and the dimensions of the subblocks. Puzzles will always be square (eg. of length 81, 100, 144, 256) where the side length of the puzzle, `N`, will be a composite integer no larger than 25. From your slider puzzle days, you already have code to determine the subblock size. (eg. If the puzzle length is 144, the puzzle size is 12, and the subblock size is 3 high by 4 wide).

2B) `setGlobals()` should also determine the symbol set, `SYMSET`, of the puzzle. Not all puzzles include all N of their symbols. In those cases, your code should supply the missing symbol. The following conventions usually apply for the indicated N:

9:	{1-9}
12:	{1-9} ∪ {A-C}
16:	{0-9} ∪ {A-F} OR {1-9} ∪ {A-G}

2C) Determine the N constraint sets for rows, the N constraint sets for columns, and the N constraint sets for subblocks. A constraint set is a set of index positions such that no two squares at any of the index positions may have the same tile. Put each of the constraint sets into a global list (ie. a list of constraint sets). The constraint sets for the columns and rows will be straightforward, but for the subblocks it is a bit trickier, so make certain that you have them correct. A puzzle size of `N=6` may be useful for testing.

3) Write an `isInvalid(pzl)` function utilizing the list of constraint sets from the prior part

4) On each iteration through the main loop, feed the sudoku puzzle to `bruteForce()` and then display the results. Identify the puzzle number (starting from 1), the solved puzzle a check sum (see below), and the time it took to solve it.

5) Write a simple `checkSum(pzl)` function: sum the ascii value of each symbol in your puzzle and subtract $48N^2$. Until your code is rock solid, do this as a simple way of double checking your Sudoku solution correctness – output this number next to your Sudoku solutions.

Your final output should resemble the following:

```
…
13: 2459813761692735 … 8349165654812793 405 0.203125s
14: 4628319577954261 … 4259671517643892 405 0.53125s
15: 1372568499283145 … 1428675784965123 405 0.0625s
16: 5238167497845931 … 5624378876351492 405 0.015625s
17: 1769235845248176 … 1236945349571826 405 7.15625s
18: 1439862576794253 … 6143825534892716 405 0.671875s
…
```