COMP 560: KenKen Solver

Cynthia Dong 27 September 2019

Setup

From the input file, four objects are created:

1. A 2D array "board", which contains the letters at each cell in the puzzle.
2. A dictionary of "pieces", where the key is the letter that defines the piece. Each key is tied to a custom object "Piece" which contains booleans for whether it's filled, solved, its letter, the coordinates that make up the piece, and (for backtracking) the number of empty cells in the piece.
3. A dictionary of "rules", where each rule is keyed by the letter and the value is a tuple of the format (number, operation).
4. A "solution":
    a. For the two backtracking searches, this is a 2D array initialized to 0, to be filled with the solution to the problem.
    b. For the local searches, this is a custom object with a "grid" that is a randomly filled 2D array, a list of pieces, a "fitness" value, and a "probability" for the genetic algorithm.

Simple Backtracking Search: kk0.py

This backtracking search begins with a "solution" filled with 0's. The first node tackled is at the index (0, 0). A list of values is generated for each node with the function "findRemainingValues", which takes a list of numbers from 1 to the length of the board and gets rid of any numbers that would cause a repeat in the row or column of the index.

The backtracking search tries the first value in this "valuesToTry" list. If filling the node completes a piece, the search checks if the math works out with the function "mathCheck", which returns 1 if it passes and 0 if it doesn't. The search then moves on to the next node in the row recursively, calling "findRemaningValues" again to regenerate the non-constrained values for the current node. If there aren't any values in "valuesToTry", the search will return to the previous node, trying the next value in "valuesToTry" for that node. This forward-checking continues for the entire board until the board is correct.

This solution will always give you an answer, but for large boards, it may take an untenable amount of time.

"Best" Backtracking Search: kk1.py

My improved backtracking search *improves*, eponymously, on the simple backtracking search by increasing the constraints placed on the values the search tries for each node, resulting in a more pruned search tree. The "findRemainingValues" function remains the same, checking for repeats in rows and columns and returning values that aren't yet in the row or column. However, I order this list with the "findMRV" function, which tests each value in the next node and calls "findRemainingValues" on that node.

The values are then ordered from the value that returns the least remaining values to the value that returns the most remaining values. This means that the search will check nodes that have the least possible "next" values first, which cuts down on the tree size – if the next value is correct, then it saves having to check all of the other children of the other values. If the next value is incorrect, wrong values with fewer children are cut off of the tree first.

I also altered the "mathCheck" function by placing stricter restrictions on the values allowed for each piece. Each time a node is filled, the search checks the piece, even if it isn't completely filled. If the piece is, in fact, completely filled, the search does the standard check to see if the cells in the piece arithmetically satisfy the rule for the piece. If the piece isn't filled, I do some calculations for addition and multiplication to prune the tree a little further.

For addition, I mimic filling the rest of the cells in the piece with the maximum values possible, then add them up. If this doesn't add up to greater than the total, the search knows that the values it's already filled the piece with are too low to ever add up to the right answer, and it returns false for "mathCheck." Likewise, if the current total is already greater than the rule, the search returns false.

The checks for multiplication are very similar to the checks for addition, but instead of adding up the maximum possible values, the search mimics filling the empty cells in the piece and multiplying them together. As with addition, if the maximum value given the current piece-filling is too low, or if the total is already too high, "mathCheck" returns false.

Local Search: kk2.localsearch.py

For the local search, a pre-made solution, generated randomly (but without repeats in each row) is run through the "localSearch" function. The search keeps track of the solution with the best "fitness" (more on that in a second). A list of violations — rows and columns with repeats, and pieces whose totals don't quite add up — is generated, and the search pops off a

random violation. If the violation is a row or column, the search "swaps" the row or column with another randomly generated row or column. If the violation is a piece, the search swaps a value within the piece with a value in the same row. The search keeps the solution with the best fitness.

The number of constraints is calculated as the fitness of the solution with a utility function. The utility function is the sum of the number of repeats in the columns, the number of repeats in the rows, and the difference between the rule value and the actual value for each piece: Utility = Repeats in Columns + Repeats in Rows + Abs(total - rule). The node chosen to be changed each iteration is whichever random violation gets popped off of the violation list.

To prevent the search from getting stuck on any local hilltops (the opposite of Sisyphus, I imagine), the search will shuffle the values of the best solution if the best solution doesn't change for 50 iterations. I also tried to use simulated annealing to cut down on local maxima, but that proved ineffective, so I took it out.

Full disclaimer, my implementation of the local search is not very effective, nor is my implementation of the next search, the genetic algorithm.

Genetic Algorithm: kk2.ga.py

The genetic algorithm tries to get around the local maxima problem of the min-conflicts local search by introducing reproduction and mutation. I was drawn like a moth to the flame by the novelty of treating a computer science assignment like my own personal farm, but I soon discovered that genetic algorithms can be quite complicated.

The search begins by generating a population of "chromosomes", or potential solutions with the assistance of a simple backtracking search that only checks for repeats. The algorithm then iterates over the population, calling "reproduce" in order to generate a new population. 40% of the time, a child solution will be "adjustingSwap"-ed, meaning that its values will be shuffled around until there are no repeats or until adjustingSwap reaches 500 iterations. "AdjustingSwap" works by swapping two random values in a solution, then swapping the newly-created repeats in the column or row until no repeats remain.

The utility function which I used for this search is a little different: "fitness" is generated by summing up the cells in rows and columns, and pieces that have repeats or are in violation of the rules, respectively. A higher fitness means that the solution is closer to the right answer. The

maximum value for this utility function, before normalization, is the length of the board squared, times 3. This value is normalized to a decimal by dividing by the maximum value.

These fitness values are then used to calculate a probability from 0 to 1 for each "chromosome" in the population. Probability = IndividualFitness / TotalFitness. The probability is used to choose which solutions will "reproduce". (Fortunately, humans don't work like this).

The search sorts the population by fitness and keeps the three "best" solutions to make sure that the search doesn't inadvertently kill off a potentially good solution — balancing randomness with optimization. As an extra measure to avoid local maxima, the search "mutates" the 2nd- and 3rd-best solutions by swapping values within the solutions.

Once the search has reached the correct solution or the maximum number of iterations, the search will end and return its best attempt at a solution.

The genetic algorithm seems to be able to generate the correct solution about 1/10 times for a 4x4 KenKen, but it seems to be absolutely hopeless for anything over that. It is most likely due to my utility function, though this was the best out of all of the ones which I tried. Different probabilities for mutation, reproduction, etc. would probably also improve this search method.