Conor Thomason

Due 9/27/2019 @ Midnight

Homework 1 – KenKen

The backtracking solution (and subsequently the improved version) was implemented using a recursive function iterating through the array. A loop iterates through all of the possible values for each square (eventually), concluding when a safe solution has been reached. This is judged by the constraints safeRow, safeColumn, and kenKenValid. safeRow and safeColumn are relatively straightforward, they go through the array for every element horizontally and vertically, checking for duplicates. If they are found, the constraint is not met, and a new value is tried.

A somewhat unique case was with kenKenValid, a method that relied upon the implementation of the Cage, ConstraintCell and SolutionConstraint classes.

- **Cage** kept the ConstraintCells in easily-checked groups, allowing various methods such as nearlyFilled and filledCage, all of would be essential for checking a kenKenvValid state later.
- **ConstraintCells** were the objects that held the data of the problem; they were assigned a key, value, and cost (the cost was used only for local search). When a value was selected, the key would be found, matched with the corresponding cage, and the cage would produce all of boxes, filled or otherwise. If they were all filled, the kenKenValid check would take place.
- The **SolutionConstraints** were used for the final kenKenValid check; they held the area restrictions, the arithmetic operator, as well as the value to be reached by the final operation. A calculation would be attempted by all of these operations, by taking all the nodes stored in the array by Cages in ConstraintCells, and would be checked against the SolutionConstraints. If it was invalid, the method would backtrack, iterate a value, and continue on until a solution was reached.

It is worth nothing that KenKenValid was *not* used for every iteration; it was only used when a Cage was filled, hence why the implementation of the aforementioned filledCage method to be critical to the success of the solution.

Once a valid solution was checked and reached, it would return true. As a recursive function is used, it would return true all the way up. The method calling it would see it was true, and print the result.

1) How did you implement your best backtracking search?

The implementation of the backtracking search largely relies upon large quantities of coincidental data; I thought that the easiest way to always get a net positive in terms of efficiency was to manipulate the numbers themselves. Therefore, I decided to use a greatest common denominator function.

By applying a GCD function once I found that a KenKen “cage” (the block that dictates what arithmetic operation to apply) was nearly full. The GCD iterated through the list, found the greatest common divisor between all of the values provided that were contained in the list, and filled the last block with that value. If the last value didn’t work, then there was no way any other permutation of a value in that block, nor the other blocks, would fulfill the constraint. Therefore, it was reasonable to assume that the KenKen region wasn’t satisfied, and to backtrack.

This solution wasn't great for the smaller problems; due to the relatively few cases in which a method as this may be applied, smaller problems frequently don't take advantage of this method at all. Indeed, for a 9x9 problem, the Simple Backtrack solution had 28 million calls of the recursive function, whereas the Improved Backtrack solution had only 200k less. The significance increases as the size increases, but not spectacularly.

If I were to improve this problem in the future, it would be worth finding more methods similar in nature to this to apply; functions that don't apply constant constraints, but are situationally relevant, would be ideal.

2) What did you use for a utility function to guide your search?

I used the Hill-Climbing Greedy descent algorithm for my implementation of local search.

The function begins by assigning a cost to all of the cells, based upon how many constraints they fail to meet; if they are unique in a row, unique in a column, or fail to meet the KenKen cage rules. The array is filled with an array of n*k numbers of each number, n being the size of the array and k being the type of number [1,2,3,…,etc]. The best cell relative to the current cell being selected would be chosen by its minimal cost; every cell in the array must have a corresponding neighboring cell due to its square nature, so this is sufficient. These cells would then swap, the function would test to see if the swap increased or decreased the summation of all of the costs. If the new total cost was less, it would maintain, otherwise it would swap the values back and move on.

An issue was realized far too late into this process; it was extremely slow for even moderately significant problems, hence the reason the software will likely stall once it is run. The problem currently has a maximum time limit of 100 seconds; this isn't a mistake, this is deliberate, due to the fact this was the limit set upon the algorithm when it worked for the first time.

The algorithm appears to be slow due to a rather mistaken approach towards how the algorithm works; the current implementation relies upon the idea that you go through the array, element by element, swapping them around as you go until eventually the solution is reached. This is not the intended method of solving the problem. The algorithm states that a single cell is selected, the best cell is found, the values are swapped, you maintain the position of the now-swapped cell, and you check again. You repeat this process until the problem is solved.

If this issue were to be approached again, these factors would be taken into account to dramatically increase the efficiency and speed of the problem; in its current state, the problems are not insignificant.

3) How did you choose which nodes to change for each iteration?

As mentioned on question 2, each iteration swap was determined on a case-by-case basis according to the cost of the current node and the cost of any nodes surround it in an x-y direction. Comparisons would be made of all surrounding nodes, linked to each other in a linked-list/tree style of networking, and the one with the lowest cost would be selected for swapping. The costs were assigned by the assignCosts() function, which would go through each of the constraints established by the previously implemented backtracking solutions; safeRow, safeColumn, and kenKenValid. For each failed constraint, it subtracted 1 from 3, to a minimum of 0, being no constraints are violated.

Statement of individual contribution:

Any and all work submitted for this assignment was the work of Conor Thomason alone; no work submitted came to be as a result of other individuals, whether they be students, faculty, or individuals outside of UNC.