COMP 560: KenKen Puzzle Write-Up

1. For our refined backtracking search what we did was we took the original logic of our simple backtracking search and adjusted it to take advantage of the fact that the "/" and "*" operators only have a certain number of possible values that can satisfy it. What we mean by this is that our simple backtracking search would start at an unsolved node, increment it by 1, check to see if it satisfies the KenKen logic(no duplicate values in the row or column, and that the target solution is satisfied for each cage), and if it fails we increment it again by 1 until we find the solution, or until we have exceeded the size of the puzzle. For our best backtracking search instead of incrementing by 1 we decided to make a list of possible numbers for the "/" and "*" operations and to loop over these possible numbers instead of all the possible numbers that are less than the puzzle size + 1. For example, if a cage has a dividing operation and a total number of "2", then the only possible values that could be in the cage are 1, 2, 3, 4, and 6. Implementing this change took the number of iterations for the example given on the instructions from 1525 to 887.

2. For local search, we would generate a random row number and a random column number and call checkCage. Our checkCage in our local search algorithm, however, was different than the backtracking algorithm. In this method, we would take the operation and use it on all of the nodes in the cage. Then, we would subtract the cage number minus the number we got from using the operation on our nodes. We

did this to find the error between the number we want and the number we calculated. Next, we would call checkCage on the neighboring nodes to see if the error was less than the error we already calculated. If it is less, then we switch the two nodes' solution numbers in the board matrix. So in conclusion, our utility function was the error between the cage number and the number we got from using the operation on each node. Our local search algorithm does not always find a solution because it could easily get stuck at a local maximum and not the global maximum we were trying to get. To keep the runtime small, we restricted the number of iterations to 500. We could, however, easily change this number in our algorithm to get a more accurate solution.

3. If the error calculated in checkCage for the neighbor was lower than the original node, then we would switch the node's values.

Individual Contributions:

We all worked together in person for most of it, however, we had a few contributions that we each individually did.

Everett: Implemented the improved backtracking algorithm and the local search algorithm.

Yosef: Implemented the BoardMaker class.

David: Debugged most of the programs and implemented the Cage class.

Together: Discussed the logic behind all of the algorithms and BoardMaker class.

Also debugged together.