Carlos Vasquez
1111-2307
CAP 4621
Project Intelligence Report

What Will Be Implemented:

Firstly, it has come to my attention (after having solved a couple Numbrix grids) that there is no true “algorithm” to solving a Numbrix problem. It seems as though there is always a chance for there to be a guessing “phase”. This guessing is a mixture of enumerating the potential solutions and ignoring the “blatantly” wrong enumerations. However, this “blatant” incorrectness is difficult to translate into code. I will elaborate on this enumeration later. The purpose of stating this is to show that a “true” algorithm is not possible and a “heuristic” is needed (I cannot consider a brute force or “guessing” method an algorithm although it is technically correct to call it such). The “solver” I will be implementing will contain two phases. The first phase is an algorithmic approach and the second phase is a heuristic approach. (I use the terms algorithm and heuristic loosely). In other words, I will be implementing a constraint satisfaction search.

Pre-Solving:
First, the solver will need to analyze the grid and create the data it will use in the next two phases. The solver will keep track of the numbers currently in the grid and attempt to build a/multiple snake(s) from the numbers. What I mean by snakes is a trace of numbers that are horizontally or vertically aligned in ascending/descending order. Hence, we have two data sets:

- Snake(s)
- Existing numbers

The end goal will be to make a single snake. For example, take the following grid:

2	3		5
1			6
			7

We can make two snakes from the above grid: {1, 2, 3} and {5, 6, 7}.

Phase 1- Apply the Constraint:
In phase 1, the goal will be to solve for the next head or tail of the snake. Given the above snake(s), we can find some useful data: the next ascending number(s) that need to be found (the head) and/or the next descending number(s) that need to be found (the tail). Using the above example, one of the next heads we can solve for is 4 (the “head” of 3). The algorithm will look at the surrounding empty squares and see if there is a “sure” placement for 4. Sure enough, in the above example, we can place 4 below 3 to form a head for {1, 2, 3} and a tail for {5, 6, 7}. This will combine both snakes into one snake: {1, 2, 3, 4, 5, 6, 7}. The algorithm will iteratively solve for the next heads and tails and continue to combine snakes until we have one snake of size n x n where n is the size length of the grid. However, as we can see from this example, we cannot (necessarily) be sure of where the next head should be placed. The next (and only) head we can look for is 8. However, we do not know if it can surely be placed in the cell below or the cell to

the left of 7. Once the solver has come to the point where it cannot solve for more “heads” or “tails”, it progress to the next phase.

Phase 2 – Heuristic search:
In this approach, the solver will attempt to start “guessing” where the next heads and tails should go. The heuristic will use a depth first search with the known length of the search. The solver will first choose which “head” or “tail” to start at. It will do so by finding the differences between each head and tail that should connect and start by attempting the approach on the path and tail with the shortest distance between them. This is easily done because the solver has kept track of all the snakes in the grid. It also knows that snake 1 can only connect to snake 2 and snake 2 can only connect to snake 3 (because we are working in ascending order). Thus, the workload is reduced. The solver then start solving for the head with the shortest distance the next tail it must attach to.

The solver will essentially enumerate the paths it can take to complete the next snake. Once it has done so, it will keep track of the path taken and the “guesses” that were added (in case they were incorrect guesses). The solver will then repeat the cycle of applying phase 1 and phase 2 until the grid is filled or until the grid cannot be filled.

Post “completion”
Once the grid is detected to have been filled or has detected that it can no longer add new values, it will check itself. If it sees that the grid is incorrect, it will remove the last “guesses” and enumerate a new path. The solver will continue this process of enumerating paths and removing invalid paths until a solution can be found.

What Will Try to Be Implemented:
After making the above solver, I will attempt to make a relatively efficient brute force solver to see if there is any merit in the brute force approach. The above solver is an “intelligent” way to approach the problem, but it does not necessarily make use of the strengths of a computer. Firstly, computers are first and so, do not necessarily need to be “intelligent” in order to solve a problem quickly. Secondly, given a relatively small table size, a brute force method might be able to exploit caching. Caching might allow the brute force method to quickly determine paths that will not work and simply move on.

Furthermore, the fact that the entire table need not be checked in order to determine if it is a solution can be exploited for speed as well. If within two segments of the number one, we fail to continue sequentially in a non-diagonal direction, then we can stop the search and move on to a new table after just two iterations of the table. The hope is that caching and this efficient error detection might be enough to create a fast solver for the Numbrix game.