Carlos Vasquez

1111-2307

CAP 4621

Final Report

1. **Description of Game**

The game implemented is called Numbrix. The goal of the game is to fill out a N by N grid with numbers between one and and each number can only be used once. The player is given this grid with some of the cells already populated with a number. From there, the player must attempt to fill in the entire grid by finding every increasingly or decreasingly consecutive number in the vertical or horizontal directions (no diagonal directions) from a cell. Once the player completes the grid, the player should be able to start from the cell with value one and trace a non-terminating line of consecutive numbers in a non-diagonal direction up until the line reaches the number .

1. **Description of Implementation Approach**

Considering the competitive nature of the project, there was a focus on speed and memory. The goal was to solve the Numbrix grid quickly while consuming as little resources as possible. Due to the use of recursive objects in the program, I ended up utilizing multiple static variables to help reduce the memory that needs to be consumed when solving the grid.

1. **Description of Programs, Procedures, Methods and Variables**

All the solver components of the program can be found under the numbrixgame.system.solver package, although it also makes use of the Log class in the numbrix.system package. The core class in the solver package is the Solver class which utilizes a constraint search to solve the Numbrix gird. To start, Solver is constructed and given a NumbrixSystem object (system) with which to obtain the grid. Then, solved() is called to begin the solution. This will start the clock and call the initialize() method which will initialize the ConstraintSearch (constraint) and HeuristicSearch (heuristic) objects used by Solver along with the Snake (sake) used by the Solver.

Snake is a very important class in this implementation. Snake will keep a list of every consecutive value found along with their positions. It does so by keeping a LinkedList of LinkedLists (snake) which contain each discovered consecutive value in ascending order. The LinkedLists are of the Triple type which is a custom class that essentially keeps a value, x position, and y position. The importance of this data structure will be elaborated on in part six of the report. For now, know that Snake supports the following (among other) key methods: the removal of an element given its value via the remove() method, returns the Triple with a given value via the find(), returns the the last or first element of a list with a given value via the findTip() method, and returns both ends of the LinkedList that contains the specified value.

Once the Snake has been initialized, the Solver will initialize the HeuristicSearch which, along with the Snake, is a static data type. This is important because the Solver will be recreated in a series of recursive calls and it is important to not waste time or memory creating data structures that need only be created once and can be used statically.

Of important note is the fact that the ConstraintSearch and HeuristicSearch objects both extend the SearchMethod abstract class which, among other things, allows the ConstraintSearches and the HeuristicSearch share the Snake (snake) and NumbrixSystem (system) objects as a static data type. These objects are set after the construction of the HeuristicSearch. After these two objects have been set, the initialize method finishes and the process returns to the solve() method which moves on to starting the constraint search by calling constraintSatisfactionSearch(). Herein lies the core functionality of the Solver class.

constraintSatisfactionSearch() simply makes a call to the constraintSearch() method in Solver which will return a boolean of whether or not a solution has been found. If one was not found, it calls heuristics startSearch() method. A more in depth discussion on HeuristicSearch.startSearch() can be found in part size of this paper, but the important thing to note is that startSearch() will do a brute force search with the goal of combining two lists in the snake into one list. It will then check if the grid has been solved and if not, it will create a new Solver and call Solver.search(). Hence, there is this recursive relation between Solver and HeuristicSearch.

A bit of information on constraintSearch(), this method will simply call constraints startSearch() method in the forwards and backwards directions until no more constraints can be found. More information on the intelligence behind this search can be found in part six of this paper. However, it is important to note that ConstrantSearch keeps track of all additions made by the CosntraintSearch object and can undo these changes by calling on it’s solver objects remove() method and passing in a Triple. This log of moves is a Stack of type Triple called additions. The important methods in here are simply the startSearch() method which starts the search() method which recursively searches for the next cell to populate.

On the other hand, HeuristSearches.startSearch() will prepare the HeuristcSearch and then begin a recursive brute force search by calling the search() method. Because every HeuristicSearch has its own Solver (solver), it also has its own ConstraintSearch from its solver. Thus, the search() method attempts to populate a target cell by calling search() on a cell (triple) from a particular direction (direcetion) with a specified count (nodeCount) and a variable that tells the method if the search is directed in a forward or backward direction (increment). More information about this method will be elaborated on in part six of the report, but what is important to note is that the search method recursively calls itself by decrementing the nodeCount variable. Once this count reaches 1, it stops searching deeper for another cell. Furthermore, the recursive nature of the method makes this a depth first search with a known length. This length is the distance needed to combine the initial triple with the next known triple from the snake. Once the last node has been reached or the method is unable to continue further, the search will check to see if the board has been solved. If it has not been solved, it will create a new Solver and call its search method to start the recursive chain again. If it finds that the guesses made were bad (that is, the solvers search() method returns false), HeuristicSearch.search() will remove the triple it added to snake and the grid and will return a false. This will recursively exit the method and backtrack to the previous search() method so that it may try a different triple (a different direction) or it may undo its own changes and backtrack the same way. If the entire stack of search() methods returns false, the process returns to the calling method (Solver.solve()) which will then call ConstraintSearch.undo() to undo changes made by constraint. In other words, this will undo additions made by the most recent constraint object so that a new branch of guesses can be made without the remnants from the bad guesses made by HeuristicSearch.

1. **Flowchart**
2. **UML**
3. **Description of Intelligence Implemented**
4. **Discussion of Changes Wished For**

**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.