Sina Nejati
nejat001@cougars.csusm.edu
571 – Artificial Intelligence
Prof. Guillen
11/09/2011
Design Document: Individual Project 2 (Sudoku Puzzles)

Introduction:

I decided to implement a Sudoku solver mainly because I had never played Sudoku and it seemed to be more fun learning Sudoku and solving it at the same time. It also seemed more practical because so many people are playing Sudoku (Including my dad!)And I can showcase my program to them.

Implementation:

I chose a simple back tracking search algorithm for the Sudoku puzzle because it is guaranteed to solve the problem no matter how hard it is. This algorithm compared to AC3 takes more time and space since we are not taking all constraints into consideration. Back tracking works for a wider range of Sudoku puzzles.

The back tracking algorithm I Implemented works by assigning a number to a each block of the Sudoku puzzle then checking if it's node consistence and it moves on until there are no possibly values for the block that is being analyzed, from then we backtrack to the latest block we analyzed and change the value given to that block (by incrementing it once) then move forward again. If there are no remaining values for the last block we move on to the previous block and so on until either all the blocks have a value or no solution is found.

Design:

Overview:

In implementing this algorithm we define the Sudoku puzzle as a double array, **grid [I] [j]**. [Figure 1] We then continue to assign numbers to our 9 to 9 grid from a sample Sudoku puzzle and putting 0 for the empty cells. I have two classes the Sudoku and the Solver. The first one is where the program is run. In the Sudoku class the grid is defined and passed to a **validate** function which validates the grid to check if it matches the three Sudoku consistencies that we already have in the description of our problem. Upon meeting these consistencies, the grid is then passed to **btsearch** function in the solver class; the purpose of the btsearch function is to complete the 0 values in the grid to values under 9 while keeping the grid valid in each step.

0	0	0	2	6	0	7	0	1
6	8	0	0	7	0	0	9	0
1	9	0	0	0	4	5	0	0
8	2	0	1	0	0	0	4	0
0	0	4	6	0	2	9	0	0
0	5	0	0	0	3	0	2	8
0	0	9	3	0	0	0	7	4
0	4	0	0	5	0	0	3	6
7	0	3	0	1	8	0	0	0

Figure 1 – A 9 by 9 Double array

Classes:

Sudoku: we store the main program here; the grids are passed to functions called from the solver class.

Solver: Includes the tools necessary to solve a Sudoku puzzle.

Functions:

Validate:

Validates either the whole grid or a cell in the grid by checking if the value of cell in a grid is present in the row , column or the 3 by 3 [ ] . If it is valid it's will return false if not it moves on to return true. And for the grid it's will check the constraints for all the cells in the gird. Inputs for this function are either the grid or the coordination of the cell in a grid.

Print Grid:

This method is a simple function to print the grid in any time.

CellList:

Inputs a grid and checks all the cells in this grid to find the coordinates of the empty cells which by design we set their values to be zero. It then proceeds to store these coordinates (i and j) in a separate array for the search function to deploy the back tracking search. It works by finding the number of empty cells first (n), then creating an n to 2 array. The method will store the I and j of the empty cells as the coordinates for this array. In the end this method will return an n to 2 arrays.

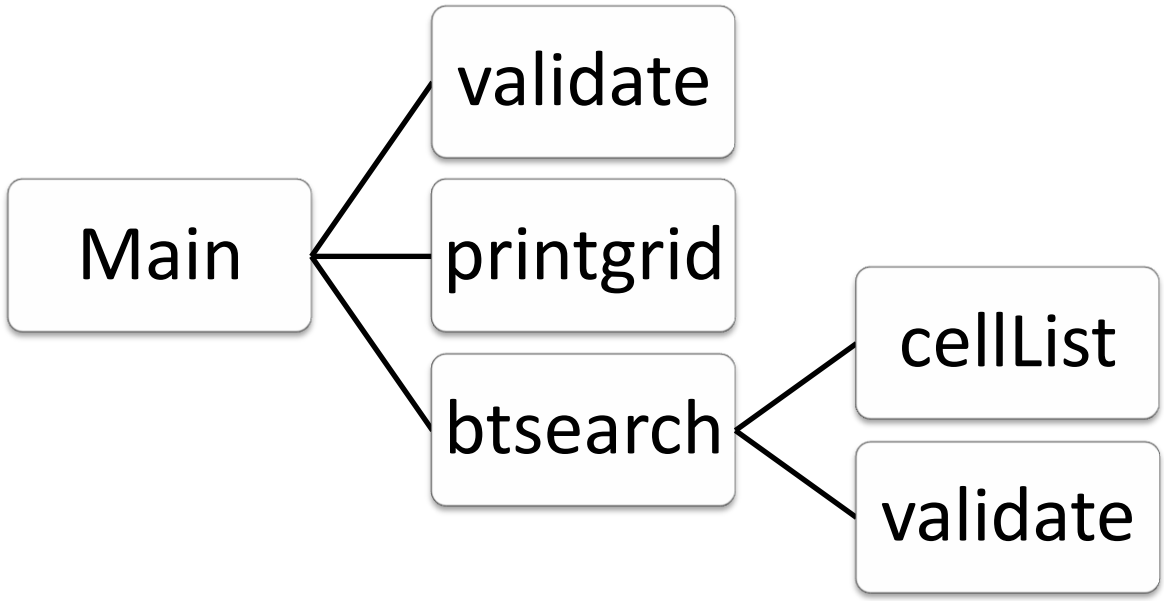
BT search:

This method starts with the first empty cell which its coordinates are stored in the cellist by the **Cellist** function. Method continues to Increment the value of that cell by one and validates the new grid if the grid is valid it moves on to the next cell store in the cell list by the CellList array. If the value after incrimination is equal to 9 it means that no values are valid for that specific cell therefore we deploy the backtracking by moving to the previous cell and since we already have a value we just increment it again and so on. If there is a cell that has no values after backtracking is done, this function returns false, else it processed to return true.

Copy:

It copies the values of a double array to another one.

Design Diagram:



Example:

I tested this algorithm with three Sudoku puzzles from Arizona university website[i] while it took 1 millisecond to solve the easiest Sudoku the hardest one took a lot more time and computations in comparison to the easy ones. I believe that while this algorithm works for the entire Sudoku puzzle it might not be the ideal choice when you are dealing with hard Sudoku puzzles.

```
Enter the type of Sudoku you would like to solve: hard
 0 |  2 |  0 |  0 |  0 |  0 |  0 |  0 |  0 |
-----------------------------------------
 0 |  0 |  0 |  6 |  0 |  0 |  0 |  0 |  3 |
-----------------------------------------
 0 |  7 |  4 |  0 |  8 |  0 |  0 |  0 |  0 |
-----------------------------------------
 0 |  0 |  0 |  0 |  0 |  3 |  0 |  0 |  2 |
-----------------------------------------
 0 |  8 |  0 |  0 |  4 |  0 |  0 |  1 |  0 |
-----------------------------------------
 6 |  0 |  0 |  5 |  0 |  0 |  0 |  0 |  0 |
-----------------------------------------
 0 |  0 |  0 |  0 |  1 |  0 |  7 |  8 |  0 |
-----------------------------------------
 5 |  0 |  0 |  0 |  0 |  9 |  0 |  0 |  0 |
-----------------------------------------
 0 |  0 |  0 |  0 |  0 |  0 |  0 |  4 |  0 |
-----------------------------------------
Computations needed to solve :705624
Found!
It took : 74 Miliseconds
 1 |  2 |  6 |  4 |  3 |  7 |  9 |  5 |  8 |
-----------------------------------------
 8 |  9 |  5 |  6 |  2 |  1 |  4 |  7 |  3 |
-----------------------------------------
 3 |  7 |  4 |  9 |  8 |  5 |  1 |  2 |  6 |
-----------------------------------------
 4 |  5 |  7 |  1 |  9 |  3 |  8 |  6 |  2 |
-----------------------------------------
 9 |  8 |  3 |  2 |  4 |  6 |  5 |  1 |  7 |
-----------------------------------------
 6 |  1 |  2 |  5 |  7 |  8 |  3 |  9 |  4 |
-----------------------------------------
 2 |  6 |  9 |  3 |  1 |  4 |  7 |  8 |  5 |
-----------------------------------------
 5 |  4 |  8 |  7 |  6 |  9 |  2 |  3 |  1 |
-----------------------------------------
 7 |  3 |  1 |  8 |  5 |  2 |  6 |  4 |  9 |
-----------------------------------------
```

Conclusion:

In solving this problem I learned that while backtracking seems like the most efficient algorithm it still comes short taking the time and computations into consideration and for solving that we need to implement heuristics. I had a hard time trying to get the backtracking working. I had to reconstruct my search function few times to get it working too.

[i] http://dingo.sbs.arizona.edu/~sandiway/sudoku/examples.html