CS171 Project Progress report #4: MRV & DH Heuristic

Date: March 06, 2016

1 Scope

The purpose of this assignment is to implement the MRV & DH Heuristic component of Sudoku solver. Our team will be continuing off of the provide Java shell and changes made from the last assignment.

2 Progress

We've implemented MRV and DH based on the explanation from lecture. MRV is simple as it checks all the unassigned variables in the puzzle and looks for the variable with the smallest domain size and returns it. This means that the variable is chosen to be assigned since it has the fewest legal moves. DH checks all the unassigned variables in the puzzle, for each unassigned variable, DH checks its neighbors and keeps a count of how many neighbors are unassigned. After all the unassigned variables are checked, DH returns the variables that has the most number of neighbors that are unassigned. This means that the variable is chosen to be assigned since it has the most constraints on remaining variables.

MRV with DH is more complicated since DH is used as a tiebreaker for MRV, therefore MRV needs to return a list of variables that are tied in domain size and are most constrained (minimum domain size). MRV by itself only returns one variable. Therefore we're written a third function to be called when both MRV and DH are specified as tokens and call that function when both tokens are specified. This function first uses a modified MRV heuristic where it keeps track of what the minimum domain size is and all variable that all have that domain size. It will then return these variables and call the DH function on them as a tiebreaker.

3 Problems & Questions

There were no problems with MRV and DH, however it was difficult to figure out how to implement MRV with DH, as the code for the shell only allows either MRV or DH to be called, and MRV did not satisfy our needs, initially we tried to modified the MRV function, to take in a parameter so that it will return the most constrained values, however we realized that the return types are different, thus this is impossible. A third function was necessary for MRV with DH to be implemented correctly.

4 Results

The program takes in correct inputs (input file, output file, timeout, and MRV (Minimum Remain Variables) or DH (Degree Heuristic), or both MRV and DH (MRV with DH as tiebreaker for most constrained variables), correctly solves the Sudoku puzzle using the back track search implementation along with heuristic functions to choose the variable that is most likely to lead to success. Some of the most difficult puzzles can finally be solved within a reasonable time limit (5 minutes) with the ACP, MAC, MRV, DH tokens, however the node count is rather large.

Appendix

```
private Variable getMRV()
{
    Variable leastRemaining = null;
    for(Variable v : network.getVariables())
    {
        if(!v.isAssigned())
        {
            if (leastRemaining == null || v.size() < leastRemaining.size())
            {
                leastRemaining = v;
            }
        }
    }
    return leastRemaining;
}

private Variable getDegree(List<Variable> values)
{
    Variable degreeheuristic = null;
    int max_neighbors_unassigned = 0;
    for(Variable v : values)
    {
        if(!v.isAssigned())
        {
            int neighbors_unassigned = 0;
            for(Variable vOther : network.getNeighborsOfVariable(v))
            {
                if(!vOther.isAssigned())
                {
                    neighbors_unassigned++;
                }
            }
            if (neighbors_unassigned > max_neighbors_unassigned)
            {
                max_neighbors_unassigned = neighbors_unassigned;
                degreeheuristic = v;
            }
        }
    }
    return degreeheuristic;
}
```

```
private Variable getMRVDH()
{
        List<Variable> leastRemaining = new ArrayList<Variable>();
        int minimum = -1;
        for(Variable v : network.getVariables())
        {
                if(!v.isAssigned())
                {
                        if (minimum == -1 || v.size() < minimum)
                        {
                                minimum = v.size();
                                leastRemaining.clear();
                                leastRemaining.add(v);
                        }
                        else if (v.size() == minimum)
                        {
                                leastRemaining.add(v);
                        }
                }
        }

        if (leastRemaining.isEmpty())
                return null;

        if (leastRemaining.size() == 1)
                return leastRemaining.get(0);

        return getDegree(leastRemaining);
}
```