Rodrigo Hernandez

48035611

CS 171

Progress Report: MRV, LCV, and DH

# Scope

This update implements the following Heuristics: MRV,LCV, and DH for solving Monster Sudoku problem.

# Progress

The Sudoku solver uses a switch-case statement when determining which consistency check to use. If the FC argument is passed as an argument, then the forward checking consistency is used. The forward checking function works in the following way: for each variable, it checks it's neighbouring nodes to check if they share the same value as the current variable. Forward checking also deletes from the domain of the nieghbouring nodes the current value of the current node. This results in the domains of each nodes dwindling as it is checked for consisntency. If at any point neighbouring nodes share the same value or if the domain of a node becomes empty as a result of forward checking, it determines the current board as not consistent and the solver backtracks.

* Added an additional enum for the variable heuristic called MRVDH, which is set as the variable heuristic if both MRV and DH are present
* If the heuristic is MRV or MRVDH, then the getMRV function is called.
  + This function finds the variable with the most remaining values.
  + If there is a tie and the MRVDH heuristic is on, then it breaks the ties by finding the one with the highest degree by using the findDegree function
* If the heuristic DH, then it calls the function getDegree.
  + This function finds the variable with the highest degree.
  + This function itself uses the findDegree function
* If the value heuristic is LCV, then it returns the list of values ordered by their LCV value
  + This is done the following way: for every value of a variable, check the variable's neighbours and add a count of how many times it appears in the neighbours domain. You then sort the list of values by this value in increasing order and return it
* Added code to write the outputs to a file
* Added code to write a sudoku board in tuple form.

# Problems and Questions

The major issue in this part of the project was figuring out how to sort the values by the LCV heuristic. While it was easy to find how many constraints a value had, finding how to sort them by this value was a little troublesome. I ended up having pairs where the key was the number of constraints added and the value was the value from the domain. This was then sorted using Java's collection.sort() function. The values where then extracted in order and put in a list that was eventually returned.

Appendix

***getMRV Function***

/\*\*

\* **TODO**: Implement MRV heuristic

\* **@return** variable with minimum remaining values that isn't assigned, null if all variables are assigned.

\*/

**private** Variable getMRV()

{

**int** min = Integer.***MAX\_VALUE***; // Initialize min to be highest possible value

**int** minIndex = -1; // Store the index of the variable with the MRV to avoid

// searching for variable again

**int** currentIndex = 0;

**for**(Variable v: network.getVariables())

{

**if**(!v.isAssigned())

{

**if** (v.size() < min)

{

min = v.size();

minIndex = currentIndex;

}

// If DH is also on, break ties with lowest DH

**if**(v.size() == min && varHeuristics == VariableSelectionHeuristic.***MRVDH***)

{

**if**(findDegree(v,network) > findDegree(network.getVariables().get(minIndex),network))

{

min = v.size();

minIndex = currentIndex;

}

}

}

currentIndex++;

}

**if**(minIndex == -1) // Means every node has been assigned.

**return** **null**;

**return** network.getVariables().get(minIndex);

}

***getDegree Function***

/\*\*

\* **TODO**: Implement Degree heuristic

\* **@return** variable constrained by the most unassigned variables, null if all variables are assigned.

\*/

**private** Variable getDegree()

{

**int** maxDegree = -1; // Initialize min degree to be highest possible value

**int** maxIndex = -1; // save index to avoid searching for minDegree variable again

**int** currentIndex = 0;

**for**(Variable v: network.getVariables())

{

**if**(!v.isAssigned())

{

**int** degree = findDegree(v,network);

// Now check if degree is the minimum.

// If so, update minDegree and save Index

**if** (degree > maxDegree)

{

maxDegree = degree;

maxIndex = currentIndex;

}

}

currentIndex++;

}

**if**(maxIndex == -1) // Means every node has been assigned.

**return** **null**;

**return** network.getVariables().get(maxIndex);

}

***findDegree function***

/\*\*

\* Finds the degree of Variable v

\* **@param** v Variable to find degree for

\* **@param** cn Constraint Network variable belongs to

\* **@return** degree of Variable v

\*/

**private** **int** findDegree(Variable v, ConstraintNetwork cn)

{

**int** degree = 0;

List<Variable> neighbours = cn.getNeighborsOfVariable(v);

// For each neighobur, add one to degree of variable v is

// neighbour is unassigned

**for**(Variable n: neighbours)

{

**if**(!n.isAssigned())

degree++;

}

**return** degree;

}

***getValueLCVOrder Function***

/\*\*

\* **TODO**: LCV heuristic

\*/

**public** List<Integer> getValuesLCVOrder(Variable v)

{

// Key is the number of constraints

// Value is the value in Domain

List<Entry<Integer,Integer>> pairs = **new** ArrayList<Entry<Integer,Integer>>();

List<Variable> neighbours = network.getNeighborsOfVariable(v);

**for**(**int** value: v.getDomain().getValues())

{

**int** count = 0;

**for**(Variable n: neighbours)

{

**if**(n.getDomain().contains(value))

count++;

}

pairs.add(**new** AbstractMap.SimpleEntry<Integer,Integer>(count,value));

}

// Comparator used to get pairs in order of constraints.

Comparator<Entry<Integer,Integer>> valueComparator = **new** Comparator<Entry<Integer,Integer>>()

{

@Override

**public** **int** compare(Entry<Integer,Integer> p1, Entry<Integer,Integer> p2)

{

**return** p1.getKey().compareTo(p2.getKey());

}

};

Collections.*sort*(pairs,valueComparator);

List<Integer> result = **new** ArrayList<Integer>();

// Add only values onto the result

**for**(Entry<Integer,Integer> p: pairs)

{

result.add(p.getValue());

}

**return** result;

}