CS171 Project Progress Report: MRV, DH, and LCV

Name: Nicky Huynh
ID: 76579985

Date: March 6, 2016

1. Scope
   I have implemented the MRV, DH, and LCV algorithms for solving the Sudoku problem.
2. Progress
   At this point, I have completed all parts of the Monster Sudoku project except for the arc consistency sections. Other than those sections, the only parts that have been changed are the representation of a puzzle in a File form. These changes can be seen in the SudokuFile.java and the BTSolver.java files.
3. Problems and Questions
   I still have the same problem regarding the rotation of the Sudoku puzzle after it has been solved. Although the puzzle is correct, it is rotated, so the results may differ from the tester's puzzle. In addition, I believe my forward checking algorithm is incorrect because the number of assignments it returns is less than the number of squares in the sudoku puzzle, but it outputs correctly. I also have some questions about the number of assignments and backtracks that each option should add or subtract. I understand that a better search pattern will most of the time result in more resources being used, but which ones will do which? If I could get some feedback on this, I would be grateful.
4. Results
   The above "number of assignments" issues were the only surprising results. I have found that not all of the algorithms will make your search run faster.

Appendix

MRV -

```
private Variable getMRV()
{
        List<Variable> unassigned = new ArrayList<Variable>();
        for(Variable v : network.getVariables())
        {
                if(!v.isAssigned())
                {
                        unassigned.add(v);
                }
        }

        //if there are no unassigned variables then your sudoku puzzle is complete
        if(unassigned.size() == 0)
        {
```

```
        return null;
}

int vSize; //holds the size of variable's values
int minSize; //holds size of the current min's values
int vNeighbor; //holds the number of neighbors of V (DH checking)
int minNeighbor; //holds the number of neighbors of min

Variable min = unassigned.get(0);
for(Variable v : unassigned)
{
        vSize = v.getDomain().getValues().size();
        minSize = min.getDomain().getValues().size();
        if(vSize < minSize) //if new var has less values then it is new MRV
        {
                min = v;
        }
        else if(vSize == minSize) //use dh as tiebreaker
        {
                if(dh)
                {
                        vNeighbor = 0;
                        minNeighbor = 0;
                        for(Variable var : network.getNeighborsOfVariable(v))
                        {
                                if(!var.isAssigned())
                                {
                                        vNeighbor++;
                                }
                        }

                        for(Variable var : network.getNeighborsOfVariable(min))
                        {
                                if(!var.isAssigned())
                                {
                                        minNeighbor++;
                                }
                        }

                        if (vNeighbor > minNeighbor)
                        {
                                min = v;
                        }
```

```
                        }
                    }
                }

                return min;
        }
```

DH -

```
        private Variable getDegree()
        {
                int numConstraints = -1;
                int neighbors;
                Variable next = null; //if there are no variables return null
                for(Variable v : network.getVariables())
                {
                        if (!v.isAssigned())
                        {
                                neighbors = 0;
                                for(Variable var : network.getNeighborsOfVariable(v))
                                {
                                        if(!var.isAssigned())
                                        {
                                                neighbors++; //increment if there are neighbors
unassigned
                                        }
                                }
                                if(neighbors > numConstraints)
                                {
                                        numConstraints = neighbors; //means that current var has
more neighbors unassigned
                                        next = v;
                                }
                        }
                }
                return next;
        }
```

LCV -

```
public List<Integer> getValuesLCVOrder(Variable v)
        {
                List<Integer> values = v.getDomain().getValues();
                List<Variable> neighbors = network.getNeighborsOfVariable(v);
                final List<Integer> domains = new ArrayList<Integer>();
```

```
		for(Variable var : neighbors)
		{ //add all neighbors, including repeat variables
			domains.addAll(var.getDomain().getValues());
		}

		Comparator<Integer> valueComparator = new Comparator<Integer>(){
			@Override
			public int compare(Integer i1, Integer i2)
			{
				int count1 = 0; //count occurrences of i1 in domains
				int count2 = 0; //count occurrences of i2 in domains
				for(Integer i : domains)
				{
					if(i == i1)
					{
						count1++;
					}
					if(i == i2)
					{
						count2++;
					}
				} //return the lower one
				return
((Integer)(domains.size()-count2)).compareTo((Integer)(domains.size()-count1));
			}
		};
		Collections.sort(values, valueComparator); //sort by lcv
		return values;
	}
```