Progress Report

Team Windwolf

Roldan Gammad, 29423247

Sai Smaran Macha, 92269171

**1 SCOPE**

We have implemented Minimum Remaining Value, Degree Heuristic, and Least Constraining Value algorithms for backtracking. We also fixed some bugs in our code. All code is implemented in Java using the provided Java coding shell.

**2 PROGRESS**

MRV

We created a list of all assigned variables in the network, iterated through that list to find the variable with the smallest domain. It does not function by itself because it depends on domain size but it significantly decreases the amount of backtracking the solver does with forward checking.

DH

We utilize a HashMap to store variables and the number of unassigned variables constraining them. We then iterate through the map and find the key with the highest value and return that.

DH + MRV

We created a separate function that utilizes the MRV algorithm to create a list of variables to which we pick a variable utilize DH techniques. We also created a new VariableSelectionHeuristic enum called “MRVDH” to handle the case where MRV and DH are called in the same command line.

LCV

We created a two lists, one storing values in a variable's domain and another storing number of neighboring variables that are constrained by each value in the domain. We iterate through both lists storing each into a HashMap the Key being a domain value and the Value being the number of variables it constrains. We then utilize Collection.sort on the domain values list with a new Comparator that compares the value of each key in the HashMap to return a sorted list of domain values.

**3 ISSUES/CONCERNS**

MRV does not appear to work on small problems.

MRV is more apparent on bigger problems.

LCV and MRV do not reduce solution times without FC.

Whenever we include DH as one of the tokens, we get various issues.

by itself, it times out, but when paired with FC on some puzzles it finds a solution within the timeout limit

The same goes for MRV+DH.

We have concluded that our DH algorithm may not be properly implemented, thus causing these issues.

**4 APPENDIX**

**main.java:**

added new VariableSelectionHeuristic, MRVDH in token check

} else if (DH && MRV) {

solver.setVariableSelectionHeuristic(VariableSelectionHeuristic.MRVDH);

**outputLog.java**

switched board rows and columns back to correct state when producing output.

for(int i = 0; i < solnBoard.getN(); i++){

for(int j = 0; j < solnBoard.getN(); j++){

retStr += (Integer.toString(sfBoard[j][i]) + ", ");

}

**BTSolver.java**

**VariableSelectionHeuristic addition**

public enum VariableSelectionHeuristic {

None, MinimumRemainingValue, Degree, MRVDH

};

**in selectNextVariable switch function**

case MRVDH:

next = getMRVDH();

break;

**MRV function**

private Variable getMRV() {

List<Variable> vList = new ArrayList<Variable>();

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

if (!v.isAssigned()) {

vList.add(v);

}

}

if (!vList.isEmpty()) {

int minSize = vList.get(0).getDomain().size();

Variable minV = vList.get(0);

for (int i = 1; i < vList.size(); i++) {

if (minSize > vList.get(i).getDomain().size()) {

minSize = vList.get(i).getDomain().size();

minV = vList.get(i);

}

}

return minV;

}

return null;

}

**DH function**

private Variable getDegree() {

Map<Variable, Integer> varMap = new HashMap<Variable, Integer>();

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

if (!v.isAssigned()) {

int count = 0;

for (Variable vOther : network.getNeighborsOfVariable(v)) {

if (!vOther.isAssigned()) {

count++;

}

}

varMap.put(v, count);

}

}

if (!varMap.isEmpty()) {

int maxValue = Collections.max(varMap.values());

Variable retVar = null;

for (Map.Entry<Variable, Integer> var : varMap.entrySet()) {

if (var.getValue() == maxValue) {

retVar = var.getKey();

}

}

return retVar;

}

return null;

}

}

**MRV + DH function**

private Variable getMRVDH() {

List<Variable> vList = new ArrayList<Variable>();

List<Variable> dhList = new ArrayList<Variable>();

List<Integer> constraintCount = new ArrayList<Integer>();

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

if (!v.isAssigned()) {

int count = 0;

for (Variable vOther : network.getNeighborsOfVariable(v)) {

if (!vOther.isAssigned()) {

count++;

}

}

constraintCount.add(count);

vList.add(v);

}

}

if (!vList.isEmpty()) {

int minSize = vList.get(0).getDomain().size();

Variable minV = vList.get(0);

for (int i = 1; i < vList.size(); i++) {

if (minSize > vList.get(i).getDomain().size()) {

minSize = vList.get(i).getDomain().size();

minV = vList.get(i);

} else if (minSize == vList.get(i).getDomain().size()) {

dhList.add(vList.get(i));

}

}

int maxDeg = constraintCount.get(0);

Variable retV = dhList.get(0);

for (Variable v : dhList) {

int temp = constraintCount.get(vList.indexOf(v));

if (maxDeg < temp) {

retV = v;

maxDeg = temp;

}

}

return retV;

}

return null;

}

**LCV function**

public List<Integer> getValuesLCVOrder(Variable v) {

List<Integer> values = v.getDomain().getValues();

List<Integer> countVal = new ArrayList<Integer>();

Map<Integer,Integer> valueMap = new HashMap<Integer,Integer>();

for (int val : values) {

int count = 0;

for (Variable vOther : network.getNeighborsOfVariable(v)) {

if (vOther.isAssigned()) {

if (vOther.getAssignment() != val) {

count++;

}

}

}

countVal.add(count);

}

for(int i = 0; i < values.size(); i++){

valueMap.put(values.get(i), countVal.get(i));

}

final Map<Integer, Integer> mapforComp = valueMap;

Collections.sort(values,

new Comparator<Integer>() {

@Override

public int compare(Integer i1, Integer i2) {

Integer i1Key = i1;

Integer i2Key = i2;

Integer i1Value = mapforComp.get(i1Key);

Integer i2Value = mapforComp.get(i2Key);

return i1Value.compareTo(i2Value);

}

});

return values;

}