Constraint Satisfaction Problem

Seyed Mahdi Basiri Azad (Erfan)

February 21, 2016

1 Overview

In this project we wrote a Constraint-Satisfaction-Problem solver to find solutions to problems such as the Australia map coloring, n-Queen problem, Sudoku puzzle and circuit board problem.
At the core the solver is nothing but a simple backtracking DFS algorithm. However we try to decrease the number of recursive calls to the backtracking algorithm by using heuristics such as Inference and other optimization such as LCV(least constraining value) and MRV(minimum remaining values).
Below is my code for the backtracking algorithm. later we will look at each one of the optimization and their performance:

```
private Map<Integer, Integer> backtracking(Map<Integer, Integer> partialSolution) {
incrementNodeCount();
if(isComplete(partialSolution)){
return partialSolution;
}
Integer varID;
if(useMRV) {
varID = selectUnassignedVariable(partialSolution);
}else{
varID = FakeSelectUnasignedVariable(partialSolution);
}
Iterable<Integer> domainVals;
if(useLCV){
domainVals = orderDomainValues(varID, partialSolution);
}else{
domainVals = FakeOrderDomainValues(varID, partialSolution);
}
for(Integer value : domainVals){
HashMap<Integer, Set<Integer>> removed = new HashMap<>();//as backup
if(isConsistent(varID, value, partialSolution)){
partialSolution.put(varID, value);

boolean inferences;
if(useInference) {
inferences = inference(varID, partialSolution, removed);
}else{
inferences = true;
}
if(inferences) { //if inference worked
//add inferences to the assignment
Map<Integer,Integer> result = backtracking(partialSolution);
if(result != null){
return result;
}
}
}
//not consistent remove {var, value} and inferences from partialSolution
revertChanges(varID, partialSolution, removed);
}
return null;
}
```

As we can see the backtracking algorithm is simple a depth-first-search starting from one of the unassigned variables and continues until all the variables are assigned correctly (satisfy the constraints) or if the domain of one or more variables get empty before a consistency can be established.

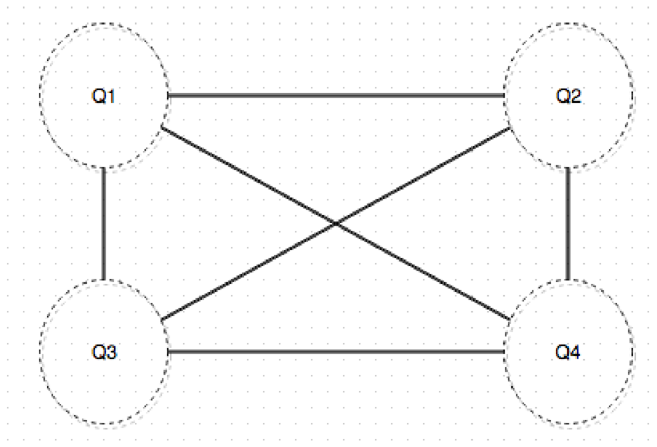
2 Optimization

Backtracking by itself is a simple DFS algorithm and in order to improve its performance we try to cut down the search in each branch if we get an indication that the said branch will not end with a solution. For this purpose we use the inference function to check the arc-consistency between the arcs starting from one variable and using MAC-3 algorithm propagate that consistency through the graph. If the inference finds out that a certain variable will have a domain of size zero after the revisions of its arc-consistency then the backtracking algorithm will be stopped from searching for possible solutions in that branch.

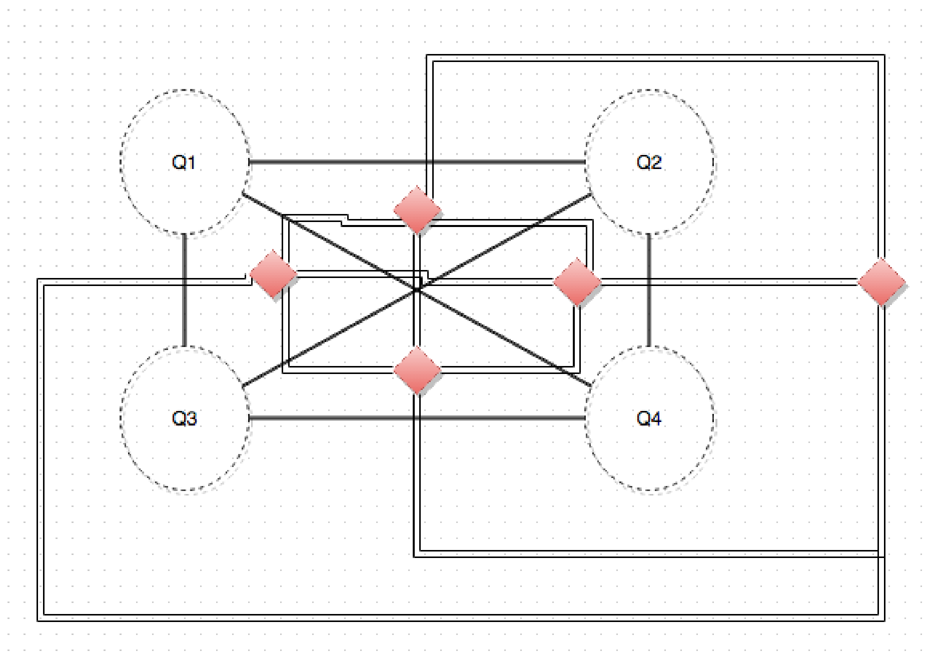
Along with inference we use the MRV algorithm for selecting the unassigned variable to progress with and we use the LCV to start with the least constraining value with respect to other variable domains.

3 n-Queen Problem

The aim is to pun n Queens in on a n*n chess board in such way that they don't attack each other. Below is a constraint graph for a 4-Queen case:



And the graph below shows the dual graph of the 4-Queen problem:



There are many symmetric solutions in the n-queen problems. we could reduce the number of solutions to the n-queen problem by changing the set up of the CSP. For example in a way that rotating the board by 90 degrees will not produce another solution. In this way we reduce the number of solutions by 4 fold while maintaining the fundamental 12 solutions.

below is a sample result for the 20-Queen puzzle:

```
Nodes explored during last search:  2292
Constraints checked during last search 288945
Search time is 2.74 second
[1, 3, 5, 7, 9, 11, 13, 16, 18, 20, 4, 19, 8, 12, 14, 6, 2, 10, 15, 17]
Process finished with exit code 0
```

In the given result I am using the Inference, LCV and MRV optimzations.

NOTE:There are boolean flags on top of ConstraintSatisfactionProblem.java file.

4 Sudoku

the aim of the Sudoku solver if to solve different instances of the Sudoku puzzle.

Below are my results for the Sudoku puzzle (the LONG-TEST version):

```
Nodes explored during last search:  82
Constraints checked during last search 3240
Search time is 0.09 second
[5, 1, 8, 7, 6, 2, 9, 3, 4]
[7, 4, 3, 5, 1, 9, 2, 6, 8]
[2, 9, 6, 3, 8, 4, 5, 7, 1]
[4, 8, 2, 6, 3, 7, 1, 9, 5]
[9, 3, 5, 8, 4, 1, 6, 2, 7]
[1, 6, 7, 2, 9, 5, 8, 4, 3]
[6, 7, 1, 4, 2, 8, 3, 5, 9]
[8, 2, 4, 9, 5, 3, 7, 1, 6]
[3, 5, 9, 1, 7, 6, 4, 8, 2]
Process finished with exit code 0
```

Again I was using all the optimization methods while solving this instance of the puzzle.

5 CircuitBoard problem

After implementing the CSP solver it was interesting to be able to frame a problem such as circuit board problem and be able to solve it with our CSP. I created the CircuitBoard class that denies the variables and constrains of the problem and uses an instance of CSP to solve the problem. For this purpose I also creates a private ComponentNode class to define the properties and domain of each component. Below is my code for the ComponentNode class:

```
  private class ComponentNode{
  int ID;
  int width;
  int height;
  Set<Integer> domain;


  public ComponentNode(int id, int w, int h, Set<Integer> s){
    ID = id;
    width = w;
    height = h;
    domain = s;
  }
  }
```

The domain of each variable/component was defined based on the size of its width and height and the size of the board. I took the lower left part of each component to to be the defining point of it. So while creating the set of white-list constraints i used the defining points of each component along with the values of their width and height to create the constraints. I have created the **getConstraints()** method for creating the set of constraints between two components.

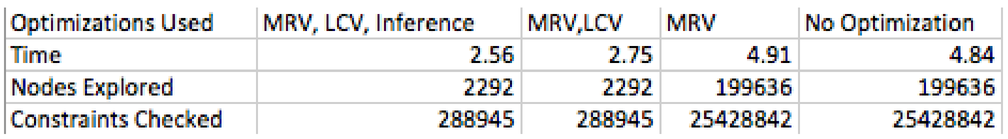
Below is the result of solving the sample circuit board problem using all the optimization methods (Inference, LCV, MRV):

```
Nodes explored during last search:  23
Constraints checked during last search 303
Search time is 0.03 second
[3, 3, 1, 1, 1, 2, 2, 2, 2, 2]
[3, 3, 1, 1, 1, 2, 2, 2, 2, 2]
[3, 3, 4, 4, 4, 4, 4, 4, 4, 0]
Process finished with exit code 0
```

The components are represented by numbers.

6 Comparing Results of Inference, MRV and LCV

Table below shows the difference between the results of the 20-queen problem based on different optimization.



Optimizations Used	MRV, LCV, Inference	MRV,LCV	MRV	No Optimization
Time	2.56	2.75	4.91	4.84
Nodes Explored	2292	2292	199636	199636
Constraints Checked	288945	288945	25428842	25428842

From the results we can see that MRV did not have much effect on the time, number of nodes explored or the number of constraints that were checked. On the other hand LCV was more exciting! Notice that after removing the LCV the time has improved meaning that the process got faster however the number of nodes explored has increased significantly!