Constraint Satisfaction Problem

Vivek Sah

February 20, 2016

1 Introduction

In this paper, we will discuss constraint satisfaction problem (CSP) . A CSP consists of three components, variables , domains, and constraints. Variables are like nodes in our search graph. Domain represent the possible values that the variables can taken. Constraints are set of relation which restrict the values that a pair of variables can take. Each constraint contains a pair of variables and a set of values that the pair can take. Examples of such problems are n-Queen, Sudoku, Chip board layout problem. We will discuss how to use backtracking algorithm to solve such problems. We will also try to optimize our search algorithm and analyze the results. In the last section, we will discuss how to formulate a chessboard layout problem.

2 Backtracking

We will implement backtracking algorithm as discussed in *Artificial Intelligence* by Russell and Norvig. Backtracking algorithm a variation of a regular DFS search.we modify the DFS to satisfy the constraint satisfaction problem. The algorithm repeatedly chooses an unassigned variable using *selectUnassignedVariable*, and then tries all values in the domain of that variable which are ordered according to *orderDomainValues* function , trying to find a solution. We make an assignment and use inference to find if the assignment resulted in success or failure. If we find success, we return the result otherwise, we try another value.

```
private Map<Integer, Integer> backtracking(Map<Integer,
    Integer> partialSolution) {
    incrementNodeCount();
  if(partialSolution.keySet().size() ==
      variables.keySet().size()){
       return partialSolution;
  }
  int variable = selectUnassignedVariable(partialSolution);

  for(int value : orderDomainValues(variable,partialSolution)){

    Map<Integer,Set<Integer>> removed = new HashMap<>();

    if(isConsistent(value,variable,partialSolution)){
        partialSolution.put(variable,value);
```

```
            if (inference(variable,value,partialSolution,removed))
                {
          Map<Integer, Integer> result =
              backtracking(partialSolution);
          if (result != null) {
               return result;
          }
            }
        }
        //if inference did not work, revert back the changes
        for(int removedVariable: removed.keySet()){
            for(int remVal: removed.get(removedVariable)){
          variables.get(removedVariable).add(remVal);
            }
        }
        partialSolution.remove(variable);
        removed.clear();
          }
    return null;
        }
```

2.1 Heuristics

2.1.1 Minimum Remaining Values (MRV)

While we select an unassigned variable, we can return the next variable in the set of variables in a uniform order, but it will make search slower. This algorithm picks a variable that is most likely to cause a failure soon, thereby preventing to traverse a significant part of tree. This heuristic will allow for fast-fail, which means if there is no solution, there is no point in searching three. Let us look at the implementation:

```
private Integer selectUnassignedVariableMRV(Map<Integer,
    Integer> partialSolution) {
            int minVar = 0;
            int minSize = Integer.MAX_VALUE;
            for (int k: variables.keySet()){
                if(!partialSolution.keySet().contains(k)){
                    if(variables.get(k).size() < minSize){
                        minSize = variables.get(k).size();
                        minVar = k;
                    }
                }
            }
            return minVar;
        }
```

2.1.2 Least Constraining Values (LCV)

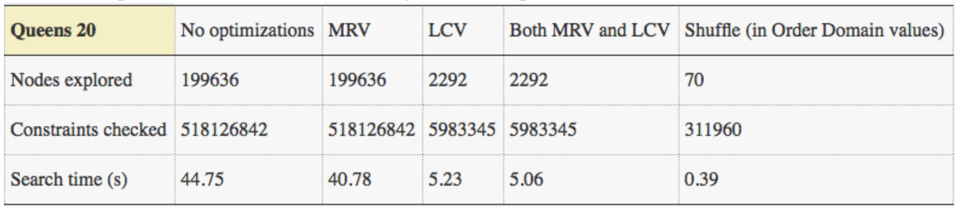
After we select a variable, we must decide on the order in which to examine its values. For this, the LCV heuristic will be effective. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. Overall, the algorithm is trying to leave the maximum flexibility for subsequent variable assignments. Here, we are trying to find the solution faster. So, we want to fail slow, so that if there is a solution, we get the solution fast. In the implementation, we will make a domain node class so that we can store how many constraints each domain satisfies. Let's look at a snippet of the implementation:

```
for (domainNode domain: domainScoreNodes){
    for(int neighbor: Neighbors.get(var)){
      for(int neighborDomain: variables.get(neighbor)){
         Pair<Integer,Integer> keypair = new Pair<>(var,
            neighbor);
         Pair<Integer,Integer> domainpair = new
            Pair<>(domain.getDomain(), neighborDomain);
         if(constraints.keySet().contains(keypair)){
              if(constraints.get(keypair).contains(domainpair)){
                   domain.setScore(domain.getScore()+1);
              }
                 }
      }
  }
}
```

3 Results

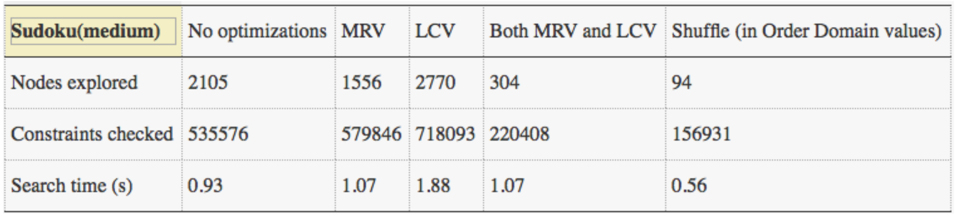
3.1 Queens

Here are the performance results on Queens-20 problem:



Queens 20	No optimizations	MRV	LCV	Both MRV and LCV	Shuffle (in Order Domain values)
Nodes explored	199636	199636	2292	2292	70
Constraints checked	518126842	518126842	5983345	5983345	311960
Search time (s)	44.75	40.78	5.23	5.06	0.39

Here are the performance results on Queens problem:

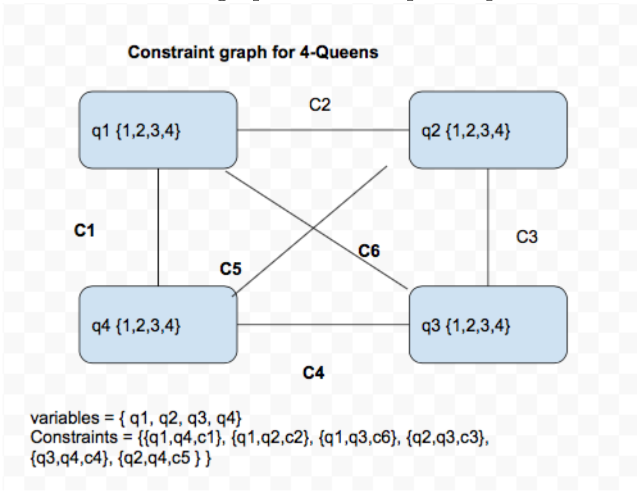


Sudoku(medium)	No optimizations	MRV	LCV	Both MRV and LCV	Shuffle (in Order Domain values)
Nodes explored	2105	1556	2770	304	94
Constraints checked	535576	579846	718093	220408	156931
Search time (s)	0.93	1.07	1.88	1.07	0.56

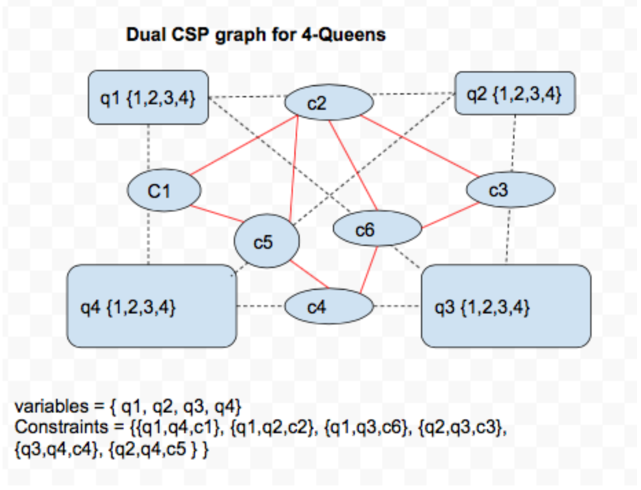
Comparing these values, we can see that although MRV results in fewer nodes explored in few cases, it produces best results when combined with LCV. There is also another surprising heuristic, that is, when returning ordered domains, we just shuffle them. This produces a surprisingly good result. It could be because shuffling takes less time and it should clearly produce better results than static order.

4 Discussion Questions

Below is a constraint graph for the 4-queens problem.


Constraint graph for 4-Queens
q1 {1,2,3,4}
C2
q2 {1,2,3,4}
C1
C5
C6
C3
q4 {1,2,3,4}
q3 {1,2,3,4}
C4
variables = { q1, q2, q3, q4}
Constraints = {{q1,q4,c1}, {q1,q2,c2}, {q1,q3,c6}, {q2,q3,c3},
{q3,q4,c4}, {q2,q4,c5 } }

Below is a dual constraint graph for the 4-queens problem.


Dual CSP graph for 4-Queens
q1 {1,2,3,4}
c2
q2 {1,2,3,4}
C1
c3
c5
c6
q4 {1,2,3,4}
c4
q3 {1,2,3,4}
variables = { q1, q2, q3, q4}
Constraints = {{q1,q4,c1}, {q1,q2,c2}, {q1,q3,c6}, {q2,q3,c3},
{q3,q4,c4}, {q2,q4,c5 } }

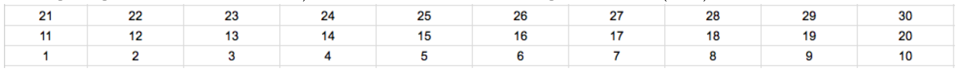
For 4-queens problem, we needed 8 edges.

5 Circuit Board Problem

In this problem, we are given a rectangular circuit board of size n x m, and k rectangular components of arbitrary sizes. Our task is to lay the components out in such a way that they do not overlap. For further discussions, we will use the follwing components(each are assigned a numerical value starting from 1 in respective order):

bbbbb, cc aaa , bbbbb , cc, cc ,eeeeeee, aaa

We will use a circuit board of size 10 x 3 for demonstration (stored as domain according to the integer given in the table). We will set the origin to be (1,1) which is lower left corner.



21	22	23	24	25	26	27	28	29	30
11	12	13	14	15	16	17	18	19	20
1	2	3	4	5	6	7	8	9	10

To initialize the problem, we will first add variable with their respective domain to the CSP problem. We loop through the variables set and for each variable, we loop through each position in table and determine if the variable can fit in that position. We make a private *Chip* class which stores variable's width and height. We make use of this width and height value to determine the domain. We convert each allowed position to an integer using the formula: $i + numBoardColumns * (j - 1)$ where i and j are x and y coordinates. After we populate the variables map, we then proceed to populate the constraint.

Populating constraint is pretty straightforward. We take two variables and loop through their domains. For each pair of domains, we determine if they are allowed. We do that by checking if the variables(can be pictured as a rectangular block) do not overlap each other. If they don't overlap, we add the constraint to the constraints set for that pair of variables. To check overlap, we need x and y coordinates from the integer domain values. We do that in the following way:

$intx1 = domain\%(numBoardColumns) == 0?numBoardColumns : domain\%(numBoardColumns)$
$inty1 = (domain/numBoardColumns) + 1;$

We do the overlap checking as shown below:

```
 if( ((x2>(x1+chips.get(i-1).width-1))  ||
     (x2<(x1-chips.get(j-1).width+1)))){
       constraint.add(new
          Pair<>(x1+numBoardColumns*(y1-1),x2+numBoardColumns*(y2-1)
          ));

        } else if ((y2>(y1+chips.get(i-1).height-1))  ||
            (y2<(y1-chips.get(j-1).height+1))){
       constraint.add(new
          Pair<>(x1+numBoardColumns*(y1-1),x2+numBoardColumns*(y2-1)
          ));
  }
```

5.1 Results

5.1.1 For given circuit board problem

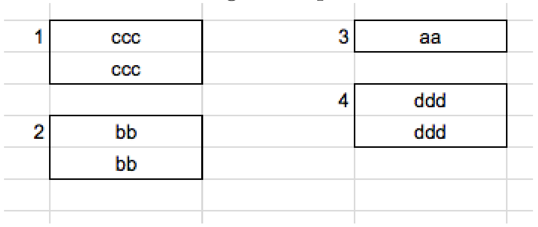
For the above given example, the CSP returns a solution (using both MRV and LCV)¿ Here's is the result.

[6, 6, 4, 4, 4, 4, 4, 8, 8, 8]
[5, 5, 1, 1, 1, 1, 1, 3, 3, 3]
[0, 7, 7, 7, 7, 7, 7, 7, 2, 2]
which translates to:

[c c b b b b b a a a]
[c c b b b b b a a a]
[. e e e e e e e c c]

5.1.2 For a more general problem

Let us look at a more general problem: The variables are defined as:



We are looking at a 5 x 4 grid.
The solution returned by the CSP is:
[4, 4, 4, 0, 0]
[4, 4, 4, 2, 2]
[1, 1, 1, 2, 2]
[1, 1, 1, 3, 3]
which translates to:
[d, d, d, 0, 0]
[d, d, d, b, b]
[c, c, c, b, b]
[c, c, c, a, a]