Constraint Satisfaction

Jing Ge

February 20, 2016

1 Introduction

This assignment has five main parts. The first part shows the implementation of a generic constraint satisfaction problem solver, includes heuristics functions MRV (minimum remaining values) and LCV (least constrained variable), an AC3(arc consistency 3) function to propagate the constraints, and a backtracking function to find solution by using FC(forward checking) or MAC-3(maintaining arc consistency 3). The second and third part discusses how to solve Sudoku and N-Queens problem by using the general CSP solver. The next section discusses the circuit-board layout problem that could be solved by CSP solver. The last part introduces paper related to CSP in improvement of coarse-grained arc consistency algorithm.

2 Implementation of Constraint Satisfaction Problem Solver

Constraint satisfaction problems are problems defined as a set of variables that each variable has a value and must satisfy a number of constraints or limitations. In order to solve a CSP problem, we check arc consistency of all variables first, eliminate all values that could not find a solution, sometimes we can find a solution only by inference over constraints. But if we cannot find a solution, then we must search to find a solution by using backtracking algorithm. There are several important functions used in backtracking to optimize the search algorithm and we explained them in the next few parts.

2.1 AC3

A variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints. The AC3 algorithm allows us to reduce the possible choices of values for the variable by checking consistency with this variable and its neighbors (the variables have constraint with this variable). The basic idea of AC3 is maintaining a queue, which contains all arcs(X and Y have a constraint) initially, then pops off an arc and makes X arc-consistent with respect to Y. The X's domain may be reduced in this progress. If D_x's size does't change, then we know the arc is safe and do nothing. If D_x's size is revised down to zero, then we could know that there is no solution to this CSP and AC3 returns failure. If this revises D_x but D_x still has values, then we add all X's neighbors and X into the queue, since their domains may be changed after we change D_x. The AC3 algorithm could efficiently reduce the domains of all variables and even find a solution.

AC3 (`enforceConsistency` and `revise`) is implemented in `ConstraintSatisfactionProblemr.java`. Here's my code:

```
private boolean enforceConsistency() {
    Queue<Pair<Integer, Integer>> queue = new LinkedList<>();
    // add all the arcs into queue
    for (Pair<Integer, Integer> pair : constraints.keySet())
        queue.add(pair);
    while (!queue.isEmpty()) {
        Pair<Integer, Integer> arc = queue.poll();
```

```
        // if the domain of X1 has been changed
        if (revise(arc.getKey(), arc.getValue())) {
            // the domain's size equals 0, can't find a solution
            if (variables.get(arc.getKey()).size() == 0)
                return false;
            // add all neighbors of X1 into queue
            for(Integer neighbor : neighbors.get(arc.getKey())){
                Pair<Integer, Integer> pair = new Pair<>(neighbor, arc.getKey());
                queue.add(pair);
            }
        }
    }
    return true;
}
```

2.2 Backtracking

The basic idea of Backtracking is very simple. Since AC3 could't find a solution in some cases, we could assign a value to a variable X. First, we check if this value could satisfy the partial solution we already have, then check the state of some arcs. If the state is safe, which means we can use this value for X, we add this value to the partial solution set and search in the next deeper level. If the partial solution fails (such as the next level) then we backtrack to the last safe level and assign another value in D_x to X and apply the search again. If we could assign every variable a value then we know we find a solution. If we find that there is no way to assign a value to a variable without some violations of constraint, then we know this CSP does not have a solution. We use some heuristics function such as MRV and LCV to determine which variable will be assigned and the order of choosing a value for this variable. We also have different ways to check if the state of whole variables is safe or not by using FC or MAC3.

verb'backtracking' is implemented in `ConstraintSatisfactionProblemr.java`. Here's my code:

```
private Map<Integer, Integer> backtracking(Map<Integer, Integer> partialSolution) {
    // every var has been assigned, find a solution
    if (partialSolution.size() == variables.size())
        return partialSolution;
    // select a var has not been assigned
    Integer var = selectUnassignedVariable(partialSolution);
    // get its domain
    Iterable<Integer> domain = orderDomainValues(var, partialSolution);
    Iterator<Integer> it = domain.iterator();
    while (it.hasNext()) {
        // select a value
        Integer value = it.next();
        boolean isValid = true;
        // record values deleted in this level
        Map<Integer, Set<Integer>> removed = new HashMap<>();
        //check if this value is valid or not
        for (Map.Entry<Integer, Integer> entry : partialSolution.entrySet()) {
            // please look at java file
            if (//... can't satisfy!)
                isValid = false;
                break;
        }
        partialSolution.put(var, value);
        if (isValid) {
```

```
            // check with some arcs
            if (inference(var, value, partialSolution, removed)) {
                // search in next level
                Map<Integer, Integer> solution = backtracking(partialSolution);
                if (solution != null)
                    return solution;
            }
        }
        partialSolution.remove(var);
        // if can't find a solution, add all we have removed to original set
        ... recover domains
        }
    }

    return null;
}
```

2.3 Heuristics Function: MRV and LCV

There are two ways we can accelerate our search algorithm. In `selectUnassignedVariable`, we can choose the next unassigned variable in a specific order to improve the search algorithm. In this solver, we use minimum remaining values heuristic(MRV). Variable with the fewest legal values will be chose first in this function. The reason is that we want the variable selection fails as soon as possible, then we can revise down the domain of a variable to 1 or 0 (to find a solution or return failure). Another way is using least-constraining-value heuristic(LCV). We choose the value that will rules out the fewest choices of this variable's neighbors' domains, which means we can leave more choices to other variables. The efficiency of those two heuristics will be discussed in N-Queens and Sudoku problems.

MRV(`selectUnassignedVariable`) and LCV(`orderDomainValues`) are implemented in `ConstraintSatisfactionProblemr.java`. Here's my code:

```
private Integer selectUnassignedVariable(Map<Integer, Integer> partialSolution) {
        int minId = 0;
        int minSize = Integer.MAX_VALUE;
        for (Integer variable : variables.keySet()) {
            if (!partialSolution.containsKey(variable)) {
                if (variables.get(variable).size() < minSize) {
                    minSize = variables.get(variable).size();
                    minId = variable;
                }
            }
        }
        return minId;
}

private Iterable<Integer> orderDomainValues(Integer var, Map<Integer, Integer> partialSolution) {
       Map<Integer, Integer> deleteMap = new HashMap<>();
        for (Integer value : variables.get(var)) {
            int clashed = 0;
            ... find the number of values revised down by the value;
            deleteMap.put(value, clashed);
        }
        // sort by those numbers
        List<Map.Entry<Integer, Integer>> usingForSorting = new ArrayList<>(deleteMap.entrySet());
```

```
        Collections.sort(usingForSorting, new Comparator<Map.Entry<Integer, Integer>>() {
            @Override
            public int compare(Map.Entry<Integer, Integer> o1, Map.Entry<Integer, Integer> o2) {
                return o1.getValue() - o2.getValue();
            }
        });
        return ... sorted list;
}
```

2.4 Inference: FC and MAC3

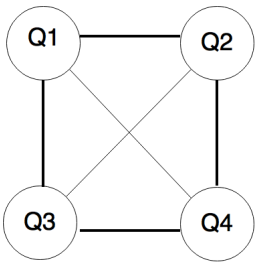
Every time we assign a value to a variable, we can apply inference function to reduce domains of its neighbors. One of the methods is forward checking(FC). We check all neighbors of this variable, delete all invalid values of their domains. The other method maintaining arc consistency(MAC3) could be more powerful. After a variable X is assigned a value, we add all arcs that are formed by X and its neighbors into the queue, then apply the inference procedure calls AC3. MAC3 will check the queue until the queue is empty while FC, as we could see, will only check the elements that were added into the queue in MAC3 initially.

FC and MAC3(both are in `inference`) are implemented in `ConstraintSatisfactionProblemr.java`. The code is too long, please see it in my java file.

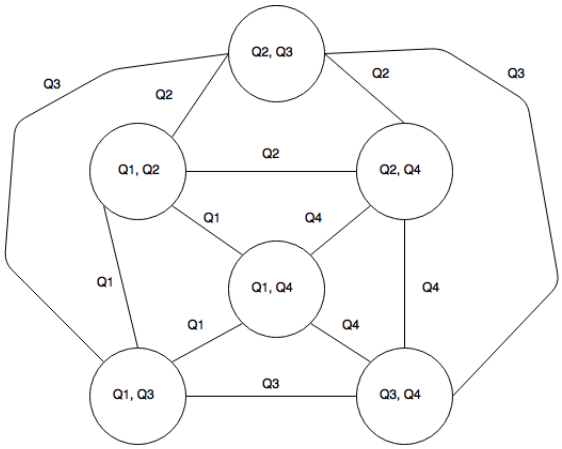
3 N-Queens Problem

3.1 Q & A

1. The constraint graph is as follows.



2. The dual of the constraint graph is as follows.



3. We can simply get the number by induction: $Num = (n-1)*(n-2)+(n-2)*(n-3)+\cdot+2*1+1$

4. Since all other solutions are rotated or mirrored by the fundamental solutions, we could simply stop searching at the half point. For example, The first queen could be assigned with 1-8, we will stop at 5 and rotate and mirror all solutions we already have to get all solutions.

3.2 Test

`Test 1`: Test with 20 Queens, the follow chart shows the time with or without every function.

`Test result of 20 Queens`

Methods	Nodes explored	Constraints checked	MRV	LCV	Time
Simple DFS	3993270	25437316	×	×	1.66
FC	145911	6708375	×	×	1.15
FC	16758	850228	✓	✓	1.00
MAC3	519260	6929094	×	×	2.11
MAC3	89700	1501257	✓	✓	0.85

`Test 2`: Test with 30 Queens.

`Test result of 30 Queens`

Methods	Nodes explored	Constraints checked	MRV	LCV	Time
Simple DFS	1692889875	Overflow	×	×	905.42
FC	43661545	Overflow	×	×	242.80
FC	1887	44038	✓	×	0.08
FC	8661533	604062523	×	✓	223.78
FC	1410098	115736084	✓	✓	41.02
MAC3	41035095	525660357	×	×	106.78
MAC3	12667791	151666759	✓	×	28.97
MAC3	39627927	506708041	×	✓	113.57
MAC3	12243570	145719285	✓	✓	31.83

We could see that FC + MRV's result is the most efficient one. I think the reason is that the constraints of this problem are not large and tricky enough so that LCV costs too much on finding the order of values, which is useless.

`Test 3`: Test with 40 - 60 Queens, FC + MRV.

`Test result of N Queens`

N	Nodes explored	Constraints checked	Time
40	3243	92554	0.06
50	6968	380117	0.13
60	8805	493139	0.20

We could see that FC + MRV is really brilliant.

4 Sudoku Problem

`Test 1`: Test with `mediumBoard`, the follow chart shows the time with or without every function.

`Test result of mediumBoard`

Methods	Nodes explored	Constraints checked	MRV	LCV	Time
Simple DFS	20897	118239	×	×	0.10
FC	17534	94320	×	×	0.12
FC	16493	36192	✓	✓	0.07
MAC3	18166	34472	×	×	0.06
MAC3	18250	35763	✓	✓	0.08

We could see that the `mediumBoard` is too easy and the improvement is not clear enough.

`Test 2`: Test with `hardBoard`.

`Test result of hardBoard`

Methods	Nodes explored	Constraints checked	MRV	LCV	Time
Simple DFS	486196028	Overflow	×	×	395.39
FC	43661545	Overflow	×	×	188.23
FC	1672015	148501435	✓	×	11.33
FC	41780596	Overflow	×	✓	362.74
FC	1613707	143285490	✓	✓	17.47
MAC3	41035095	525660357	×	×	106.78
MAC3	12667791	151666759	✓	×	28.97
MAC3	39627927	506708041	×	✓	113.57
MAC3	12243570	145719285	✓	✓	31.83

FC + MRC is still the fastest algorithm in sudoku problem. But a simple DFS + MRV is faster than DFS + FC.

5 Circuit-Board Layout Problem

The first step of this problem is building a CSP model. We have a circuit board of width n and height m. The variables of this problem is each component of width w and height h. Since the components cannot rotate in this problem, we can convert the variables for this CSP be the locations of the upper left corner of each component. Assume that the upper left corner of the board has coordinates (0, 0), the variable's location coordinates is (a, b). We can transfer coordinates (a, b) to a Integer by apply $p = a * n + b$. Then we can check if a variable's location is valid or not by apply $a + h \leq m$ && $b + w \leq n$. After we get domains of all variables, we need to find all constraints of them. The constraint of this problem is two components cannot overlap, which could be checked by their bounder. Then we can apply CSP solver and output the result.

```
// Create domains
// Notice: the upper left corner of the board has coordinates (0, 0)
for (int i = 0; i < rec[0].length; i++) {
    for (int j = 0; j < rows * cols; j++) {
        // calculate row and column of this component
        int row = j / cols, col = j % cols;
        // check overflow
        if(row + rec[0][i] <= rows && col + rec[1][i] <= cols)
            domain.add(j);
    }
}

// Create constraints
for (int i = 0; i < rec[0].length - 1; i++) {
```

```
    for (int j = i + 1; j < rec[0].length; j++) {
        Set<Pair<Integer, Integer>> constraint = new HashSet<>();
        for(Integer first : list.get(i)){
            int l = first % cols, u = first / cols,
                   r = l + rec[1][i], d = u + rec[0][i];
            for(Integer second : list.get(j)){
                int ll = second % cols, uu = second / cols,
                       rr = ll + rec[1][j], dd = uu + rec[0][j];
                int left = Math.max(l, ll), up = Math.max(u, uu),
                       right = Math.min(r, rr), down = Math.min(d, dd);
                // check overlap
                if(right <= left || down <= up)
                    constraint.add(new Pair<>(first, second));
            }
        }
        solver.addConstraint(i, j, constraint);
    }
}
```

`Test 1`: Test with a 3*10 board, 4 components{{2, 3}, {2, 5}, {3, 2}, {1,7}}

```
[c, c, ., d, d, d, d, d, d, d]
[c, c, b, b, b, b, b, a, a, a]
[c, c, b, b, b, b, b, a, a, a]
```

`Test 2`: Test with a 4*10 board, 7 components{{2, 3}, {2, 5}, {3, 2}, {1,7}, {2, 1}, {1, 7}, {1, 2}}

```
[c, c, a, a, a, b, b, b, b, b]
[c, c, a, a, a, b, b, b, b, b]
[c, c, e, f, f, f, f, f, f, f]
[g, g, e, d, d, d, d, d, d, d]
```

We could see that the CSP solver could give a solution to circuit board problem.

6 Literature Review

I read "An optimal coarse-grained arc consistency algorithm". This paper is about improvement of constraint propagation. This paper mentioned two classes of propagation algorithms for general constraints: fine-grained algorithms and coarse-grained algorithms. AC3 is a coarse-grained algorithm that once a value is removed, it will be propagated to the related variables. While fine-grained algorithms were written with a value-oriented propagation where the deletion of a value in the domain of a variable will be propagated only to the affected values in the domains of other variables. The disadvantage of coarse-grained algorithm is its worst case time complexity, AC3's worst case time complexity could be $O(ed^3)$, where e is the number of constraints and d is the size of the maximum domain. The worst case time complexity of a fine-grained algorithm could be $O(ed^2)$.This paper proposed a optimized a coarse-grained algorithm, AC2001/3.1, but it is competitive with the best fine-grained algorithms. The main idea of AC2001/3.1 is that a constraint (x_i, x_j) may be revised many times, we could saved the value's index at which we break out the loop and store the support in a structure $Last((x_i, a), x_j)$. When we need to revisit this arc, we could simply search from this index instead of starting from the start since we have already visited those values before. The results shows that this algorithm has a optimal worst case time complexity in comparison with either AC3 and some fine-grained algorithms.