Constraint Satisfaction Problem

Ping-Jung Liu October 15th 2017

1. Introduction

Constraint Satisfaction Problem(CSP) is a type of problem that could be defined with a list of variable X, a set of the domain values of each variable Xi, and a set of constraints indicating the possible variable assignments. CSP includes but not limit to map coloring problem, circuit board problem, sudoku, and 8-queens. In this project I implemented a general CSP solver using backtracking and min_conflict that can solve any CSP problem given inputs of the right format. To boost the efficiency of the solver, I implemented a few techniques like minimum remaining value(MRV), least constraining value(LCV), and maintaining arc consistency(MAC3). To test the functionality of the solver, I designed models for map coloring problem, circuit board problem, and sudoku that can translate the problems into the right format for the CSP solver.

2-1. CSP Solver--Innate Structures

The whole CSP structure is implemented in CSP.py. A CSP can be specified by three inputs. X, a list of all variables. V, a set of the domain values of each variable Xi. Ex. {X1: V11, V12, V13 X2: V21, V22}. C, a set of constraints. Ex. {(X1, X2): [(V11, V22), (V13, V21)]}; this indicates that only the combinations included in the constraints are legal assignments of values. Each individual problem model will transform its own problem statement into the three inputs.

The CSP solver also inlcudes a list of arc and a graph. Problems models do not need to generate the two because the solver will build them once it received the three inputs X, V, C. The arc list contains all directed arcs of between variables. The graph indicates the neighbors of each variable. These two are useful structures that can make the search algorithms and heuristics easier to implement.

2-2. Backtracking

Backtracking is the basic algorithms of the CSP solver. It will recursively look for good variable-value assignments and assign a value each round until the partial solution is an actual solution. The function consistent can check whether a new variable-value assignment is valid given the current partial solution. It makes sure that the new assignment does not have conflict with all the neighbors already in the partial solution. Backtracking uses this function to determine whether to continue with the assignment or not.

consistent:

```
def consistent(self, var, value, partial_sol):

    if var in self.graph:
        for neighbor in self.graph[var]:
            if neighbor in partial_sol:

                if (var, neighbor) in self.C and not (value, partial_sol[neighbor]) in self.C[(var, neighbor)]:
                    return False
                elif (neighbor, var) in self.C and not (partial_sol[neighbor], value) in self.C[(neighbor, var)]:
                    return False

    return True
```

The main idea of backtracking:

```
if self.consistent(var, value, partial_sol) and not var in partial_sol:

    partial_sol[var] = value

    # if inference returns true, continue backtracking
    if self.inference(var, value, partial_sol, removed):

        next_sol = self.backtracking(partial_sol)
        if not next_sol == None:
            return next_sol
```

Ignoring inference for now, we can see that the function tries to find a new partial_sol each time after ensuring the assignment was consistent and will only return the new solution if it is not None.

2-3. Minimum Remaining Values(MRV)

Backtracking will assign a value to an uassigned variable each round. MRV finds the variable with the smallest domain size, so it is more likely to cause failure, thereby pruning the tree. The performance of MRV depends greatly on the problem though, since it is only a heuristic.

MRV:

```
# select unassigned variable
def suv(self, partial_sol):

    min_size = inf

    # look for the variable with least domain size
    for var in self.X:
        if not var in partial_sol:
            if self.MRV:
                if len(self.V[var]) < min_size:
                    min_size = len(self.V[var])
                    mrv_var = var
            else:
                return var

    return mrv_var
```

2-4. Least Constraining Value(LCV)

After selecting a variable, LCV will order the domain so that the value with less conflicts with others come first, to increase the chance of success. Notice for MRV we want fail first to prune the tree, but for LCV we want to assign the correct value as fast as possible.

LCV:

```
rank = []

for val in self.V[var]:
    count = 0

    if var in self.graph:
        for neighbor in self.graph[var]:
            for n_val in self.V[neighbor]:
                if (neighbor, var) in self.C and not (n_val, val) in self.C[(neighbor, var)]:
                    count = count + 1
                elif (var, neighbor) in self.C and not (val, n_val) in self.C[(var, neighbor)]:
                    count = count + 1

        rank.append((count, val))

rank.sort()

result = []
for i in range(0, len(rank)):
    result.append(rank[i][1])
return result
```

2-5. Inference and MAC3

An inference infer consequences from a given assignment. For example, the regualr inference will set the domain of an assigned variable to only the assignment value, instead of the whole value domain at the beginning of search. The deleted values will be stored in another set so that if the assignment turns out to be incorrect, the program can then restore the deleted values to V, the value domain.

Regular Inference:

```
# add related values to removed
if var in removed:
    removed[var] = list(set(removed[var] + self.V[var]))
```

```
else:
    removed[var] = self.V[var]

# remove all other value except the guess value
self.V[var] = [value]

if not self.MAC:
    return True
```

Maintain Arc Consistency(MAC3) is an inference technique that can maintain consistency throughout the search process. After each assignment, MAC3 can make sure no value of other variable can have conflict with the changed domain. MAC3 makes use of the function AC3, which checks the consistency of a arc queue and constantly revise the domains of related variable to ensure the whole graph is consistent. The function revise deletes conflicted values in the domains of one single arc.

MAC3:

```
# perform ac3 on all arc(j, i)
arc_que = deque()
for arc in self.arc:
    if not arc[0] in partial_sol and arc[1] == var:
        arc_que.append(arc)

return self.ac3(arc_que, removed, True)
```

AC3:

```
def ac3(self, arc_que, removed, infer):

    while len(arc_que) > 0:
        arc = arc_que.pop()

        # if revised, add all related arc to the que
        if self.revise(arc, removed, infer):

            if len(self.V[arc[0]]) == 0:
                return False
            elif arc[0] in self.graph:
                for neighbor in self.graph[arc[0]]:
                    if not neighbor == arc[1]:
                        arc_que.append((neighbor, arc[0]))
    return True
```

The code for revise is straightforward but tedious, so I will not show it here. The function basically deletes bad values from related domains and saves them in removed for future usuages.

3. Map Coloring Problem

The map coloring problem is implemented in mapCSP.py. The objective of the traditional map coloring problem is to assign each territory a color so that neighboring territories have have the same color. For example. Given to neighboring territories A and B and two colors Red and Blue. The variable list X of this problem will be [A, B]. The inital value domain of this problem will be {A: [Red, Blue], B: [Red, Blue]}. The constraints of this problem will be {(A, B): [(Red, Blue), (Blue, Red)]}. This is one of the simplist CSP and this was implemented to show the functionality of backtracking.

Test Case 1:
Territories: ["A", "B", "C", "D"]
Colors: [1, 2] Neighbors: {"A":["B"], "B":["A", "C", "D"], "C":["B"], "D":["B"]}
Results:

```
Nodes Visited:
5
Try:
5
Solution:
{'A': 1, 'B': 2, 'C': 1, 'D': 1}
```

"Try" indicates the number of times backtracking tried a new value on a variable.

Test Case 2 (Autralian Map Problem): Territories: ["WA", "NT", "Q", "SA", "NSW", "V", "T"]
Colors: [1, 2, 3]
Neighbors: {"WA":["NT", "SA"], "NT":["WA", "SA", "Q"], "SA":["WA", "NT", "Q", "NSW", "V"], "Q":["NT", "SA", "NSW"], "NSW":["Q", "SA", "V"], "V":["NSW", "SA"], "T":[]}
Results:

```
Nodes Visited:
8
Try:
11
Solution:
{'NSW': 2, 'WA': 1, 'T': 1, 'V': 1, 'Q': 1, 'NT': 2, 'SA': 3}
```

4. Circuit Board Problem

The circuit board problem is implemented in circuitCSP.py. Provided a rectangular circuit board and numerous rectangular components, the goal of the program is to put every component onto the board without overlap. The variables are the components, and the values are the possible lower left locations of these components.

Given a board of width n and height m, and a component of width w and height h, the domain of the component (w, h) will then be [(0, 0), (1, 0)...(n-w, 0), (0, 1)...(n-w, 1)...(n-w, m-h)]

Given a board of width 10 and height 3, and two components of (width, height) = (3, 2) and (5, 2), the constraints of them will be:

```
[((0, 0), (3, 0)), ((0, 0), (3, 1)), ((0, 0), (4, 0)), ((0, 0), (4, 1)), ((0, 0), (5, 0)), ((0, 0), (5, 1)),
 ((0, 1), (3, 0)), ((0, 1), (3, 1)), ((0, 1), (4, 0)), ((0, 1), (4, 1)), ((0, 1), (5, 0)), ((0, 1), (5, 1)),
((1, 0), (4, 0)), ((1, 0), (4, 1)), ((1, 0), (5, 0)), ((1, 0), (5, 1)), ((1, 1), (4, 0)), ((1, 1), (4, 1)),
((1, 1), (5, 0)), ((1, 1), (5, 1)), ((2, 0), (5, 0)), ((2, 0), (5, 1)), ((2, 1), (5, 0)), ((2, 1), (5, 1)),
 ((5, 0), (0, 0)), ((5, 0), (0, 1)), ((5, 1), (0, 0)), ((5, 1), (0, 1)), ((6, 0), (0, 0)), ((6, 0), (0, 1)),
 ((6, 0), (1, 0)), ((6, 0), (1, 1)), ((6, 1), (0, 0)), ((6, 1), (0, 1)), ((6, 1), (1, 0)), ((6, 1), (1, 1)),
 ((7, 0), (0, 0)), ((7, 0), (0, 1)), ((7, 0), (1, 0)), ((7, 0), (1, 1)), ((7, 0), (2, 0)), ((7, 0), (2, 1)),
 ((7, 1), (0, 0)), ((7, 1), (0, 1)), ((7, 1), (1, 0)), ((7, 1), (1, 1)), ((7, 1), (2, 0)), ((7, 1), (2, 1))]
```

In the model, the problem will transform the components into integer ids for easier identifications, which means components input (3, 2) (5, 2) (4, 4) will be 0, 1, 2. It will then try out all the locations on the board to obtain the value domain of each component. As for the constraints, I simply saved all the possible location combinations of two components into the constraint set. For example given a 4 * 1 board and two 2 * 1 components, the constraints will be {(0, 1): [((0, 0), (2, 0)), ((2, 0), (0, 0))]}

Test Case 1 Board: 10 * 3 Components: (3, 2), (5, 2), (7, 1), (2, 3):

```
Nodes Visited:
5
Try:
20
Solution:
{(2, 3): (8, 0), (3, 2): (0, 0), (5, 2): (3, 0), (7, 1): (0, 2)}
2222222#33
0001111133
0001111133
```

Test Case 2 Board 10 * 10 Components: ((4, 4), (5, 5), (3, 3), (2, 4), (4, 2), (6, 1), (4, 1), (1, 9), (2, 5), (2, 2)

```
Nodes Visited:
11
Try:
342
Solution:
{(2, 5): (7, 0), (5, 5): (0, 4), (1, 9): (9, 0), (2, 2): (7, 5), (4, 2): (5, 7), (6, 1): (0, 9), (4, 4): (0, 0), (4, 1): (6, 9), (
5555556666
1111144447
```

```
1111144447
1111133997
1111133997
1111133887
0000#33887
0000222887
0000222887
0000222887
```

5. Sudoku (Extra Credit)

The Sudoku Problem is implemented in sudokuCSP.py. I choose to introduce this extra credit first because MRV tremendously helps the search speed of Sudoku and I would like to use this as the example in the testing section. The rules of Sudoku are simple. In a square consists of nine 3 * 3 squares, the numbers in each row, column, and 3 * 3 square must all be different. The variables are each square, assigned and unassigned. Value domain for assigned square is simply the number, whereas that for unassigned square is 1 through 9. The constraints make sure each row, column, and 3 * 3 square has all 1 to 9.

Test Case 1:
Results:

```
Nodes Visited:
82
Try:
81
Solution:
534678912
672195348
198342567
859761423
426853791
713924856
961537284
287419635
345286179
```

Test Case 2:

Results:

```
Nodes Visited:
82
Try:
81
Solution:
761349825
382751694
594286731
625973418
837614952
419528376
943165287
258497163
176832549
```

6-1. Testing--MRV

First, to demonstrate MRV is implemented correctly, I used a 5 * 5 circuit board and three components in the exact order (2, 2) (3, 3) (1, 1); the id of them are respectively 0, 1, 2. Since the value domain for this problem indicates all possible locations of each component, the domain size of (3, 3) must be the smallest. Without MRV, the select unassigned variable function returned 0 1 2, as expected. With MRV, the function returned 1 0 2, which was exactly what MRV should be doing.

Second, to show the power of MRV, I used the Sudoku problem as test cases. For both the provided test cases above, the program ran each of them for more than three minutes but still could not find the answers (I was impatient so I stopped the program). But with only MRV enabled, the program could solve both test cases in less than 1s, which showed an incredible boost to search speed.

6-2. Testing--LCV

LCV helps the search speed in the circuit board problem. Given a 10 * 5 board and components (3, 2) (5, 2) (2, 3) (1, 1) (2, 1) (4, 1), the result of a search without all heuristics and inference as follow:

```
Nodes Visited:
7
Try:
58
Solution:
{(3, 2): (0, 0), (2, 3): (5, 0), (5, 2): (0, 2), (4, 1): (0, 4), (1, 1): (3, 1), (2, 1): (3, 0)}
3333######
11111#####
1111122###
0005#22###
0004422###
```

Result of a search with only LCV turned on:

```
Nodes Visited:
7
Try:
6
Solution:
{(3, 2): (0, 0), (2, 3): (8, 0), (5, 2): (0, 3), (4, 1): (0, 2), (1, 1): (3, 1), (2, 1): (3, 0)}
11111#####
11111#####
3333####22
0005####22
00044###22
```

Though in this case MRV did not do much when tested, LCV obviously demonstrated its potency.

6-3. Testing--MAC3

I did not think of a simple way to show MAC3 was implemented correctly, but its functionality is so profound that everytime I turned it on the program instantly solved any problem. I will again use Sudoku to show:

Sudoku test case 1 with only MRV:

```
Nodes Visited:
82
Try:
81
Solution:
534678912
672195348
198342567
859761423
426853791
713924856
961537284
287419635
345286179
```

Sudoku test case 2 with only MRV:

```
Nodes Visited:
82
Try:
```

```
81
Solution:
761349825
382751694
594286731
625973418
837614952
419528376
943165287
258497163
176832549
```

7 Non-rectangular Components for Circuit Board (Extra Credit)

This modified version is implemented in new_circuitCSP.py. I modified the circuit board problem so that the components do not have to be rectangles. They could be any shape that can be formed with one rectangle on top of the other. This includes all kinds of L-shape and T-shape components and can create a wide range of variations. The variable list consists of the components' specifications.

For Ex. (1, 3, 5, 1, -2) indicates that the bottom component is 1 * 3, top component is 5 * 1, the shift between the two is -2. This will look like the following component:

00000
$$0$$
$$0$$
$$0$$

Another Ex. (4, 2, 1, 2, 1):

$0$$$
$0$$$
0000$
0000$

The value domain contains the possible locations of the left bottom corner of the bottom component. The constraints, just like the original problem, ensures no components overlap.

Test Case 1 Board 5 * 5 Components (3, 1, 1, 4, 0), (2, 1, 2, 2, -1), (2, 2, 1, 3, 1)

Results:

```
Nodes Visited:
4
Try:
3
011#2
011#2
0#112
0##22
00022
```

Test Case 2 Board 10 * 10 Components (2, 1, 1, 2, 1), (10, 1, 4, 5, 6), (3, 3, 4, 2, 0), (6, 1, 3, 1, 3), (3, 2, 2, 2, 1), (4, 3, 5, 1, 0)

Result:

```
Nodes Visited:
7
Try:
26
2222#55555
2222#5555#
222##5555#
222445555#
22244#1111
#0444#1111
#0444#1111
```

```
00#3331111
3333331111
1111111111
```

8. Strong Arc Consistency (Potential Extra Credit)

Before the program start backtracking, the value domain usually contain numerous conflicts. I wrote a few codes to perform AC3 on all arcs in the problem to reduce domain size before the search begin.

```
if self.Enforced_AC:
    first_ac3 = True
    arc_que = deque()
    for arc in self.arc:
        arc_que.append(arc)
    first_ac3 = self.ac3(arc_que, {}, False)
    if not first_ac3:
        return None
```

While this modification provided moderate boost for other problems, it can helped solve a hard Sudoku with 81 tries without MRV, LCV, and MAC3, which was the same performance as enabling all heuristics and inference.

9. Min-Conflict Solver (Extra Credit)

The min conflict solver is also implemented in CSP.py. The main idea of min-conflict is to initialize the solution with random assignments or greedy search. Each step the program will modify the assignment of one variable so that conflicts decrease. I would be easier to explain with the codes:

```
0 def min_conflicts(self, max_steps):
1
2   partial_sol = {}

3   # can either randomly select initial solution or use greedy method
4   for var in self.X:

5       partial_sol[var] = self.best_val(var, partial_sol)
6       #partial_sol[var] = self.V[var][0]

7   for i in range(0, max_steps):

8       if self.is_solution(partial_sol):
9           print("steps:")
10          print(i)
11          return partial_sol

12      # randomly select a conflicted variable
13      possible_var = []

14      for var in self.X:
15          if self.conflicts(var, partial_sol[var], partial_sol) > 0:
16              possible_var.append(var)

17      next_var = random.choice(possible_var)
18      # modify solution
19      partial_sol[next_var] = self.best_val(next_var, partial_sol)

20  return None
```

Line 5 and 6 are the initialization of solutions, greedy and random respectively. The program will then iterate for max steps times until it admitted failure or found a solution, in line 8-11. self.best_val returns the value with least conflicts. Each round the program will assign a value to another randomly selected conflicted variable in line 19.

I tested its functionality with map coloring and circuit boards.

Test Case 1 Autralian Map Problem:

```
Time Spent:
0:00:00.046823
Solution:
{'NT': 2, 'SA': 3, 'V': 1, 'Q': 1, 'T': 1, 'WA': 1, 'NSW': 2}
```

Test Case 2 Board 10 * 10 Components (4, 4), (5, 5), (3, 3), (2, 4), (4, 2), (6, 1), (4, 1), (1, 9), (2, 5), (2, 2)

```
Time Spent:
0:00:00.282202
Solution:
{(2, 5): (7, 0), (5, 5): (0, 4), (1, 9): (9, 0), (2, 2): (7, 5), (4, 2): (5, 7), (6, 1): (0, 9), (4, 4): (0, 0), (4, 1): (6, 9), (
5555556666
1111144447
1111144447
1111133997
1111133997
1111133887
0000#33887
0000222887
0000222887
0000222887
```