Report

COSC276, Fall2021, PA4, Xuedan Zou

A.Description

This time we implemented a general constraint search problem (CSP) model with the heuristics and inference improvement and used it to solve the map coloring problem together with the circuit-board layout problem.

First, we should build a general framework for our CSP model. We have variables, domain of each variables and the constraints between variables in a CSP. Since we want to make our general framework works on all of the problems, regardless of the original input of the given problem is string, integer or anything else, we decide to store our variables as string and domain as actual numerical value. We count each variable with its index and use this index to caculate the specific variable during the whold searching process, so the variable **list** is just used for mapping the variable from the index to the string. We then build a **dictionary** with the index of each variable as the key to express our domains. For the domain of each variable, it is heavily based on the specific given problem, but we have to make clear that it is **a list combined with multiple independent numerical values**. Finally, we give a neighbor list as a dictionary which shows the constraint relationship for each variable. We also write constraint functions based on specific problem to translate the constraint condition to our model. We have two kinds of constraint function here:

```
def constraints(self, variable, value):
```

which is passed in a variable with its value, the function then make the judgement whether this variable with its new value satisfies all of the constraint rules in our model.

```
 def pair_constraint(self, x1, value_x1, x2, value_x2):
```

which is passed in two variables with their values, the function then judge whether these two variables with given values satisfy their constraint rule.

We then need to implement the searching process: **BackTracking Search**. BackTracking Search explores an unexplored variable each time, tries to fit this variable with its possible domain values that satisfy the constraints. It repeats this process iteratively until all variables have been explored. During this searching process, we use a **list** named *assignment* to store our explored results. The index of the assignment list referes to the corresponded index of variables. We initially set all of values in *assignment* be -1, which means unexplored. Since the possible domains of variables will be changed with our searching process going on, we build a *domain_remain* table (**dictionary**) to update the possible remaining domain of each variable in real-time. Notice our function that updates this table:

```
 # this function is used to update the domain_remain table when there is a variable been
explored
    def update_domain_remain(self, next, value):
        self.domain_remain[next] = [value]
        # need to update the neighbor's remain domains in the table
        neighborlist = self.neighbors[next]
        remove = 0
        for neighbor in neighborlist:
            # if this neighbor has been explored before
            if self.assignment[neighbor] != -1:
                continue
            # for the unexplored neighbors, check if their remaining values are legal
            neighbor_remain = self.domain_remain[neighbor]
            for value in neighbor_remain:
                if self.constraints(neighbor, value) == False:
                    self.domain_remain[neighbor].remove(value)
                    remove += 1
        return remove
```

We passes in *next* (updated variable) with its chosen value to this function. It will then check all the unexplored neighbors of this updated variable and see if their remaining values are legal. We then remove all the illegal values of those neighbors and updates our table. Note that we can at the same time calculate how many values in total are removed and this can be used in our later LCV implementation.

Next, we implement the heuristic method to order the unexplored variables (MRV, degree heuristic) and order the legal values of a given variable (LCV). MRV heuristic always chooses the variable with the fewest legal values to explore first. Since we use the *domain_remain table* to update the legal values for each variable in real-time, it is no hard for us to check that table and order the variables based on their remaining legal values. Compared, degree heuristic always chooses the variable with the most constraints on remaining variables. Since we have our *neighbor list*, we quickly visit that and order those variables based on their number of neighbors. Lastly, LCV heuristic always tires the value that rules out the fewest values in the remaining variables first. As we have discussed before, our *update_domain_remain* function counts the number of removed values for the remaining neighbor variables, we can try the specific variable with each of its legal value and order those values with the removed times.

Last, we add AC-3 as our inference technique. AC-3 propagates the change of the legal domain value and add those unexplored or needed reexplored pairs into a **queue**. The algorithem keeps picking out pairs to explore from this queue and add new pair to the queue when the remaining legal domain of a specific variable is getting smaller. For each given pair, we use the following functionto check if these two variables are under constraints with the combinations of their given legal values:

```
# the assist function to AC-3, return if there is a change to remain_domain
    def Remove_Inconsistent_Values(self, x1, x2):
        # if there is any pair of values from x1 and x2 satisfies the constraint between
x1 and x2
        found = False
        changed = False
        for value_x1 in self.domain_remain[x1]:
            for value_x2 in self.domain_remain[x2]:
                if self.pair_constraint(x1, value_x1, x2, value_x2) == True:
                    found = True

            if found == False:
                # no value in x2 can make a pair with this x1 value, so delete it
                self.domain_remain[x1].remove(value_x1)
                changed = True

        return changed
```

B. Evaluation

We extend our general CSP framework to two specific problems: map coloring and circuit board layout by overwriting part of the functions including the way to build our *domains* dictionary, the way to *print out the solution* and the *constraints* etc. The constraint for the map coloring problem is easy: the value of each variable shoud be different from its neighbor's value. The constraint for the circuit board layout problem is not hard either: all of the variables (rectangle) should not be overlapped. The result shows that our model works fine and solve these two problems successfully.

We test both of these two problems with and without heuristic, with and without inference. Still, the heurstic and inference only improves the speed of our searching process but by default without these improvements, we can still get the same answer. Note that for the same problem, the solution might be different with different combinations of heuristic and inference methods, but they are all the optimal.

C. Discussion

1. Map Coloring Test

For our map coloring test, we get the same result regardless of which heuristic and which inference mehod we used. We then test the time used for each mehod:

```
"default", "default", "default"
```

```
running time: 0.0005061626434326172 seconds

"MRV", "default", "default"
running time: 0.0003190040588378906 seconds

"degree heuristic", "default", "default"
running time: 0.000492095947265625 seconds

"default", "LCV", "default"
running time: 0.001062154769897461 seconds

"default", "default", "AC-3"
running time: 0.001348733901977539 seconds

"MRV", "LCV", "AC-3"
running time: 0.0018579959869384766 seconds

"degree heuristic", "LCV", "AC-3"
running time: 0.0016858577728271484 seconds

"degree heuristic", "LCV", "default"
running time: 0.000675201416015625 seconds
```

We can find out that if we use all of our heuristic and inference methods at the same time then the running time is supposed to be the longest. Compared, if we dont use any heuristic and inference method, then the running time is about the least. AC-3 inference method obviously increases the running time of our model since its complex propagating process. The degree heuristic and MRV heuristic here imporve the efficiency of our model slightly. However, the LCV method costs longer time either. Considering both the LCV and the AC-3 require to check the constraints of the given node with its neighbors, obviously this checking process spends time a lot and if the problem itself is not complex enough then there is no such great improvement compared with the cost of these methods themselves. Here the Austrilia coloring problem is kind of simple (at least not complex enough) so it takes longer time than the original one in terms of running time.

We then test our map coloring model with a much more complex problem: coloring the map of the states, with four colors solution, the running time with and without heuristics and inferencen are listed below:

```
"default", "default", "default"
running time: 715.921788930893 seconds

"MRV", "default", "default"
running time: 0.013315677642822266 seconds

"MRV", "LCV", "AC-3"
running time: 0.13325214385986328 seconds
```

```
"default", "default", "AC-3"
running time: 0.8524959087371826 seconds

"degree heuristic", "LCV", "AC-3"
running time: 5.161167144775391 seconds
```

Now we can easily notice the improvment by our heuristics and inference methods. To bg specific, with the MRV we can try out the solution luckily with a very fast speed, and with LCV and AC-3 we can significantly improve the efficiency.

2. Circuit Board

Firstly, the initial domain to this circuit board problem can be restricted as: *x+w<=n, y+h<=m*, where x and y refer to the left bottom corordinates of our component, the width and height of the component are w and h and the width and height of the board is n and m. And of course, *x>=0, y>=0*. We can then transfer our x and y values into the index value *x+(y-1)n* and store in our model.

Secondly, the constraint that enforces the fact that the two components may not overlap is in fact the way to make two rectangles not overlap. Consider the opposite situation: when the two rectangles overlap. We can then get the following judgement:

```
 if (x1_right < x2_left or x1_down > x2_up) or (x2_right < x1_left or x2_down > x1_up):
```

Where x1_right, x1_down, x1_left and x1_up refers to the most right, down, left and up coordinate value for rectangle x1 and so is similar to rectangle x2. Given to the 10x3 board with components a and b we can give out all the legal pairs of locations below where the first coordinate pair refers to the left bottom corner coordinate of component a and the second refers to that of component b:

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

Thirdly, the conversion betweent the coordinate and the index is vital to this circuit board problem. The idea here is simple: we count the index from left to right and from bottom to top. Take the 10x3 board for instance, then the index of each point that is:

```
90, 91, .......   .......    99
...     .......   .......   ...
10, 11, .......   .......    19
0,   1, 2, 3, 4, 5, 6, 7, 8,  9
```

Then we can write up two convert funciton to make the conversion between the index and the corrdinate:

```
    # to convert the coordinate to the index
    def cor_to_index(self, x, y):
        index = self.width * y + x
        return index

    # to convert the index to the coordinate
    def index_to_cor(self, index):
        x = index % self.width
        y = int((index - x)/self.width)
        cor = (x, y)
        return cor
```