CS76 Assignment 4: Constraint Satisfaction Problem

Himadri Narasimhamurthy | 1/31/19

Introduction

The constraint satisfaction problem is potentially one of the most useful mechanisms to solve problems of artificial intelligence and can be used in almost any engine where you would be assigning values to variables along some particular set of rules. I went about designing my CSP in the manner described in design notes - first I created a CSP class which can be applied to any CSP, then I applied it to Australia Map and Circuit Board.

CSP Class

First was the simple CSP class which I initialized as such - with the variables, domain, and constraints. I also included a failure indicator which can be updated on the event no solution is found and a self.checking dictionary which keeps track of where we are in the search and helps with inference.

```
def __init__(self, vars, domain, cons):
    self.vars = vars
    self.domain = domain
    self.cons = cons

    #the domain of things being checked
    self.checking = None

    #whether or not we found solution
    self.failure = False
```

I also defined several helper functions throughout this class to do simple tasks like performing and removing assignments, checking constraints, and returning the list of next to be checked.

Backtracking

Backtracking is the main search element in CSP and is done through a recursive check of the elements and whether or not they satisfy constraints. This process is aided in speed and efficiency by heuristics and inference (discussed later).

The backtracking is done recursively so it's called by other applications of the CSP with an empty assignments dictionary {} and the heuristics which we would want to use. When I designed CSP, I allowed the user to input the heuristic or inference (or none) which they would like to use when calling backtrack - this allows for heuristics to be run seperately and together.

The work is done in **rec_backtracking** which I will go through and explain by line:

We exit recursion when we have assigned a value to all variables - if we have not yet assigned all variables, then we go through the values in the val_list of whichever variable we want to explore - we can either get the first item of our lists or we can apply heuristics here.

```
def rec_backtrack(self, assignments, select, order, inference):
    .
    .
    .

    #if we have assigned all variables return
    if len(assignments)== len(self.vars):
        return assignments

    #choose a var and val list based on heuristic
    var = select(assignments)
    val_list = order(var, assignments)
    .
    .
    .
```

We exit recursion when we have assigned a value to all variables - if we have not yet assigned all variables, then we go through the values in the val_list of whichever variable we want to explore - we can either get the first item of our lists or we can apply heuristics here. We then go through all items in our value list and check if they are constrained using **check constraint** as a helper function - if so, we assign the value to the variable. We also need to keep track of where we are checking:

```
            .
            .
            .
            if self.checking is None:
                self.checking = {}
                #add all from domains list to checking for the vertex in question
                for v in range(len(self.vars)):
                    self.checking[v] = list(self.domain[v])

            #create a removed list of all that we have checked
            for checked in self.checking[var]:
                if checked != val:
                    rem.append((var, checked))

            #add the current to checking list
            self.checking[var] = [val]
            .
            .
            .
```

This is where self.checking comes in handy - if we are just starting our search, add all possible domain values to the checking dictionary for each variable. Also I made sure to create a removed list which would be key to inference here when we did a preliminary prune of what would be checked. Then we add inference and do the recursion - saving our final assignment if we find one and saving None if there is no possible solution with our constraints.

Minimum Remaining Variables Heuristic (MRV)

The first area where we can utilize a heuristic is in our selection of variable to explore before we enter the loop - we would want to get the variable that has the minimum number of possible items to explore - which shortens our search and creates a quick constraint.

I created this heuristic **mrv** which does just that and keeps track of the count for number of variables left to be checked or number left in the domain and replaces the minimum value if found. We then return the lowest value once we exit the for loop of all possible variables to explore.

The testing for MRV is clear when running it on the AusMap class since we can see that Tasmania has the fewest neighbors and therefore would have the fewest variables remaining - therefore, when the results are displayed, they start from Tasmania rather than at the beginning of the list with South Australia.

```
--------------------- SOLVE WITH MRV ------------------
        Tasmania maps to G
        Victoria maps to B
        Western_Australia maps to B
        South_Australia maps to R
        Northern_Territory maps to G
        New_South_Wales maps to G
        Queensland maps to B
```

Least Constraining Values Heuristic (LCV)

The second heuristic we can employ is when we are going to search all possible values to explore for any given variable. In this heuristic, we return an ordered set of values from least constraining to most constraining.

This is performed in a similar manner in **lcv** where we increment count every time we get to a state which is constrained (allowed), we want to get the values in an order where we have fewer choices to most choices. So after saving in a list each value and its count in a tuple, we sort based on count from least to greatest. Then add the values to a list without counts and return that

```
    .
    .
    .
    # remaining contains values and their constraint counts in tuple
        remaining.append((v, c))

    #sorted from least to greatest constraint count
    remaining.sort(key=lambda count: count[1])
    .
    .
    .
```

MAC-3 Inference

MAC-3 inference helps us decrease the size of our search if we can guarantee that we won't find a possible solution down that wing that we are searching. If we are not using inference, simply return True on the if condition to enter recursion and recurse in every case. It also helps decrease the size of our domain so that as recursion continues, it gets quicker.

First we create a queue of all possible variables and if the variable/possible variable pair is impossible on first check, we add it to the queue.

```
#mac_3 inference
def mac_3(self, var, assignments, rem):
.
.
.

    #go through all vars- checking for impossibles
    for neighbor in range(len(self.vars)):
        if neighbor == var or (len(self.cons[(var, neighbor)])<=0):
            if neighbor not in assignments:
                q.append((var, neighbor))
   .
   .
   .
```

Then, we go through the queue getting each variable/neighbor pair and check, via **removed** helper function whether we need to remove any pair from inference - then if we have gotten rid of all impossible pairs and there is no need to recurse on that path, we return False and then continue checking the next variable against the previous and add it to the queue. If we still have path options, then return True and enter recursion.

```
    .
    .
    .
    #while q not empty
    while q:
        (a, b) = q.pop()

        #check if we need to remove
        if self.removed(a, b, rem, assignments):
            #if we have removed all from the domain - all impossible
            if len(self.domain[a] == 0):
                return False

        .
        .
        .
```

AusMap Class

I used the CSP general class to run on the Australia Map coloring problem which colors the sections of a map of australia ensuring that no neighboring sections have the same color.

When the class in initialized, it should take in a list of strings for variables, a list of strings for colors, and some type of list we can parse in the form "state: neighbor neighbor;". The initialized constructor is shown below:

```
australia = AusMap(['South_Australia', 'New_South_Wales', 'Northern_Territory', 'Quee
nsland', 'Western_Australia', 'Victoria', 'Tasmania'],
                    ['G', 'B', 'R'],
                       'South_Australia: Western_Australia Northern_Territory Queensl
and New_South_Wales Victoria; Northern_Territory: Western_Australia Queensland; New_S
outh_Wales: Queensland Victoria; Tasmania: Victoria')
```

The code for parsing these strings was simple as we did not have complicated constraints - we passed variables in as is. Then for the domain, I simply added the color list (replaced with indices for colors) as the value to every key (index in variable list).

Then for constraints, I loop through each variable pair in the variable list and check whether they are neighbors - which required parsing my neighbors string into a dictionary with key state and value neighbor list. If so, then I add constraints of possible color pairs (any non-equal color pair).

When testing this class with no heuristics or inference - I was able to get a possible answer in very little time - probably due to the fact that we are dealing in a small domain here.

```
----------- SOLVE AUSMAP WITH NO HEURISTICS OR INFERENCE ----------

    South_Australia maps to G
    New_South_Wales maps to B
    Northern_Territory maps to B
    Queensland maps to R
    Western_Australia maps to R
    Victoria maps to R
    Tasmania maps to G
```

CircBoard Class

The CircBoard class applies our general CSP for the problem of fitting pieces onto a circuit board of fixed size. When designing this, I had to make decisions about how to input piece list and board and I decided on the input below - where pieces are in tuples (ASCII char, width, height) and the board is simply inputted in dimensionally (width, height):

```
    pieces1 = [('a', 3, 2), ('b', 5, 2), ('c', 2, 3), ('e', 7, 1)]
    cb = CircuitBoard(pieces1, 10, 3)
```

Now the work of parsing domain and creating constraints was more difficult in circboard than for ausmap.

The domain of a variable corresponding to a component of width w and height h, on a circuit board of width n and height m. Make sure the component fits completely on the board.

The domain of each piece (a piece is one variable) is all possible x,y locations that it could fit on the board irrespective of other pieces - only take into account board and piece size.

```
    x = self.boardx - (self.pieces[p][1])
    y = self.boardy - (self.pieces[p][2])
```

Consider components a and b above, on a 10x3 board. Write the constraint that enforces the fact that the two components may not overlap. Write out legal pairs of locations explicitly.

We then consider pieces a and b where a is 3 by 2 and b is 5 by 2. The constraint that enforces the fact the components cannot overlap is as follows:

```
    #get possible domains for both pieces
   for d1 in self.doms[i]:
         for d2 in self.doms[j]:
              #maximum and minimum of piece width/height and piece location
              high_x = max((d1[0] + p1[1]), (d2[0] + p2[1]))
              low_x =  min(d1[0], d2[0])

              high_y = max((d1[1] + p1[2]), (d2[1] + p2[2]))
              low_y = min(d1[1], d2[1])

              #if constraints are within realms of the board dimensions
              if high_x - low_x >= p1[1] + p2[1] or high_y - low_y >= p1[2] + p2[2]:
                    if high_x<= self.boardx and high_y <= self.boardy:
                          #add the possible locations
                          constraints[(i, j)].append((d1, d2))
```

This gets the maximum possible values of each domain value + piece size and minimum value of domain (since we draw from bottom left corner) and saves them as lowest and highest possible locations. Then we check them against whether two pieces can fit top to bottom or side to side and then finally we check that the pieces are not off the board. If all these conditions apply, that pair is added to constraints.

Therefore for pieces a and b we have the following legal pairs of locations:

```
    (0,0) (3,0)         (0,0) (5,1)
    (0,1) (3,1)         (1,0) (5,1)
    (1,0) (4,0)         (1,1) (5,1)
    (1,1) (4,1)         (0,0) (4,0)
    (0,0) (4,1)
    (1,0) (4,1)         (2,0) (5,1)
    (1,1) (4,0)         (2,1) (5,1)
    (0,0) (5,0)         (0,1) (5,0)
    (1,0) (5,0)         (2,0) (5,0)
    (1,1) (5,0)         (2,1) (5,0)
```

Describe how your code converts constraints, etc, to integer values for use by the generic CSP solver.

My code converts constraints to integer values in the above code snippet where each constraint has possible domain locations for two pieces saved in a dictionary. As we only go through the indexes of lists and not through the objects in the lists, we are able to avoid problems of a lack of generality.