COSC 76: Constraint Satisfaction Problem Report

Paolo Takagi-Atilano

Introduction

This report will detail how I implemented a generalized solver for Constraint Satisfaction Problems (CSPs), and then used it to solve multiple distinct types of problems, such as Map Coloring and Circuit Board problems.

CSPs have a couple different compontents. First there are variables, which end up holding are different states in the problem. Second there domains, which are the different states which the variables can be. Last there are constraints. A constraint can be described as, when given two variables, the set of all pairs of values that those variables could be. In the end, a solution to a CSP is one in which every variable is assigned to a value in the domain, and for which all the constraints are satisfied.

Since there we know that there is a limited number of variables that are members of the solution, the tree of all partial and complete solutions is of limited depth, corresponding to the number of variables in that particular CSP.

Design:

`ConstraintSatisfactionProblem.py`:

data:

`fails` - number of fails for backtrack search.

`assignment_length` - number of variables.

`assignment` - list of assignments, initialized to be of length `assignment_length`, with every element being `None`.

`domain_length` - number of values in the domain. Note: maximum domain is assumed, if some variables do not have that domain, they are not in the `constraints` hashtable, and such are not tested.

`constraints` - `Constraint` object for given (parameterized) `constraints` hashtable.

functions:

`backtrack_search(self, mrv, lcv, infer)` - runs setup, then starts recursive backtrack search.

`backtrack_search_helper(self, mrv, lcv infer, inconsistencies)` - recursive backtrack search, returns None if no solution, or `assignment` list of given `ConstraintSatisfactionproblem` object when solution reached. `mrv`, `lcv`, `infer` are booleans indicating whether to use MRV heuristic, LCV heuristic, and MAC-3 inference. `inconsistencies` are values not to include in domain from inference at previous recursive depth, empty dictionary if `infer == False`.

`is_complete(self)` - checks to see if current assignment is complete.

`no_variable_heuristic(self)` - for when MRV heuritic is not used, returns next non-null index in assignment list.

`mrv_heuristic(self)` - MRV heuristic, see below.

`lcv_heuristic(self, var)` - LCV heuristic, see below.

`mac_infer(self, var)` - MAC-3 inference, see below.

`mac_revise(self, arc, inconsistencies)` - Used for MAC-3 inference, determiens if an arc is an inconsistency or not.

`Constraint.py`:

data:

`self.constraints` - constraints hashtable where variable tuples map to set of value tuples.

functions:

`is_satisfied(self, partial_assignment)` - returns true if given partial assignemnt fulfills previously assigned constraints, false otherwise.

`possible_count(self, partial_assignment, index)` - returns number of values no longer accessible in domain for particular index given some partial assignment, used for MRV heuristic.

`constrain_count(self, partial_assignment, insert_index, domain_value)` - given value, returns (value, # of values it rules out for all adjacent variables), for assumed variable, aka `insert_index`, used for LCV heuristic.

`unassigned_heighbors(self, partial_assignment, var)` - returns set of arcs of given var and unassigned neighbors of given partial assignment for mac-3 inference.

`get_constraints(self, var1, var2` - returns set of constraints between two variables.

`MapColoringCSP.py`:

data:

`variables` - list of locations of map.

`domain` - list of colors for map.

`constraints` - corresponding constraints hashtable.

`CSP` - corresponding `ConstraintSatisfactionProblem` object.

functions:

`set_constraints(self, edges)` - sets constraints for CSP, with given list of neighbors.

`backtrack_search(self, mrv, degree, inference)` - converts information to integer format, runs backtrack serach in corresponding `ConstraintSatisfactionProblem` object, and then returns results with some syntax and the integers converted back to their original names/colors. `mrv`, `lcv`, and `inference` are boolean values determine whether or not to use such heuristics/inference.

`CircuitBoardCSP.py`:

data:

`length` - length of grid (int).

`height` - height of grid (int).

`pieces_list` - list of pieces.

`constraints` - corresponding constraints hashtable.

`CSP` - corresponding `ConstraintSatisfactionProblem object.

functions:

`set_constraints(self, square)` - sets constraints for CSP, with given whether or not the pieces are squares or not.

`coord_to_int(self, x, y)` - given x and y values, returns corresponding single int for given grid.

`int_to_coord(self, var)` - given single int for some grid, return corresponding x and y values in tuple.

`backtrack_search(self, mrv, lcv, inference)` - calls backtrack search object from CSP, and

returns output plus some syntax.

`solution_to_str(self, solution)` - given solution in integer format, returns it back with nice syntax in "circuit-board" format.

static functions:

`collision(i_loc, j_loc)` - given location, and piece information, returns whether or not the two objects collide at all or note. Resorts to brute force incase some objects are not rectangular.

`test_CSP.py`:

Tests creating different specific CSPs with hardcoded valeus corresponding to the specific problem of the specific problem type. Runs tests, and outputs them.

Backtracking Solver

The best naive method to solve CSPs is through use of backtracking. This is essentially depth-first search through the tree until either a solution is found, or the entire graph is iterated through, in which case it is assumed that problem has no solution. Depth-first is preferable to breadth-first, because it is faster nearly all of the time. Breadth-first spends the majority of its computing time iterating through states that are guarenteed not to contain a solution, as they are not at the max depth of the problem. Backtracking on the other hand, finds potential solutions (nodes at the max depth of the implied search tree) much earlier in the computing time, and as such is much more likely to find a solution faster. Worst case, backtracking and bfs are similar runtime because either backtracking was very unlucky and visited many or all of the non-solution states first, or because there is no solution found. Because backtracking is faster most of the time, and in a worst case scenario as fast as BFS, it is clearly the better algorithm.

The recursive backtrack search algorithm was written as such (sanity checks omitted):

```
def backtrack_search_helper(self, mrv, lcv, infer, inconsistencies):

    # base case: assignment is complete and valid
    if self.is_complete() and self.constraints.is_satisfied(self.assignment):
        return self.assignment

    # minimum remaining values heuristic
    if mrv:
        var = self.mrv_heuristic()
    else:
        var = self.no_variable_heuristic()

    # least constraining value heuristic
```

```
    if lcv:
        domain_list = self.lcv_heuristic(var)
    else:
        domain_list = range(self.domain_length)

    if infer:   # find infer values
        if var in inconsistencies.keys():
            for inconsistency in inconsistencies[var]:
                if inconsistency in domain_list:
                    domain_list.remove(inconsistency)   # dont consider inconsistenci
es

    for val in domain_list:

        self.assignment[var] = val

        if self.constraints.is_satisfied(self.assignment):
            if infer:
                inconsistencies = self.mac_infer(var)   # find inconsistencies from i
nference
                if inconsistencies is not None:
                    result = self.backtrack_search_helper(mrv, lcv, infer, inconsiste
ncies)
                else:
                    return None

            else:
                result = self.backtrack_search_helper(mrv, lcv, infer, inconsistencie
s)

            if result is not None:  # potential value found, use it for previous recu
rsive iteration
                return result

        else:   # increment fails
            self.fails += 1

        # backtracking
        self.assignment[var] = None

    # no solution, return None
    return None
```

Heuristics

While backtracking is a somewhat efficient method to find a solution, there are ways to potentially speed it up, by use of heuristics. As outlined before, there are aspects of CSPs that are selected: which variable to insert in, and which value to insert in that variable. As such, I implemented a heuristic for both of these situations.

Minimum Remaining Values:

This was the heuristic that I implemented in order to determine which variable to select the domain for (of all the variables that have not yet been selected). This heuristic looks at all of the variables that have not been assigned a value in the current partial assignment. It then finds the variable that is most constrained given the current partial assignment, and select that one for the next index to select to. The reasoning for this heurisitic is such: The current partial assignment can either result in a solution or not. If it does result in a solution, then choosing values for the variables that have the least choices will only speed up reaching such a solution. If the current partial assignment does not result in a solution, then choosing a value for the most constrained value is still a good idea because then it will fail faster than otherwise, in which case we can move onto a different partial assignment sooner. It can be seen that this heuristic returns an index which is used to insert into the variable assignment array at the proper location. This is a fail-fast approach for selecting variables.

The MRV Heuristic was written as such (with sanity checks omitted):

```
def mrv_heuristic(self):

    min_remaining = (None, float('-inf'))
    # iterate through unassigned variables
    for i in range(self.assignment_length):
        if self.assignment[i] is None:
            temp = self.constraints.possible_count(self.assignment, i)
            # if more constraining than previous most constraning, it is the new most
 constrainint
            if temp > min_remaining[1]:
                min_remaining = (i, temp)

    return min_remaining[0]
```

Also included is the `constraints.possible_count` function, which is in the `Constraint.py` file as it is part of a `Constraint` object:

```
def possible_count(self, partial_assignment, index):
    count = 0
    constrained_domains = set()
    for i in range(len(partial_assignment)):
        # if already set neighbor, then it is constraining given index:
        if partial_assignment[i] is not None and (i, index) in self.constraints.keys(
) and \
                        partial_assignment[i] not in constrained_domains:
            constrained_domains.add(partial_assignment[i])
            count += 1

    return count
```

Least Constraining Value:

This was the heuristic that I implemented in order to determine which value (in the domain) to select for the next value in the index. This heuristic looks to see which of the values would rule out the fewest values for the adjacent (unassigned) variables. The reasoning of this heuristic is that it makes sense to select values for variables that are not going to limit the future problem, because that means that the subsequent partial assignemnt is more likely to be one that will lead us to a solution assignment eventually. It can be seen that this heuristic actualy has to return a list, which is the order in which to select the possible domains. This is the case because if a single domain value is returned, then if that domain selection leads no solution, the other domain values must be checked as well. This is a fail-slow approach for selecting values.

The LCV Heuristic was written as such:

```
def lcv_heuristic(self, var):

    tuple_list = []
    domain_list = []

    # iterate through each possible value
    for i in range(self.domain_length):
        lc_val = self.constraints.constrain_count(self.assignment, var, i)

        #print("*", lc_val)

        # find correct index to insert
        j = 0
        index = 0
        while j < len(tuple_list):
            if tuple_list[j][1] < lc_val[1]:
                index += 1
            j += 1
        tuple_list.insert(index, lc_val)

    for i in range(len(tuple_list)):
        domain_list.append(tuple_list[i][0])

    #print("domain list:",domain_list, "\n")
    return domain_list
```

Also included is the `constraint_count` function, which is in the `Constraint.py` file as it is part of a `Constraint` object:

```
def constrain_count(self, partial_assignment, insert_index, domain_value):
    # copy the list, and add potential value
    potential_assignment = []
    for i in range(len(partial_assignment)):
        potential_assignment.append(partial_assignment[i])
    potential_assignment[insert_index] = domain_value

    count = 0

    # iterate through all unassigned neighbors of i
    for i in range(len(potential_assignment)):
        not_yet_constrained = True
        if potential_assignment[i] is None and (insert_index, i) in self.constraints.
keys():

            # also iterate through other assigned neighbors of i, to make sure value
is not already constrained
            for j in range(len(potential_assignment)):
                if j != insert_index and potential_assignment[j] is not None and (j,
i) in self.constraints.keys():
                    if potential_assignment[j] == domain_value:
                        not_yet_constrained = False

            if not_yet_constrained:
                count += 1

    return domain_value, count
```

Inference

If you could know whether or not setting some variables to some values would result in an unsolvable problem, then it would be extremely easy and fast to find solutions. You would merely pick the answers that do not result in an unsolveable probem, and once you had picked for every variable, you would end up with a solution! Inference attempts to do that. For this problem, we used MAC-3 inference. Below is the code for that:

It works by checkking the arcs of unassigned variables to the that are neighbors to the variable that just got added. That in turn will give us further information as to whether or not the value is worth considering, or if it is certainly going to lead to a non-solution. It iterates through the unassigned neighbors of the neighbor, and then compares their arcs to see if any valeus can be eliminated from their domain. It then continues the process until we have deleted all the values from the domain that we can do. This helps us make progress in knowing whether or not the value assigned to that variable is part of a solution or not. Note that this does not get rid of every possible bad value. For example, say for the map coloring problem a graph has three nodes, all

connected to each other. If all 3 can only be green and blue, then it will be considered fine by mac inference, but there is clearly no solution, as there must be 3 colors rather than 2 to solve this problem.

Below is the relevant code for MAC-3 inference:

```
def mac_infer(self, var):
    # dictionary mapping unassigned var ints to set of values that they cannot be:
    inconsistencies = {}
    queue = []

    # add all arcs with with one unassinged variables and given variable in them
    for initial in self.constraints.unassigned_neighbors(self.assignment, var):
        queue.append(initial)

    while queue:    # iterate through arcs
        arc = queue.pop()
        temp = self.mac_revise(arc, inconsistencies)

        # occurs when no possible value for some variable, means this tree is bad
        if temp is None:
            return None

        # add inconsistencies if found
        inconsistencies[arc[0]] = temp

    return inconsistencies
```

Here is the revise function:

```
def mac_revise(self, arc, inconsistencies):
    # set of all values var1 and var2 cannot be
    revised = set()
    if arc[0] in inconsistencies.keys():
        revised = inconsistencies[arc[0]]

    constraints = self.constraints.get_constraints(arc[0], arc[1])
    #print("constraints:",constraints)
    if constraints is None:
        return None

    # iterate through assignment, looking for inconsistencies
    for x in range(self.domain_length):
        cannot = 0
        for y in range(self.domain_length):
            if (x, y) not in constraints:
                cannot += 1
        if cannot == self.domain_length:
            revised.add(x)

    # no possible values, bad decision in past
    if len(revised) == self.domain_length:
        return None

    return revised
```

And finally, the two functions that are part of the `Constraints.py` file:

`unassinged_neighbors`:

```
def unassigned_neighbors(self, partial_assignment, var):
    neighbors = set()

    for i in range(len(partial_assignment)):
        if partial_assignment[i] is None and (var, i) in self.constraints.keys():
            neighbors.add((var, i))
            neighbors.add((i, var))

    return neighbors
```

`get_constraints`:

```
def get_constraints(self, var1, var2):
    if (var1, var2) in self.constraints.keys():
        return self.constraints[(var1, var2)]
    else:
        return None
```

Map Coloring

One example of a CSP problem is called Map Coloring. This problem consists of a graph and a set of colors. I set up the problem as such: Each node was a variable and the domain was all the colors that node could be painted (a graph is equivanelt to a map, neighboring regions are just adjacent nodes). Next, it is interesting to note that all neighbors in this graph have the same constraints. They just cannot have the same color node. So then I created a set of every single 2-color combination that did not involve the same color twice, and then finally I made that set the value of every adjacent node pairing as the key. That was the constraints dictionary. From then, I did some testing to see how the results went, using the Australia map outlined in the assignment. Below are the results:

Testing:

```
australia MapColoringCSP:
naive backtrack:
    {'NT': 'r', 'T': 'r', 'V': 'g', 'WA': 'g', 'NSW': 'r', 'Q': 'g', 'SA': 'b'}
    27 fails
naive backtrack w/ inference:
    {'NT': 'r', 'T': 'r', 'V': 'g', 'WA': 'g', 'NSW': 'r', 'Q': 'g', 'SA': 'b'}
    27 fails
mrv heuristic only:
    {'NT': 'r', 'T': 'r', 'V': 'g', 'WA': 'g', 'NSW': 'r', 'Q': 'g', 'SA': 'b'}
    27 fails
mrv heuristic only w/ inference:
    {'NT': 'r', 'T': 'r', 'V': 'g', 'WA': 'g', 'NSW': 'r', 'Q': 'g', 'SA': 'b'}
    27 fails
lcv heuristic only:
    {'NT': 'b', 'T': 'b', 'V': 'g', 'WA': 'g', 'NSW': 'b', 'Q': 'g', 'SA': 'r'}
    27 fails
lcv heuristic only w/ inference:
    {'NT': 'b', 'T': 'b', 'V': 'g', 'WA': 'g', 'NSW': 'b', 'Q': 'g', 'SA': 'r'}
    27 fails
both heuristics only:
    {'NT': 'b', 'T': 'b', 'V': 'g', 'WA': 'g', 'NSW': 'b', 'Q': 'g', 'SA': 'r'}
    7 fails
both heuristics only w/ inference:
    {'NT': 'b', 'T': 'b', 'V': 'g', 'WA': 'g', 'NSW': 'b', 'Q': 'g', 'SA': 'r'}
    7 fails
```

Discussion:

It is interesting to note that there are no improvements in the number of fails, except for when both heuristics are used. This is perhaps due to their effeclts not being the most profound in a map as small as this one, and such it is difficult to see much if any difference at all. Also note that this I used the worst order that I could find for inputting the order of the variables. Also, the inference was not effective for this problem, which is consistent with what was said in the textbook.

Circuit Board

Another example of a CSP problem is figuring out where to place certain components on a circuit board such that are all completely on the board, and do not overlap with each other at all.

In order to implement this sort of problem, I inputted a list of pieces, where each piece was a tuple of form `(char, int, int)`, whre the `char` was the symbol used to resresent the piece, the first `int` was the length of the piece, and the second `int` was the height of the piece. It was also necessary to input the length and height of the size of the circuit board grid. The domain for this problem is every single point in the

grid.

The next step is to set the proper constraints for the CSP. In order to do that for this problem, first I found every single potential point for the lefthand corner of each object such that the entire object fits within the grid. Then, I compared these sets between objects to find constraints between pieces. After this, I had all the information necessary to create my hashtable. This was done by checking every single available location for each of the two pieces. If those pieces did not collide (as per the collision function), then that location tuple was added to the set that eventually became the value for the key being a tuple of those two variables. From this point, the information had been processed sufficiently enough for the general CSP solver to be able to solve the problem and provide us with output.

Then, a couple functions were used to process the general CSP solver output and output it in a circuit form consistent with what was described in the assignment. Additionally, the number of failed variable assignments was included, so that it was possible to comapre efficency between the different heuristics/inferences.

Testing:

```
circuit CircuitBoardCSP:
naive backtrack:
aaabbbbbcc
aaabbbbbcc
eeeeeee.cc
31 fails

naive backtrack w/ inference:
aaabbbbbcc
aaabbbbbcc
eeeeeee.cc
18 fails

mrv heuristic only:
aaabbbbbcc
aaabbbbbcc
eeeeeee.cc
31 fails

mrv heuristic only w/ inference:
aaabbbbbcc
aaabbbbbcc
eeeeeee.cc
18 fails

lcv heuristic only:
```

```
cc.eeeeeee
ccbbbbbaaa
ccbbbbbaaa
432 fails

lcv heuristic only w/ inference:
cc.eeeeeee
ccbbbbbaaa
ccbbbbbaaa
162 fails

both heuristics only:
cc.eeeeeee
ccbbbbbaaa
ccbbbbbaaa
432 fails

both heuristics only w/ inference:
cc.eeeeeee
ccbbbbbaaa
ccbbbbbaaa
162 fails
```

Discussion:

There are some interesting results here. First, it is worth noting that the MRV heuristic was not especially effective for this problem, similar to the map coloring problem. I know that the MRV heuristic works properly because it was effective for some variable arrangements of the map coloring CSP. ALso, it was effective for the map coloring CSP heuristic when it was coupled with the LCV heurustic. The next thing you can see is that the the inference was especially effective, in that it always decreased the non-inference method. Finally, it is interesting to note that the LCV heuristic made matters a whole lot worse. I think that this is consistent with how the LCV heuristic works, for the following reasoning: the LCV will try to place the piece in a location that will be very open, because it will constrain other piece's domains the least. This is not a good way to solve the circuit board problem because the pieces end up all being very tightly packed, with very few open spaces. Hence, the LCV heuristic is actually placing pieces in very bad locations for the other pieces. That is why we see such a drastic increase in failures when the LCV heuristic is used. It is interesting to note that even in this case, using infereference still drastically decreasese the nubmer of failures from the non-inference version, and it is even perhaps more effective in this scenario. This might be because the program is taking a far from optimal path to solve the problem, so there is a lot of potential room for improvement that inferencing can make. While inferece is effective at making backtrack faster, it is worth noting that the inference algorithm is not free. So some extent some of the computational cost is just being shipped from the backtracking to the inference algorithm, so it is important to take that into consideration.

As a final conclusion, it seems that the effectiveness of heuristics and inference is highly dependent on the specifics of the CSP problem that is being solved. At least the general solver will find a solution as long as there is one to find.

Soduku (Bonus)

I also attempted to implement a `SudokuCSP` class which would solve sudoku problems. I tried implement it such that the domain of every variable was the numbers 1-9, and every variable was corresponding to a location on the board that was not filled by a given number. The constraints were then set based on which variables influence other variabes, and the given values as well. In all, it was very similar to my initial implementation of the `CircuitBoardCSP` object (see below), except for the fact that in this case, every single variable had to be filled, which made me inclined to think that it would be a functional way to implement it in this case.

In the end, it ended up running in an infinite loop when I tried it with a sudoku problem, which makes me think that there was most likely an issue with setting the correct constraints for the sudoku prioblem. I tried to debug, but there were so many variables and so many constraints that it was extremely difficult to figure out what was going wrong.

I also think that the partial assignments ended up wrong, which also makes me think that the setting constraints was failing. Also since my generic CSP solver worked for the other problems, I was inclined to think that was not the issue. Luckily, it appears that I will get another shot at writing a computer program that can solve sudoku problems in the next lab, which I look forward to doing! My sudoku code is included in this zip file, to show the approach that I took.

Literature Review (Bonus)

For further exploration, I decided to look at "Valued Constraint Satisfaction Problems: Hard and Easy Problems" by by Thomas Schiex, Helene Fargier, and Gerard Verfaillie (https://pdfs.semanticscholar.org/489e/ca86bf2dd3e30632059b8b0a658cae8da536.pdf). This text looks at existing examples of CSPs and how they take different forms, and what that means in terms of how difficult it is to solve them. It introduces the idea of valued CSP. It seems that this is for CSPs that are unable to be solved, in which case they are assigned a value which correspondings with how close they are to being solved. First, it claims that strict monotonicty is a desireable property for these problems, since it guarentees optimal consistent optimal relaxations of constraints (which is necessary if the CSP is not solveable) are always selected.

It then goes to discuss how to use extend traditional algorithms to solve such problems. It emphasizes that checking arc-consistency is very effective, as it is very prominent. They a Valued CSP problem (VCSP) as arc-consistent iff first there exists a value that defines an assignment with valuations strictly lower than that of a

secified "unaccaptable" valuation value for each variable, as well as any assignment A of one variable can be extended to an assignment A' on two variables, which has the same valuation.

It also discusses how to extend backtrack search for VCSPs. It uses a lower and upper bound of "admissable valuations", which it incrementally computes whenever a new variable is assigned.

I decided to look at this article because I think that VCSPs are a pretty interesting idea, in that you are trying to "optimize" a problem which has no solution. Furthermore, I tried a different implementation of the Circuit Board CSP initially, and failed because it would not have a complete solution. The implementation was that I considered every single point a potential variable, and the domain was which piece could be in that point. I thought this could be a good idea because then it would be extremely simple to make any sort of shape, based on custom inputs. Unfortunately, I found that it did not work because solutions did not necessarily involve every single point on the grid. Hence, many potential grids merely did not have solutions, because there were some open spaces. I thought about how I could fix this, and in the end I decided that it would have had to been acceptable to leave some values of null. But if some values were left as null, then why couldn't all values be left as null, in which case there actually was no solution? At that time I did not know the answer, but now from reading this article, I can tell that it seems quite similar to the VCSP problems that they outline. I could have potentially computed admissable valuatuations (detrmined by the number of empty spaces in the grid) and optimized that problem.