COSC 76 Project 4: Constraint Satisfaction Problem

Seungjae Jason Lee

Introduction

Objective

The objective of the project is to implement a backtracking solver for CSP and test it on different CSP problems. Also, we aim to test different heuristics and inferences to improve the backtracking solver.

CSP Rules

1. Map Coloring: The goal of the problem is to assign user-provided colors to different regions. The only rule is that neighboring regions should not have same colors.
2. Ciruit Board: Given the size of board and components, the goal is to find a way to fit all components on to the board without any overlaps.
3. CS10 Group Assignment(Extension): Given the TA list, student list and their available times, the goal is to assign a time that works for all groups; Each group must have one TA and total member of at least n/k-1 and at most n/k+1. (N = number of student + TA, K = number of TA)
4. Sudoku(Extension): Given fixed slots of the board, find a solution to the game. Every column, row, and square areas should not share same values. (same rule as regular sudoku).

Code Design

There are two main parts to the code design: backtracking and CSP model. Backtracking is implemented in backtracking.py and has functions that improve backtracking such as heuristics and inference. For the CSP model, I implemented CSP parent classes first and then made children classes based on specific rules of a problem.

Building CSP Model

Parent Class

As mentioned above, I implemented the parent class called ConstraintSatisfactionProblem. Within the class, it has three important variables:

1. self.assignment: a list where index is the variable and stored value is assigned value. By default, None is stored in a variable if a value has not been assigned yet.
2. self.domain: a list of list that contains domain for each variable
3. self.constraint: a dictionary that has all local constraints.

As for the methods, there are three important methods:

- `check_constraint_arc(self, key, value)` : checks constraint for given tuple key(binary constraint) and given tuple value. Used for arc consisteny inference.
- `check_constraint(self, variable, value)` : checks if assigning the value to the given variable satisfies constraints. It checks all constraints not just binary.
- `goal_test(self)` : checks if all variables have assigned values. If so, return True. Else, return False.

MapColoringProblem Class

For the MapColoringCSP class, the class inherits from the parent class and is implemented in MapColoringCSP.py. The class takes two inputs: list of locations and list of colors. Using these inputs, the class defines domain and variables. To build constraint, I made a method called `set_neighbor` , which builds binary constraint for two given locations. Also, I overrode `__str__` method so that it would nicely display the problem and solution if it exists.

CircuitBoardProblem Class

For the CircuitBoardProblem class, the class inherits from the parent class and is implemented in CircuitBoardProblem.py The class takes two inputs: list of components and a tuple of dimension. For a list of components, each component is a tuple of size three. First index stores a character representation of the component, and second and third indices store the left bottom corner coordinates of the component. Ex: ('a',3,2). For the tuple of dimensions, it stores (width, height) of a board. Using these inputs, the class defines domain, variable, and constraints using helper functions: `build_domain` , `build_all_constraint` , and `build_neighbor` .

Discussion: How does my algorithm build constraint? First, for each component, I find its domain so that the piece would be within the board. Then, I choose two pieces (lets say A and B) and build constraint for those two pieces. First, I evaluate x axis and for any location of A, B that do not overlap by x axis, I add them to the constraint. Then, for locations that overlaps by x axis, I check if they overlap by y-axis. If not, then I add them to

constraints. Actual code is implemented in `build_one_constraint`.

Building Backtracking Solver

Backtracking

'backtracking' function takes CSP as an input. In addition, it can also take other inputs such as AC*3(boolean), MAC*3(boolean), value*heu(function), and variable*heu(function) as optional inputs. These inputs are used later when we test heuristics and inference. `backtracking_recursive` function does the actual recursion to find a solution and takes CSP and assignment order as required inputs. assignment_order is a list that consists of indices of CSP.assignment, which is equivalent to list of variables. By default, assignment order is just in increasing order. (Ex: [0,1,2..]).

After implementing the basic backtracking function, I tested it with circuit problem and map coloring problem with examples in the assignment page. Images below display results.

```
Seungjae-Macbook-Pro:csproject4 SJLEE$ python3 MapColoringCSP.py
Map Problem with 3 colors and 7 locations
WA: r
NT: g
SA: b
Q: r
NSW: g
V: r
T: r
```

```
Seungjae-Macbook-Pro:csproject4 SJLEE$ python3 CircuitBoardProblem.py
Circuit Board Problem with 4 components and 10*3 dimensions
eeeeeee.cc
aaabbbbbcc
aaabbbbbcc
```

Inference

For inference, I implemented MAC-3 in a function called `inference_MAC_3`, using the pseudocode from the textbook. `inference_MAC_3` takes variable and CSP. For the given variable, the function finds the variable's neighbors and checks the arc consistency by checking neighbors' domains. Then, it modifies domains and makes all arcs consistent. If any of the domain becomes empty, it returns False and resets all of the domains. Else, it returns True and keeps the modified domains.

Variable Heuristic: Minimum Remaining Value

I implemented Minimum Remaining Value heuristic in `MRV` function in backtracking.py. This function takes a CSP object and an assignment order list. The function iterates over the assignment order list and finds the length of the domain. Then, it sorts the assignment order list based on the length of each domain. I used built in sort method from the python.

Value Heuristic: Least Constraining Value

I implemented Least Constraining Value in `LCV` in backtracking.py It takes CSP, assignment_index(variable) and its domain. For each value assignment in the domain, the function checks all the binary constraints and counts the changes in neighbors' domain. Then, the function sorts based on the count for each value using built-in sort method, and returns sorted domain.

Evaluation

In test.CSP.py, I made a test code for all of the methods mentioned above and some additional methods for extension.

The picture below shows the result of running MLV, LCV, and MAC-3 and AC-3 on a circuit board problem that does not have a solution. I was able to observe that MAC-3, AC-3 and MLV did reduce number of recursions. I noticed that AC-3 and MLV did also reduce the actual computation time. However, for LCV, it took more time because the domain is so big that the computation of LCV is too costly. As for the number of recursion, it did not reduce because the problem did not have any solution and had to check all solutions.

```
[Seungjae-Macbook-Pro:csproject4 SJLEE$ python3 test_CSP.py
Circuit Board Problem with 6 components and 10*10 dimensions
No Solution Found

Default----------------
number of recursion: 161
time taken without heuristics:  0.018557071685791016

MAC-3------------------
number of recursion: 137
time taken using MAC_3:  0.019275188446044922

MRV--------------------
number of recursion: 26
time taken:  0.0026900768280029297

LCV--------------------
number of recursion: 161
time taken using LCV:  0.334052085876464 84

AC-3-------------------
number of recursion: 137
time taken using AC-3:  0.011274099349975586
```

Extension

Extension 1: CS10 Problem In ClassGroupProblem.py, I implemented ClassGroupProblem, which models the group assignment problem for CS 10. It takes `time_list`, `ta_list`, and `student_list`. `time_list` stores a list of strings, which represent possible meeting times(['10AM,'11AM']). Then, ta*list and student*list consist of a list of lists, in which each list has a name and availability for each time slot(ex: ['Jason', 'O','X']). Then, I wrote methods that builds constraints and checks constraints. The result is shown below.

```
[Seungjae-Macbook-Pro:csproject4 SJLEE$ python3 ClassGroupProblem.py
Classroom Group Problem with 4 sessions, 3 TAs, 7 students
time  10AM  11AM  12AM  1PM
  t1    O    X    O    X
  t2    X    O    X    O
  t3    O    X    O    X
  s3    O    X    O    O
  s7    O    X    O    X
  s1    O    X    O    X
  s2    X    O    X    O
  s5    O    O    O    O
  s6    O    X    O    O
  s4    O    O    X    X
10AM: t1, s3, s7, s1
11AM: t2, s2, s5, s4
12AM: t3, s6
```

Extension 2: Sudoku I made my own search problem: Sudoku Problem. The problem takes a list of tuples for fixed slots. For example, tuple (5,6,7) means that there is 5 at (6,7) where (0,0) is the left bottom corner. This problem has 81 variables and 9 possible values. As for the constraint, since all of the constraints were Alldiff constraints, I overrode check_constraint method that would effectively check Alldiff constraint rather than building a constraint dictionary. I used the sudoku from wikipedia to test the problem and got the same answer.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

```
[Seungjae-Macbook-Pro:csproject4 SJLEE$ python3 SudokuProblem.py
Sudoku Problem:
53..7....
6..195...
.98....6.
8...6...3
4..8.3..1
7...2...6
.6....28.
...419..5
....8..79

Solved Sudoku Problem:
534678912
672195348
198342567
859761423
426853791
713924856
961537284
287419635
345286179
```

Extension 3: Improving Sudoku: Initially, I had my domain for each slot to be [1,2,3,4,5,6,7,8,9] for non-fixed slots and [value] for fixed slots. However, I noticed that the number of cases were too big and has to be reduced down. Thus, before the backtracking starts, I made a method called `reduce_domain` that would reduce down non-fixed slot domains by checking consistencies with fixed slot domains. Through this step, domains for non-fixed slots reduced by a lot, which made the computation way faster. This approach is equivalent to how human first approach sudoku problem (narrow down options), and the picture below shows an example of the description above.

57	579	3679	2	356	4	8	1	59
57	4	179	9	579	8	2	6	3
3	25789	2789	1	6	579	9	79	4
1	679	679	39	4	67	5	8	2689
6	3	5	8	2	1	49	9	7
2	78	478	5	9	67	1	38	68
9	1	236	7	15	256	368	4	268
45	25	234	6	8	259	7	2	1
8	267	267	4	1	3	69	5	269

```
Sudoku Problem:
53..7....
6..195...
.98....6.
8...6...3
4..8.3..1
7...2...6
.6....28.
...419..5
....8..79

Solved Sudoku Problem:
534678912
672195348
198342567
859761423
426853791
713924856
961537284
287419635
345286179

number of recursion: 1026
time taken with Reduced Domain:  0.017038822174072266
```

--> it took only 0.01 seconds!!!

Extension 4: Improving CS10 Problem For CS 10 Problem, the algorithm solves the problem by first assigning TA with time slots and then students. However, I noticed that if the last student on the list has only one availble slot and the slot is not available, the function has to go through many backtrackings to find a solution. Thus, I made `MRV_Group`, which is a MRV specially made for CS10 Problem. Given an assignment order of TA and Student, the function divides TA assignment order and student assignment order; then, it sorts each list on its own and returns TA*sorted + Student*sorted. This approach did reduce down the computation.

```
Classroom Group Problem with 5 sessions, 4 TAs, 19 students
```

```
time  10AM  11AM  12AM  1PM  2PM
  t1    O     X     O    X    O
  t2    X     X     O    O    O
  t3    O     X     X    O    O
  t4    X     O     X    X    O
  s1    O     X     O    X    O
  s2    X     X     O    O    O
  s3    O     O     X    O    O
  s4    X     O     X    X    X
  s5    O     O     O    O    O
  s6    X     X     X    O    O
  s7    O     X     X    X    O
  s8    O     O     O    X    O
  s9    O     X     O    X    X
  s10    O     X     O    X    O
  s11    O     X     O    O    O
  s12    X     X     X    X    O
  s13    O     X     O    X    X
  s14    O     X     O    O    X
  s15    O     O     X    X    O
  s16    X     X     O    O    O
  s17    X     O     O    X    O
  s18    O     X     X    X    X
  s19    O     O     X    X    O
10AM: t1, s1, s7, s9, s13, s18
12AM: t2, s2, s8, s10, s14, s16
2PM: t3, s5, s6, s11, s12
11AM: t4, s3, s4, s15, s17, s19


MRV_Group--------------------
number of recursion: 26
time taken using MRV_Group:  0.0008230209350585938

Default--------------------
number of recursion: 2982
time taken without MRV_Group:  0.08019423484802246
```

Extension 5: Implementing AC-3: In addition to implementing MAC-3, which checks only at specific variable, I also implemented AC-3. AC-3 inference checks before running backtracking and makes all domain consistent to each other. It is implemented in `inference_AC_3`

Demo is shown above in the **evaluation** part