A4: Constraint Satisfaction Problem

Kevin King

Dartmouth COSC 76, Fall 2023

Implementation

The CSP class is designed to represent a generic constraint satisfaction problem (CSP). It is initialized with a set of variables, each having a specified domain, and constraints that define the legal combinations of values for the variables. The class also accommodates variable and value selection heuristics and an inference mechanism to improve the efficiency of the solution process. It is equipped with methods that facilitate adding constraints, checking consistency, selecting variables and values, and performing backtracking search. Here is a detailed breakdown:

Initialization:

- `problem` : reference to the specific problem instance being solved (e.g., map coloring, circuit board layout)
- `variables` : list of variables for the problem
- `domains` : dictionary that maps each variable to its domain (list of possible values the variable can take)
- `constraints` : dictionary where keys are variables and values are lists of constraints associated with each variable
- `var_heuristic` : specifies the heuristic used for selecting the next variable to assign (e.g., MRV or degree)
- `val_heuristic` : specifies the heuristic for ordering the values in the domain of the variable
- `inference` : specifies whether the MAC3 inference technique is used during the search

Methods:

- `add_constraint` : adds a given constraint associated with a specific variable to the `constraints` dictionary
- `consistent` : checks if a given assignment of values to variables is consistent with the constraints.
- `select_unassigned_variable` : returns the next unassigned variable based on the order they appear in the `variables` list
- `mrv_heuristic` : implements the minimum remaining values heuristic for variable selection. It selects the variable with the fewest legal remaining values in its domain
- `lcv_heuristic` : implements the least constraining value heuristic. It orders the values of a given variable's domain based on how constraints each value participates in. Values involved in fewer constraints are considered first.
- `degree_heuristic` : implements the degree heuristic. It selects the variable that is involved in the most constraints with other unassigned variables
- `mac3_inference` : implements the maintaining arc consistency inference technique. It revises the domain of the unassigned variables to eliminate values that are not arc consistent with the current assignment. If it revises the domain to be empty, it reverts to the original domain and returns `False` , indicating failure

- `backtracking` : implements the backtracking search algorithm to find a solution to the CSP. It recursively assigns values to variables, checks consistency, applies heuristics and inferences, and backtracks when needed until a solution is found or all possibilities are exhausted.

MRV Heuristic: MRV tends to select the variable with the fewest legal values left in its domain. It opts to tackle the potentially problematic variables first.

Degree Heuristic: This heuristic selects the variable that is involved in the highest number of constraints on other unassigned variables. It's especially effective for this problem because choosing a region with more neighbors first can lead to a quicker resolution or detection of a failure.

LCV Heuristic: It orders the values to be tried within a variable's domain. It tries the value that rules out the fewest choices for neighboring variables first. It helps in reducing future conflicts and backtracks.

Inference: In the context of CSPs, inference is about removing values from the domains of unassigned variables based on the current partial assignment. This can significantly reduce the search space.

Map-Coloring Problem

`Variable` Class

This class represents a variable in the CSP. In the context of the map coloring problem, a variable would represent a region on the map that needs to be colored.

- Parameters:
  - `name` : A string representing the name of the variable.
  - `domain` : A list of possible values that can be assigned to the variable. In this case, it would be the colors available for coloring the map.
  - `value` : The actual value (color) assigned to the variable. Initially, it is set to `None` .

`Constraint_MP` Class

This class represents a binary constraint between two variables in the CSP. In the context of the map coloring problem, a constraint ensures that two adjacent regions (variables) do not have the same color.

- Parameters:
  - `variables` : A list containing two variables that are part of the constraint.
  - `constraints` : A dictionary where keys are variables, and values are lists of constraints associated with those variables.
- Methods:
  - `satisfied` : Checks if the constraint is satisfied given a particular assignment of values to variables. It returns `True` if either variable does not have an assignment yet, or if the assigned values of the two variables are different.

`MapColoringProblem` Class

- Parameters:
  - `variables` : A list of variables in the CSP.
  - `domains` : A dictionary that maps each variable to its respective domain of possible values.
  - `constraints` : A dictionary that maps each variable to a list of constraints associated with that variable.

  - `neighbors` : A dictionary that maps each variable to a list of its neighboring variables.
- Methods:
  - `add_constraint` : Adds a constraint to the list of constraints associated with a specific variable.
  - `get_neighbors` : Returns a list of neighbors associated with a specific variable.

Circuit-Board Layout Problem

`Constraint` Class

This is a generic class for constraints in the CSP. It is an abstract class meant to be subclassed by specific types of constraints.

- Parameters
  - `variables` : A list of variables that are involved in the constraint. Each constraint can involve one or more variables.
- Methods
  - `satisfied` : An abstract method meant to be overridden by subclasses to check if the constraint is satisfied with a given variable assignment.

`NoOverlapConstraint` Class

This class inherits from the `Constraint` class and implements a specific type of constraint to ensure that two components do not overlap on the circuit board.

- Parameters
  - `component1` : The first component that is being checked to ensure it does not overlap with the second component. It's one of the variables involved in this constraint.
  - `component2` : The second component that is being checked against the first for overlap. It's another variable involved in this constraint.
  - `sizes` : A dictionary containing the sizes of each component. Each key is a component, and the value is a tuple where the first element is the width and the second element is the height of the component.
- Methods
  - `satisfied` : Overrides the abstract method from the parent class to check if the two components overlap with a given assignment of positions. It calculates the positions and dimensions of the components and checks if they overlap. If either component does not have an assigned position yet, the constraint is considered satisfied.

`CircuitBoardProblem` Class

This class inherits from a not provided `CSP` class and represents the specific CSP of placing components on a circuit board.

- Parameters
  - `width` : The width of the circuit board.
  - `height` : The height of the circuit board.
  - `components` : A list of components that need to be placed on the circuit board.
  - `sizes` : A dictionary that maps each component to a tuple representing its size. The first element of the tuple is the width, and the second element is the height of the component.

- `**kwargs` : Additional keyword arguments that might be needed for the CSP superclass initialization. These can include specific heuristics or other optional parameters to configure the constraint satisfaction problem solver.
- Methods
  - `get_domains` : Calculates and returns the domains for each component, i.e., the possible positions where each component can be placed on the board without going out of bounds.
  - `display_solution` : Displays the solution on the console by printing the board with the components placed according to the given assignment.

Testing

`Map-Coloring`

Output

```
-----MAP COLORING PROBLEM-----

None, None, False
{'Western Australia': 'red', 'Northern Territory': 'green', 'Queensland': 'red',
'South Australia': 'blue', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'red'}

MRV, None, False
{'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia': 'blue',
'Queensland': 'red', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'red'}

None, LCV, False
{'Western Australia': 'red', 'Northern Territory': 'green', 'Queensland': 'red',
'South Australia': 'blue', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'red'}

MRV, LCV, False
{'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia': 'blue',
'Queensland': 'red', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'red'}

None, LCV, Inference
{'Western Australia': 'red', 'Northern Territory': 'green', 'Queensland': 'red',
'South Australia': 'blue', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'red'}

MRV, None, Inference
{'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia': 'blue',
'Queensland': 'red', 'New South Wales': 'green', 'Victoria': 'red', 'Tasmania': 'red'}

Degree, None, Inference
{'South Australia': 'red', 'Northern Territory': 'green', 'New South Wales': 'green',
'Western Australia': 'blue', 'Queensland': 'blue', 'Victoria': 'blue', 'Tasmania': 'red'}

Degree, LCV, Inference
{'South Australia': 'red', 'Northern Territory': 'green', 'New South Wales': 'green',
'Western Australia': 'blue', 'Queensland': 'blue', 'Victoria': 'blue', 'Tasmania': 'red'}
```

Every approach, regardless of the heuristic or inference method used, yields a valid solution for coloring the map. This is a typical characteristic of constraint satisfaction problems like map coloring, where multiple valid solutions can exist.

Runtime

```
-----MAP COLORING PROBLEM-----

None, None, False
Time: 2.86102294921875e-06
Time: 7.009506225585938e-05
Time: 7.915496826171875e-05
Time: 8.893013000488281e-05
Time: 9.584426879882812e-05
Time: 0.0001049041748046875
Time: 0.00011515617370605469

MRV, None, False
Time: 4.0531158447265625e-06
Time: 1.9073486328125e-05
Time: 3.790855407714844e-05
Time: 6.008148193359375e-05
Time: 8.702278137207031e-05
Time: 0.00011801719665527344
Time: 0.00015687942504882812

None, LCV, False
Time: 6.9141387939453125e-06
Time: 1.7881393432617188e-05
Time: 2.9087066650390625e-05
Time: 4.1961669921875e-05
Time: 5.1975250244140625e-05
Time: 6.389617919921875e-05
Time: 7.915496826171875e-05

MRV, LCV, False
Time: 6.9141387939453125e-06
Time: 2.4080276489257812e-05
Time: 4.601478576660156e-05
Time: 6.914138793945312e-05
Time: 0.00010085105895996094
Time: 0.0001380443572998047
Time: 0.00017404556274414062

None, LCV, Inference
Time: 3.910064697265625e-05
Time: 9.298324584960938e-05
Time: 0.00014281272888183594
Time: 0.0001971721649169922
Time: 0.0002532005310058594
Time: 0.00032901763916015625
Time: 0.0004069805145263672

MRV, None, Inference
Time: 3.886222839355469e-05
Time: 9.107589721679688e-05
Time: 0.00014781951904296875
Time: 0.00020623207092285156
Time: 0.0002720355987548828
Time: 0.0003390312194824219
Time: 0.00040912628173828125

Degree, None, Inference
Time: 3.814697265625e-05
Time: 0.00010013580322265625
Time: 0.0001471042633056641
```

```
Time: 0.00019288063049316406
Time: 0.000240325927734375
Time: 0.00028777122497558594
Time: 0.0003399848937988281

Degree, LCV, Inference
Time: 3.886222839355469e-05
Time: 8.893013000488281e-05
Time: 0.00014019012451171875
Time: 0.000186920166015625
Time: 0.0002391338348388672
Time: 0.0002899169921875
Time: 0.000347137451171875
```

The time taken to find a solution varies with different heuristics and inference methods. Generally, combining heuristics and inference mechanisms can lead to faster solutions, but it's not a strict rule as it highly depends on the specific problem and constraints.

`Circuit-Board`

Output

```
-----CIRCUIT BOARD LAYOUT PROBLEM-----

None, None, False
eeeeeee.cc
aaabbbbbcc
aaabbbbbcc

None, LCV, False
ccbbbbbaaa
ccbbbbbaaa
cceeeeeee.

MRV, None, False
ccbbbbbaaa
ccbbbbbaaa
cceeeeeee.

MRV, LCV, False
eeeeeee.cc
bbbbbaaacc
bbbbbaaacc

MRV, LCV, Inference
eeeeeee.cc
bbbbbaaacc
bbbbbaaacc

MRV, None, Inference
ccbbbbbaaa
ccbbbbbaaa
cceeeeeee.

None, LCV, Inference
ccbbbbbaaa
ccbbbbbaaa
```

```
cceeeeeee.
```

Different combinations of heuristics and inference mechanisms resulted in different layouts, indicating the flexibility in arranging the components while satisfying the constraints. There doesn't seem to be a single optimal layout as long as the constraints of non-overlapping components within the board's dimensions are met.

Runtime

```
None, None, False
Time: 7.152557373046875e-06
Time: 2.7894973754882812e-05
Time: 4.291534423828125e-05
Time: 5.1021575927734375e-05

None, LCV, False
Time: 2.6941299438476562e-05
Time: 5.507469177246094e-05
Time: 0.0001747608184814453
Time: 0.0003638267517089844

MRV, None, False
Time: 4.291534423828125e-05
Time: 0.00011110305786132812
Time: 0.0001888275146484375
Time: 0.00025081634521484375

MRV, LCV, False
Time: 0.00010180473327636719
Time: 0.00018715858459472656
Time: 0.0002830028533935547
Time: 0.00044918060302734375

MRV, LCV, Inference
Time: 0.00017404556274414062
Time: 0.0005340576171875
Time: 0.0008687973022460938
Time: 0.0011518001556396484

MRV, None, Inference
Time: 0.00014901161193847656
Time: 0.00045800209045410156
Time: 0.0007622241973876953
Time: 0.001020193099975586

None, LCV, Inference
Time: 0.00017690658569335938
Time: 0.0005083084106445312
Time: 0.0014569759368896484
Time: 0.0028259754180908203
```

The runtime increases when applying heuristics and inference, possibly due to the additional computations involved in applying these strategies. The most time-consuming method was the combination of None for variable ordering, LCV for value ordering, and Inference for pruning the search space. The simplest approach (None, None, False) was the fastest, likely due to the absence of additional computations from heuristics and inference.

Extensions

CS1 Section Assignment Problem

`SectionConstraint` Class

- Attributes:
  - `students` : A list of students to be assigned.
  - `leaders` : A list of section leaders.
- Methods:
  - `satisfied` : Returns True if the given assignment satisfies all constraints, otherwise returns False.

`StudentSectionAssignment` Class

- Attributes:
  - `students` : A list of students to be assigned.
  - `section_leaders` : A list of section leaders.
  - `domains` : The availability times of students and section leaders.
  - `constraints` : A list to store the constraints of the CSP.
  - `sections` : A dictionary to store the assigned sections after solving the CSP.
- Methods:
  - `get_neighbors` : Returns a list of all students and section leaders except the given variable.
  - `add_constraint` : Adds a constraint to the CSP.
  - `assign_sections` : Assigns students and section leaders to sections based on the solution of the CSP.

Implementation

- Utilizes a CSP framework to assign students to sections while respecting constraints.
- Uses the backtracking algorithm to explore possible assignments efficiently.
- Enhances the CSP's constraints to ensure that each section has a leader, and the number of students in each section adheres to the specified limits.
- If students are assigned to a section without a leader or constraints aren't met, the code identifies and reports it.
- Prints the final assignments of students to sections with their leaders if a valid assignment is found, including the case where some students couldn't be assigned due to their availability constraints.

Sample Test:

```
Domains (availabilities):
Student1: Tuesday 5:00, Wednesday 6:00, Monday 4:00
Student2: Monday 4:00
Student3: Wednesday 6:00, Tuesday 5:00, Monday 4:00
Student4: Monday 4:00
Student5: Monday 4:00, Wednesday 6:00
*Leader1: Wednesday 6:00, Tuesday 5:00, Monday 4:00
*Leader2: Monday 4:00, Wednesday 6:00, Tuesday 5:00

Solution found:
Student1 -> Tuesday 5:00
Student2 -> Monday 4:00
Student3 -> Wednesday 6:00
Student4 -> Monday 4:00
```

```
Student5 -> Monday 4:00
*Leader1 -> Wednesday 6:00
*Leader2 -> Monday 4:00

Sections:
Unassigned students based on availabilities: ['Student1']
Time: Monday 4:00, Section Leader: *Leader2, Students: Student2, Student4, Student5
Time: Wednesday 6:00, Section Leader: *Leader1, Students: Student3
```