report.md

Constraint Satisfaction Problems

Assignment 4
CS 276, Artificial Intelligence
Fall 2021, Dartmouth College
Spencer Bertsch

This report contains three sections, as seen below:

- Description
- Evaluation
- Responses to Discussion Questions

Description

- How do your implemented algorithms work?

Backtracking Search

Backtracking search is a recursive algorithm that assigns values to variables in a constraint satisfaction problem until one of the new assignments is not legal given the CSP's constraints. Once that occurs, then the algorithm "backtracks" and undoes the decision that caused the problem and tries another value from the variable's domain, then continues on in this way moving forward if all constraints are satisfied, and backwards whenever a new assignment violates a constraint. Inference and heuristics can be applied to the backtracking algorithm in order to improve its performance and help it avoid searching through large sections of the constraint graph unnecessarily.

AC-3 Algorithm

The AC-3 algorithm, or the inference algorithm in our implementation, is designed improve the performance of a CSP by removing some values from the domains of the variables currently being explored. It iterates through all the edges in the constraint graph and finds out for each x_i if there is a legal x_j. If there is a legal x_j, then it allows the backtracking algorithm to continue forward. If there is no value for x_j that allows the edge between x_i and x_j to be satisfied by the constraints (or by the dynamic constraint function), then we remove the inconsistent value from x_i's domain, and we add back any old edges that now need to be checked again. In this way, we are able to ensure that the direction we are traveling in through the constraint graph is a feasible one, and we are never getting into a corner which has no legal solutions which would require backtracking to get out of.

MRV Heuristic

The minimum remaining value (MRV) heuristic is a technique that orders the values in the domain before choosing and returning the first. It's used in the *select_unassigned_variable* method in the CSP. This method can either just return one of the possible values from the set of unordered values, or we can turn on the MRV heuristic or the Degree Heuristic to return one of the values from the ordered domain.

The MRV heuristic returns the variable that has the fewest possible remaining values in the constraint graph. For example if we could pick var1 that has 3 remaining values and var2 that has only a single remaining value, we would choose var2 first, assigning it the only possible value for var2.

Degree Heuristic

The degree heuristic is another way of ordering the variables in the *select_unassigned_variable* method, similar to the HRV heuristic. The degree heuristic returns the variable that has the largest number of connections in the constraint graph. For example, if we look below at the **Evaluation** section, we can find a constraint graph representing the regions of Australia. We see that the "SA" region has the most connections in the graph, so if the *select_unassigned_variable* method was choosing between "SA" and another variable and the Degree Heuristic was turned on, then "SA" would be the variable returned by the algorithm.

LCV Heuristic

The LCV heuristic works by returning a list of domains sorted so that the first value rules out the fewest neighboring values. The textbook discusses how this point is counterintuitive at first, but it makes sense that we would want to choose the variable that leaves the backtracking algorithm with as many options as possible to fill in the constraint graph.

- What design decisions did you make?

Because this assignment had no skeleton code, I set up the construction of the general CSP problem as a "general" CSP object. The methods of the CSP object provide the CSP solver with the backtracking search algorithm, the AC-3 algorithm, as well as each of the heuristics which can be enabled or disabled when the CSP object is instantiated.

In order to instantiate the CSP object, you need to pass in the variables **x**, the domains **d**, and the constraints **c**. In addition to the variables, domains, and constraints, the CSP object also takes an instance of the *solution* object which it updates during the search. This is very similar to the way that we have been keeping track of performance for the other search methods in earlier assignments. Once the CSP instance has been created, you can call the backtracking_search() method which runs backtracking on the problem and returns the solution object, providing both the solution and the number of nodes visited during the search.

The CSP object works dynamically to solve 'any' constraint satisfaction problem. The backtracking method, AC-3 method, and all heuristic methods work the same way for the map-coloring and circuits problems.

- How you laid out the problems?

I laid out the two problems by first creating a single, central class called CSP. The CSP class, as mentioned above, is capable of solving any similarly structured constraint satisfaction problem as long as the constraints are passed in as a list of tuples (in the case of map coloring), or the constraints can be found dynamically during th search (in the case of the circuits problem).

In order to test each problem, I created two files: *map_coloring.py* and *circuits.py* which each contain the variables, domains, and constraints for each problem. You can test each problem by running each of these files as executable scripts.

Evaluation

Do your implemented algorithms actually work? How well?

The CSP solver is able to solve both the map-coloring and the circuit design problems. Inference and heuristics all work for each problem as well, but some heuristics can introduce lower performance for the CSP solver depending on the way that they are constructed and the combination of heuristics and inference used.

The optimal solution for the map-coloring problem was found after searching 7 nodes, and the optimal solution to the circuits' problem was found after searching 19 nodes. Something interesting that I noticed was that for the MRV and Degree heuristics, I achieved better or consistent performance for the map-coloring problem when the (max/min) functions were reversed. When the MRV returned the variable with the maximum number of remaining values, it performed slightly better or the same for the map problem; subsequently when I chose the node in the constraint graph with the minimum connections using the degree heuristic then the backtracking algorithm exhibited slightly better or constant performance. See below tables for examples of each configuration:

	Backtracking Alone	W/ Inference	W/ LCV	W/ Degree Heuristic	W/ MRV	W/ All Activated
Nodes Visited	11	10	8	11	11	7

Table showing Map-Coloring performance with different combinations of inference and heuristics - MRV and Degree Heuristic Logic reversed

	Backtracking Alone	W/ Inference	W/ LCV	W/ Degree Heuristic	W/ MRV	W/ All Activated
Nodes Visited	11	10	8	14	11	8

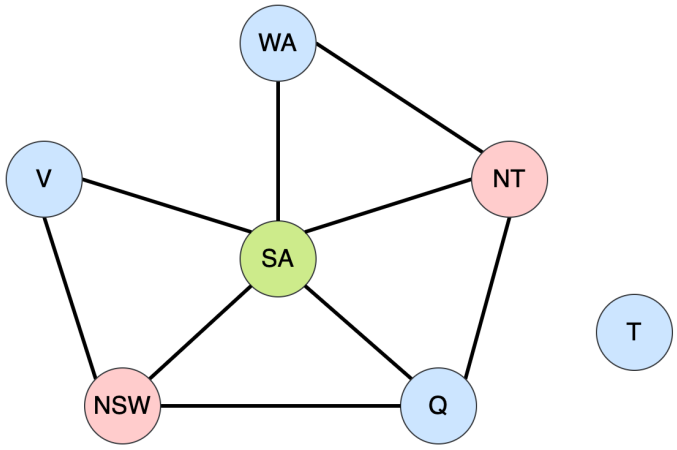
Table showing Map-Coloring performance with different combinations of inference and heuristics - MRV and Degree Heuristic Logic Corrected

The reason for this behavior is likely that these problems are relatively small. It would be helpful to test this problem on a constraint graph representing each state in the United States, or each country in Europe so that the relationships between the heuristics and the ways that each perform can be tested more thoroughly.

Optimal solution for the map-coloring problem:

```
Constraint Satisfaction Problem: Map Coloring
Nodes Visited: 7
Answer: {'WA': 'blue', 'SA': 'green', 'NT': 'red', 'Q': 'blue',
         'NSW': 'red', 'V': 'blue', 'T': 'blue'}
```

As seen in the above table, the best performance for the map-coloring problem as achieved when all heuristics and inference is turned on.



Graphic showing a feasible map-coloring solution found using the generic CSP solver

This is one of the many viable solutions to this map coloring problem. Another solution, for example, could easily be found by exchanging all green variables with red variables and vice versa. The initialization of the problem - the ordering of the variables and the constraints that we pass into the solver - dictate the solution found and returned by the backtracking search.

Optimal solution for the circuits problem:

The circuits problem was also solved by the generic CSP solver, yielding different feasible solutions as different combinations of heuristics and inference are activated and deactivated.

```
-------------- CIRCUIT BOARD LAYOUT --------------
 ['c', 'c', '.', 'e', 'e', 'e', 'e', 'e', 'e', 'e']
 ['c', 'c', 'b', 'b', 'b', 'b', 'b', 'a', 'a', 'a']
 ['c', 'c', 'b', 'b', 'b', 'b', 'b', 'a', 'a', 'a']

Constraint Satisfaction Problem: Circuit Design
```

```
Nodes Visited: 19
Answer: {(3, 2): (7, 1), (5, 2): (2, 1), (2, 3): (0, 0), (7, 1): (3, 0)}
```

The optimal performance for the circuits problem was achieved when the LCV heuristic was activated. When This heuristic was turned on, A solution to the circuits problen was found after visiting 19 nodes. See the below tables for an overview of the performance of the backtrackinig algorithm applied to the circuit layout problem with different heuristics and inference activated.

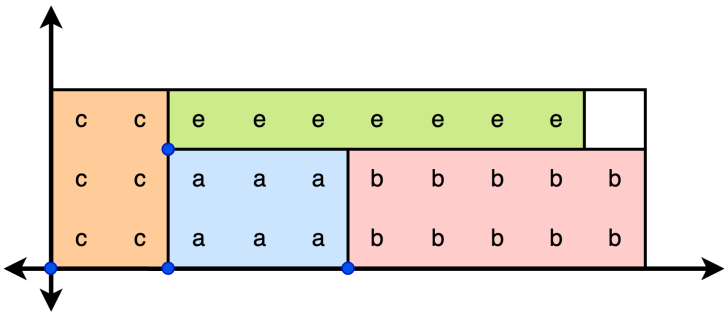
	Backtracking Alone	W/ Inference	W/ LCV	W/ Degree Heuristic	W/ MRV
Nodes Visited	20	20	19	20	20

Table showing Map-Coloring performance with different combinations of inference and heuristics - MRV and Degree Heuristic Logic reversed

	Backtracking Alone	W/ Inference	W/ LCV	W/ Degree Heuristic	W/ MRV
Nodes Visited	20	20	19	20	30

Table showing Map-Coloring performance with different combinations of inference and heuristics - MRV and Degree Heuristic Logic Corrected

Note that the solution to this problem, like the map coloring problem, is not unique. The below diagram and the console log show different solutions produced by the CSP solver using different heuristics. Each solution is acceptable, but using different heuristics for each problem simply improves performance of the algorithm by reducing the number of nodes that need to be visited during each search.



Graphic showing a feasible board state after the CSP solver completes the circuits problem

See below for another board design generated by the CSP solver when different heuristic are activated.

```
-------------- CIRCUIT BOARD LAYOUT --------------
 ['a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'c', 'c']
 ['a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'c', 'c']
 ['e', 'e', 'e', 'e', 'e', 'e', 'e', '.', 'c', 'c']
```

Responses to Discussion Questions

1. Describe the results from the test of your solver with and without heuristic, and with and without inference on the map coloring problem.

The map-coloring problem's performance is improved when inference is turned on, and the performance is improved even further when heuristics are turned on as well. The optimal solution (in which only 7 nodes are visited before a solution is found) can be achieved when inference and all the heuristics are activated. As noted above, by reversing the logic for the Degree Heuristic and the MRV Heuristic, an optimal value of 7 nodes visited could be achieved; otherwise an optimal value of 8 nodes was achieved. See the below tables for a look at how the heuristics and inference impact the performance of backtracking on the map-coloring problem.

	Backtracking Alone	W/ Inference	W/ LCV	W/ Degree Heuristic	W/ MRV	W/ All Activated
Nodes Visited	11	10	8	11	11	7

Table showing Map-Coloring performance with different combinations of inference and heuristics - MRV and Degree Heuristic Logic reversed

	Backtracking Alone	W/ Inference	W/ LCV	W/ Degree Heuristic	W/ MRV	W/ All Activated
Nodes Visited	11	10	8	14	11	8

Table showing Map-Coloring performance with different combinations of inference and heuristics - MRV and Degree Heuristic Logic Corrected

2. Describe 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 size of the domain would be [(n-w)*(m-h)]. The piece can shift to the right until it hits the right wall, leaving (n-w) spaces open from the origin (0,0). Similarly, moving the piece up will cause it to run into the top of the board, leaving (m-h) many spaces below the piece.

The piece will be able to move in a grid which will be a subset of the (n*m) grid created by the board. The domain of the piece will exist as this smaller grid with height (m-h) and width (n-w).

More specifically, we could represent the domain as a list of tuples, each of which represents an [x,y] coordinate. We could start at the origin (0,0) and move right, adding tuples (1,0), (2,0) until we get to (n-w, 0). We could then move up and cover the next row, then the next, until we reached a height of (m-h). If we were adding values iteratively from origin upwards, left to right, the last tuple coordinate added would be ((n-w), (m-h)).

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

The constraint that I used that does not allow these components to overlap can be described in the following way:

[(a) is below (b) OR (a) is above (b) OR (a) is left of (b) OR (a) is right of (b)].

As long as the above constraints are satisfied, then we know that the two pieces are not overlapping. We could also consider the constraint of being off the board, but I wrote code to produce the initial domains for each piece on the board given it's size, so it's impossible for pieces to fall off the board given my construction. A more specific formulation of the above constraint statement is: A top corner of (a) is below a bottom corner of (b) OR a bottom corner of (a) is above a top corner of (b) OR a left corner of (a) is right of a right corner of (b) OR a right corner of (a) if left of a left corner of (b). This is the logic that I implemented in my CSP solver for the circuits problem.

A total list of legal positions for pieces (a) and (b) on a 10x3 board can be seen below:

Positions when piece (a) is on the left and (b) is on the right

{a: (0,0), b: (3,0)}
{a: (0,0), b: (3,1)}
{a: (0,0), b: (4,0)}
{a: (0,0), b: (4,1)}
{a: (0,0), b: (5,0)}
{a: (0,0), b: (5,1)}
{a: (0,1), b: (3,0)}
{a: (0,1), b: (3,1)}
{a: (0,1), b: (4,0)}
{a: (0,1), b: (4,1)}
{a: (0,1), b: (5,0)}
{a: (0,1), b: (5,1)}
{a: (1,0), b: (4,0)}
{a: (1,0), b: (4,1)}
{a: (1,0), b: (5,0)}
{a: (1,0), b: (5,1)}
{a: (1,1), b: (4,0)}
{a: (1,1), b: (4,1)}
{a: (1,1), b: (5,0)}
{a: (1,1), b: (5,1)}
{a: (2,0), b: (5,0)}
{a: (2,0), b: (5,1)}
{a: (2,1), b: (5,0)}
{a: (2,1), b: (5,1)}

Positions when piece (b) is on the left and (a) is on the right

{a: (7,0), b: (0,0)}
{a: (7,0), b: (0,1)}
{a: (7,0), b: (1,0)}
{a: (7,0), b: (1,1)}
{a: (7,0), b: (2,0)}
{a: (7,0), b: (2,1)}
{a: (7,1), b: (0,0)}
{a: (7,1), b: (0,1)}
{a: (7,1), b: (1,0)}
{a: (7,1), b: (1,1)}
{a: (7,1), b: (2,0)}
{a: (7,1), b: (2,1)}
{a: (6,0), b: (0,0)}
{a: (6,0), b: (0,1)}
{a: (6,0), b: (1,0)}
{a: (6,0), b: (1,1)}
{a: (6,1), b: (0,0)}
{a: (6,1), b: (0,1)}
{a: (6,1), b: (1,0)}
{a: (6,1), b: (1,1)}
{a: (5,0), b: (0,0)}
{a: (5,0), b: (0,1)}
{a: (5,1), b: (0,0)}
{a: (5,1), b: (0,1)}

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

This is an interesting question; the generic CSP solver uses constraints in different ways based on the type of problem being solved. The constraints in the map-coloring problem are hard coded as a list of tuples, each tuple representing two variables (countries) that cannot be adjacent. The constraints for the circuit design problem on the other hand are found dynamically as the logic in the *test_consistency* method yields legal and illegal positions for pieces on the board. The conversion to integer values occurs in the *test_consistency* function in which each string value in the input to the problem is considered using logic that often converts the constraints, domains, and variables to integers, then returns a bool value. The majority of the generic CSP solver uses the exact same methods for each type of CSP problem, treating the inputs from either problem the same way. As mentioned, the *test_consistency* method is the main place where the generic CSP solver differs; one part of the function is designed for testing legality of the assignment for map-coloring, and the other part is designed to test legality for the layout of the circuit board.