PA 4 - CSP

CS 76 AI, Fall 2021

Melissa Valencia

Overall Design

`CSPModel.py` contains the general constraint problem satisfaction model and the algorithms necessary for solving the problem. These algorithms include the recursive search algorithm, the heuristics for variable and value selections, and the inference technique, AC-3. The recursive search algorithm, `backtracking_search` and `backtrack`, essentially take a depth first search approach where the reordering of the selected variables and the constraints of said variable are taken into account allowing the consideration of only values that do not conflict with previous assignments. To improve this backtracking search, we can apply heuristics to the variable and domain selection approach. These heuristics include the `minimum_remaining_value` heuristic and the `degree` heuristic for variable selection and the `least_constraining value` heuristic for value selection. The MRV heuristic will return the variable that has the least remaining possible values. The Degree heuristic will return the variable that has the most amount of constraints. The LCV will return the domain value which is the least constraining for the variable given. To further improve the search, we can apply an inference technique, AC-3, for reducing the space of possible assignments and early failure detection. The two problems solved through this `CSPModel` are in `MapColoringCSP.py` and `CircuitCSP.py`. Each of these respectively provide the variables, domains, constraints, and constraint satisfaction method, `check_consistency`, necessary for finding a valid solution.

CSP Model

A constraint satisfaction problem is solved when each given variable has a value from the given domain that satisfies all the specified constraints on the variable. With the use of heuristics and inference techniques, the CSP backtracking search algorithm can help detect early failures by identifying the possible assignments that do not satisfy the constraints. A CSP is made of three main components: the variables, the domains, and the constraints. Each of these are defined within the specific problems to be solved which are then passed in. In my `CSPModel.py`, I chose to add a fourth component, a domain dictionary which keeps track of all the variables and their possible domains, which are then altered based on the value removals occuring when the inference is set to true.

Implementation

In order to solve the CSP problems specified, this CSP Model ultimately uses a backtracking search algorithm to find a valid solution based on the given constraints. This backtracking search relies on various helper functions to choose values for a variable at a time and backtrack when the variable no longer has any valid values left to assign. Valid values refers to values in the variable's domain which satisfy its constraints. First, the `backtrack` function checks whether a solution has been found yet by comparing the length of the assignment to the length of variables, assuring that each variable has been assigned a valid value. Then, the function chooses an unassigned variable, returned from `select_unassigned_variable`. The `select_unassigned_variable` function is given a specified heuristic, either "MRV" or "DH" or "None". If given "None", the variable returned will simply be the next unassigned variable in the variable set. If given "MRV" or "DH", the variable returned will be that from either calling the `minimum_remaining_value` function or the `degree` function. Once the variable is obtained, the backtracking algorithm chooses the order in which to check the values by calling `order_domain_values`. A specified heuristic, either "LCV" or "None", are passed into the `order_domain_value` function. If given "None", the domain returned will simply be the domain assigned to all the variables. If given "LCV", the values returned will be the ones which rule out the fewest choices for the variables' neighbors, essentially the domain will be set in order by the least constraining domain values. Then, the backtracking algorithm will go through these values and check that they satisfy the constraints for the given variable, using `check_consistency`. `check_consistency` is defined in each CSP problem separately based on their specified constraints. If this value is valid per the constraints of the variable, then the variable and its value will be added to the assignment. Since the inference is set to true, it will then run the inference technique function, `ac3`. AC-3 keeps track of a set of arcs, in this case the variables and their neighbors, to consider. It will then check a pair (variable, neighbor) and see if there are values in the domain that can be eliminated. If AC-3 returns true, then `backtrack` will be called recursively and we will check for a result returned. Eventually, if a solution exists, it will be returned.

Map Coloring CSP

The Map Coloring CSP is characterized by being given variables which represent the states in a country, a domain of the colors that can be assigned to the variables, and a constraint stating the colors of neighboring states cannot be the same. The goal here is to assign a color to every state, ensuring that all neighboring states will be a differing color from the domain.

Implementation

To use my CSPModel for finding a solution to this Map Coloring CSP, I developed my own test cases as seen in `MapColoringCSP.py` and passed the respective components into `CSPModel.py` to then find a solution using `backtracing_search`.

Discussion Questions

I tested this MapColoringCSP on the map of Australia. I set the variables to be the states, the domain to be the values [red, blue, green], and the constraints to be defined by the neighbors of each state. I also specified the constraint satisfaction check, ensuring that an assigned variable and its neighbors are not the same value. Inference can be changed to false by simply making the `is_inference` function return false. However, I chose to keep inference as true always as returned by this function. My test cases in `MapColoringCSP.py` use various combinations of heuristics to find a solution. My output from this is shown below.

```
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'QU': 'red', 'NSW': 'green',
'VI': 'red', 'TA': 'red'}
backtrack calls 8
```

```
{'SA': 'red', 'NT': 'green', 'QU': 'blue', 'NSW': 'green', 'WA': 'blue',
'VI': 'blue', 'TA': 'red'}
backtrack calls 8
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'QU': 'red', 'NSW': 'green',
'VI': 'red', 'TA': 'red'}
backtrack calls 8
{'SA': 'red', 'NT': 'green', 'QU': 'blue', 'NSW': 'green', 'WA': 'blue',
'VI': 'blue', 'TA': 'red'}
backtrack calls 8
```

Circuit-Board Layout CSP

The Circuit-Board Layout CSP relies on various classes included in `CircuitCSP.py` and `CircuitBoard.py`. In `CircuitBoard.py`, the class necessary for setting the circuit board dimensions, the function used for adding a rectangle to the board, and for printing the board. This file also includes the class necessary for setting the rectangle dimensions and functions for finding the rectangle's positions and checking for intersection between two rectangles. `CircuitCSP.py` sets the variables, domains, constraints, and consistency check function and passes the respective components into `CSPModel.py` to then find a solution using `backtracing_search`. There are test cases in both `CircuitCSP.py` and `CircuitBoard.py`. I varied heuristics within my test cases in `CircuitCSP.py`. My output from it is shown below.

```
{j: (0, 0), k: (3, 0), l: (8, 0), m: (0, 2)}
--board--
mmmmmmm.ll
jjjkkkkkll
jjjkkkkkll
```

Discussion Questions

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 would be those coordinates (x,y) within the dimensions of the circuit board for which a component of width w and height h can fit in. The constraint that enforces the fact that the two components may not overlap is dependent on the comparison between the left and right most points and the top and bottom most points of the two components. There should not be any intersection between these points.