PA-4 [21F] [Benjamin Cape] [CSP]

Description

My Algorithms work almost identically to the psuedocode provided in the book. In order to make copying data structure simpler, and consolidate references into a single location, I created a map in the specific problem definition from variable objects to integers. In this way, behind the scenes, in the CSP logic we are always dealing with integers, while in the implementation of each specific problem, and to the viewer we see the actual variable names.

Constraints are represented as binary pairs. Since both of the problems we needed to implement only rely on binary relations this is OK for our implementation. An extension of this could easily create triples, and quadruples, name them with an Enum differently, that way the `is_consistent` function would know how to determine whether an assignment were consistent. Since each constrain in our definition is the same, the `is_consistent` is very simple, and runs in O(n) time.

This is the only function that each specific problem needs to implement. Otherwise, the algorithm for finding a consistent and complete assignment is standardized for all CSPs that can define a BinaryCSP with the following structure:

```
{
    variables: List[int],
    constraints: Mapping[Tuple[int, int], str],
    Mapping[int, Set[int]],
}
```

Given these defined ways, where each variable is an `int`, each domain value is also an `int`, and each type of constraint is a `str`, we can easily calculate a CSP solution.

There are two toggle-able values, `var_h` and `val_h`. The first allows for specifying a heuristic for next variable selection, and the latter allows for specifying a heuristic for calculating the next value. You can either input an enum value to specify a generic heuristic, or you can pass a function in to specify a unique, user-specified heuristic. I decided to do this because it allows for experimentation on possibly better heuristics.

The `backtracking` function is the biggest component of the CSP logic. Here we make sure to do correct influencing. There are three things that the inference function can return:

- {} - an empty dictionary, meaning we have a valid (or based on AC-3 we think it's valid) assignment.
- None - we know we have an invalid assignment (this is equivalent to *failure* in the textbook)
- {...} - a dictionary with some values. These are the values for which the size of the remaining domain is 1, meaning we can add them into our

assignment.

If we don't get a None value, we continue down the DFS search tree continuing to add to our assignments. If we have values in our dictionary, GREAT, if not, then keep going but we haven't increased efficiency yet. (It might happen later)

Since we change teh domains of the variables in the inference, if we get a None value because we removed too many values, when we don't realize a solution, we need to reset the domain values. That is why we `deepcopy(.)` the `self` (CSP). This way we can simply reset to what we had before and try again with the next value for the current variable.

Since we don't ever mutate the assignment, we only ever copy it (essentially created a persistent data structure within the DFS search tree) we don't need to reset the assignment such as in the psuedocode.

I laid out the problem such that you can define any CSP, with a definite set of variables, a board, or anything. The only things that need to change are how you structure the constraints, the variables, and then check if the assignment is consistent. Since you define the constraint shape, and you define the consistency check. almost all the power is in the hands of the developer.

Evaluation

Yes, my algorithms work. And they work quite well it appears. With heuristics we can solve the color game in checks linear to the number of variables, even fewer sometimes. We see a dramatic decrease in running time and in values checked when we use heuristics and when we use inference checking.

Discussion

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

```
With Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 3
|- Assignment: {'WA': 0, 'OR': 1, 'CA': 0, 'ID': 2, 'MT': 0, 'BC': 1, 'WY': 1}
|- Variable Heuristic: None
|- Value Heuristic: None
|- Time Taken: 1.65 ms
|- Values Checked: 11
|- Calls to Backtracking: 8
```

```
|-----

Without Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 3
|- Assignment: {'WA': 0, 'OR': 1, 'CA': 0, 'ID': 2, 'MT': 0, 'BC': 1, 'WY': 1}
|- Variable Heuristic: None
|- Value Heuristic: None
|- Time Taken: 1.368 ms
|- Values Checked: 12
|- Calls to Backtracking: 8
|-----

With Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 3
|- Assignment: {'WA': 0, 'OR': 1, 'CA': 0, 'ID': 2, 'BC': 1, 'MT': 0, 'WY': 1}
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: None
|- Time Taken: 1.594 ms
|- Values Checked: 12
|- Calls to Backtracking: 8
|-----

Without Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 3
|- Assignment: {'WA': 0, 'OR': 1, 'CA': 0, 'ID': 2, 'MT': 0, 'BC': 1, 'WY': 1}
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: None
|- Time Taken: 1.43 ms
|- Values Checked: 12
|- Calls to Backtracking: 8
|-----
```

```
With Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 3
|- Assignment: {'WA': 0, 'OR': 1, 'CA': 0, 'ID': 2, 'BC': 1, 'MT': 0, 'WY': 1}
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 1.016 ms
|- Values Checked: 7
|- Calls to Backtracking: 8
|-----

Without Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 3
|- Assignment: {'WA': 0, 'OR': 0, 'CA': 0, 'ID': 0, 'MT': 0, 'BC': 0, 'WY': 0}
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 0.707 ms
|- Values Checked: 7
|- Calls to Backtracking: 8
|-----

With Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 2
|- Assignment: None
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 1.778 ms
|- Values Checked: 14
|- Calls to Backtracking: 7
|-----
```

```
Without Inference:

|-----
|- Solution:
|-
|- Variables: ['WA', 'OR', 'CA', 'ID', 'MT', 'BC', 'WY']
|- Colors: 2
|- Assignment: None
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 5.011 ms
|- Values Checked: 46
|- Calls to Backtracking: 23
|-----
```

Here are my results from the coloring problem with and without heuristics and with and without inferences. It is interesting to note that with the heuristic and without inference we sometimes do worse than otherwise. This is probably because we are on such a small graph that we aren't seeing the benefits of the heuristic.

Note that we always do better though with inferences.

2. (circuit-board) In your write-up, 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.

I consider each piece to be a variable. The domain of the variable is described by the following set.

$$\{(x, y)|x \in [0, n - w + 1] \wedge y \in [0, m - h + 1]\}$$

The $n - w + 1$ and $m - h + 1$ ensures that the pieces will always be on the board. The (x, y) that gets assigned to a piece is the bottom left location for the piece.

3. (circuit-board) 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.

There are a few ways to do this. If we want to think of each piece as a tetris piece, then we need a most costly measure.

This costly measure is calculating every interior coordinate of any two given pieces and checking if there is any overlap.

An easier solution of all pieces are rectangular is to do a similar overlapping rectangular check.

I implemented the former so that I can extend the problem to tetris pieces and not just rectangular pieces. Because of this extension I cannot explicitly write out the legal pairs. BUT, if we were talking about rectangles then check out the `doOverlap` function in `circut.py`

But for the simple problem of the 2-d circut pieces (just width and height) I implemented the more clever rectangle overlap. Explicitly writing this down is noted above in the `doOverlap` function. Define S as the set of all possible starting locations for b, given that a is already placed at position (a_x, a_y) and has dimensions: a_w, a_h.

$S = \{(x, y) | x > a_x + a_w \vee y > a_y + a_h \vee a_x > x + b_w \vee a_y > y + b_h\}$

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

This is already described in the description, and it is specific to each problem. For example, with coloring we actually don't need to do this because strings are the same as integers according to python (both primitives),

Testing with the circut problem

```
With Inference:

|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (5, 1), (5, 1), (2, 1), (2, 1), (2, 1), (7, 1)
|- Assignment:
|-   aaaddddddd
|-   bbbbbbbbbb
|-   ccccaaa.cc
|- Variable Heuristic: None
|- Value Heuristic: None
|- Time Taken: 25.686 ms
|- Values Checked: 10
|- Calls to Backtracking: 9
|-----

Without Inference:

|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (5, 1), (5, 1), (2, 1), (2, 1), (2, 1), (7, 1)
|- Assignment:
|-   aaaddddddd
|-   bbbbbbbbbb
```

```
|-   ccccaaa.cc
|- Variable Heuristic: None
|- Value Heuristic: None
|- Time Taken: 156.605 ms
|- Values Checked: 110178
|- Calls to Backtracking: 7549
|-----

With Inference:

|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (5, 1), (5, 1), (2, 1), (2, 1), (2, 1), (7, 1)
|- Assignment:
|-   .bbbbbcccc
|-   ddddddddaaa
|-   aaabbbbbcc
|- Variable Heuristic: VarHeuristic.MRV
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 49.358 ms
|- Values Checked: 63
|- Calls to Backtracking: 28
|-----

Without Inference:

|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (5, 1), (5, 1), (2, 1), (2, 1), (2, 1), (7, 1)
|- Assignment:
|-   .bbbbbcccc
|-   ddddddddaaa
|-   bbbbbaaacc
|- Variable Heuristic: VarHeuristic.MRV
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 547.663 ms
|- Values Checked: 3266
|- Calls to Backtracking: 129
|-----

With Inference:

|-----
|- Solution:
```

```
|-
|- Variables: (3, 1), (3, 1), (5, 1), (5, 1), (2, 1), (2, 1), (2, 1), (7, 1)
|- Assignment:
|-   .bbbbbcccc
|-   ddddddddaaa
|-   aaabbbbbcc
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 51.05 ms
|- Values Checked: 63
|- Calls to Backtracking: 28
|-----
```

```
Without Inference:
```

```
|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (5, 1), (5, 1), (2, 1), (2, 1), (2, 1), (7, 1)
|- Assignment:
|-   .bbbbbcccc
|-   ddddddddaaa
|-   bbbbbaaacc
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 599.002 ms
|- Values Checked: 3266
|- Calls to Backtracking: 129
|-----
```

Notice the SUBSTANTIAL speed increase when we use inference. It is interesting to note that combining heuristic and inference does worse than no heuristics and inference. This might show that my heuristics don't work properly, but I don't believe that is the case, since with the coloring problem we do in fact see the heuristics doing a good job. Rather, This is simply a function of the ordering being incredibly good. Notice if I change the order of the variables I get worse output:

```
With Inference:
```

```
|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (2, 1), (2, 1), (2, 1), (5, 1), (5, 1), (7, 1)
|- Assignment:
|-   aaaddddddd
|-   bbbbbbbbbb
```

```
|-   ccccaaa.cc
|- Variable Heuristic: None
|- Value Heuristic: None
|- Time Taken: 138.229 ms
|- Values Checked: 170
|- Calls to Backtracking: 26
|-----
```

Whereas the values for with heuristic is the same. The reasons for this are that I was lucky in picking an ordering that was in fact better than the heuristics!

Extra Credit

For the extra credit, I have attempted to complete the cs1 assignment problem, though that proved difficult.

I feel like I was unable to properly define the constraints such that this puzzle is consistently solved properly. Thought I was close. Code can be seen in `cs.py` One thing I think trying to solve this problem helped me with was better defining a generic CSP solver! Here is a solution I got with the code on a small problem, but running the code multiple times doesn't always solve it... strange

```
With Inference:

|-----
|- Solution:
|-
|- Assignment:
|-   Section ('Monday', 1800)
|-       - Harold Reynolds
|-       - Justin Petrarca
|-       - Marilyn Seat
|-       - Alexander Yutzy
|-       - *Krista Hays
|-   Section ('Friday', 1000)
|-       - Carol Hazel
|-       - Steven Leeper
|-       - Joe Douglas
|-       - Goldie Simmons
|-       - *Brent Schaeffer
|-   Section ('Sunday', 1800)
|-       - Timothy Lennard
|-       - Jennifer Petrovic
|-       - Todd Huggins
|-       - Aaron Dale
|-       - *Allen Facundo
|-   Section ('Wednesday', 1700)
```

```
|-       - Thomas Gray
|-       - Norma Beard
|-       - Elizabeth Faiella
|-       - Wanda Alix
|-       - *Javier Panning
|-   Section ('Tuesday', 1400)
|-       - Timothy Mercer
|-       - Elizabeth Young
|-       - Juanita Jenkins
|-       - Jeff Walker
|-       - *Melvin Heath
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: None
|- Time Taken: 973.219 ms
|- Values Checked: 66
|- Calls to Backtracking: 26
|-----
```

Without Inference:

```
|-----
|- Solution:
|-
|- Assignment:
|-   Section ('Monday', 1800)
|-       - Harold Reynolds
|-       - Justin Petrarca
|-       - Marilyn Seat
|-       - Alexander Yutzy
|-       - *Krista Hays
|-   Section ('Friday', 1000)
|-       - Carol Hazel
|-       - Steven Leeper
|-       - Joe Douglas
|-       - Goldie Simmons
|-       - *Brent Schaeffer
|-   Section ('Sunday', 1800)
|-       - Timothy Lennard
|-       - Jennifer Petrovic
|-       - Todd Huggins
|-       - Aaron Dale
|-       - *Allen Facundo
|-   Section ('Wednesday', 1700)
|-       - Thomas Gray
|-       - Norma Beard
|-       - Elizabeth Faiella
```

```
|-       - Wanda Alix
|-       - *Javier Panning
|-   Section ('Tuesday', 1400)
|-       - Timothy Mercer
|-       - Elizabeth Young
|-       - Juanita Jenkins
|-       - Jeff Walker
|-       - *Melvin Heath
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: None
|- Time Taken: 199.615 ms
|- Values Checked: 66
|- Calls to Backtracking: 26
|-----
```

I have also implemented a more tetris like solution to the circut problem. See `circut_EXTRA.py`.

`With Inference:`

```
|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (5, 2), (1, 2), (3, 2), (2, 1), (2, 1)
|- Assignment:
|-   eeeffd..gg
|-   eeecfdaaag
|-   bbbccccccg
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
|- Time Taken: 34.741 ms
|- Values Checked: 9
|- Calls to Backtracking: 8
|-----
```

`Without Inference:`

```
|-----
|- Solution:
|-
|- Variables: (3, 1), (3, 1), (5, 2), (1, 2), (3, 2), (2, 1), (2, 1)
|- Assignment:
|-   eeeffd..gg
|-   eeecfdaaag
|-   bbbccccccg
|- Variable Heuristic: VarHeuristic.DEGREE_TIEBREAKER
|- Value Heuristic: ValHeuristic.LCV
```

```
|- Time Taken: 134.064 ms
|- Values Checked: 276
|- Calls to Backtracking: 18
|-----
```

The solution for a tetris-like circut puzzle.

This is accomplished by the technique noted in the description above of checking if any of the realized locations overlap. I could speed this up by checking the same rectangular check on each block. That would be effective for larger graphs.