Lauren Kidman
COSC76: Artificial Intelligence 24F
18 November 2024

Report for PA4: CSP

Table of Contents:

The Assignment + *README

In this programming assignment, you will write a general-purpose constraint solving algorithm, and apply it to solve different CSPs. The learning objective is to implement the methods to solve the CSP, discussed in class.

*To run the CSP PA in Python, make sure all five files are in the same folder; run AustralianMapCSP.py and CircuitBoardCSP.py to test results, altering the heuristic and inference binary statements with True/False to experiment with their impacts on the results.

ConstraintSatisfactionProblem, i.e. General CSP solver:

---

1. Description

My general CSP solver is designed to handle a variety of constraint satisfaction problems–instead of being tailored to one specific problem at a time. At its core is a recursive backtracking algorithm, which assigns values to variables while enforcing constraints to ensure the assignments are valid. The majority of the solver was

implemented based on the pseudocode and principles discussed in class, as well as the course textbook. The main function, backtracking_search, initiates the recursive process to solve the CSP. The recursive function call_backtrack attempts to assign values to each variable one by one. If a complete and valid assignment is found, it returns the solution; otherwise, it backtracks to explore other possibilities. This method relies on helper functions to select variables, order values, check constraints, and apply inferences, which I implemented as they came up while coding backtracking.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Backtracking pseudocode from Lecture 10 slides

As per the assignment instructions, to improve the efficiency of the backtracking algorithm, I implemented several heuristics and an inference method:

Minimum Remaining Values (MRV): Chooses the variable with the fewest legal values left in its domain

Degree Heuristic: If there is a tie in MRV, selects the variable that is most constrained by other variables

Least Constraining Value (LCV): Orders the values in the domain to minimize the impact on the remaining variables, therefore reducing the likelihood of needing to backtrack

AC-3 Inference: Enforces “arc consistency” across the CSP, helping to prune the search space by removing values from variable domains that are inconsistent with constraints

I wanted my solver to be structured so that these heuristics and inference techniques can be easily toggled on or off; thus I used boolean parameters passed to the functions, which can be changed in each CSP's testing file. This design made it extremely easy for me to experiment and perform testing without altering the general solver code structure.

A significant challenge I faced was designing the is_legal function to be flexible enough to handle different CSPs. Initially, is_legal was specifically tailored to the map coloring problem, where the primary constraint was that neighboring regions could not have the same color. However, when I began implementing the circuit board problem, it became clear that each problem had its own unique constraints that did not translate into a general solver very well. The circuit board problem, for instance, requires checking for overlap between components, which involves four parameters instead of the two needed for map coloring.

To address this, I designed each CSP problem (such as map coloring and circuit board layout) to have its own constraint-checking function instead of including it in the general solver. I then introduced a flag system within the general solver to identify the type of problem being solved. The is_legal function uses this flag to determine which constraint logic to apply. Specifically, the flag (problem_type) is set to values like 'map_coloring' or 'circuit_board,' and the is_legal function uses the appropriate logic based on this flag. This structure also makes my general solver easily extendable should I want to test new CSP problems in the future–I can just add another flag.

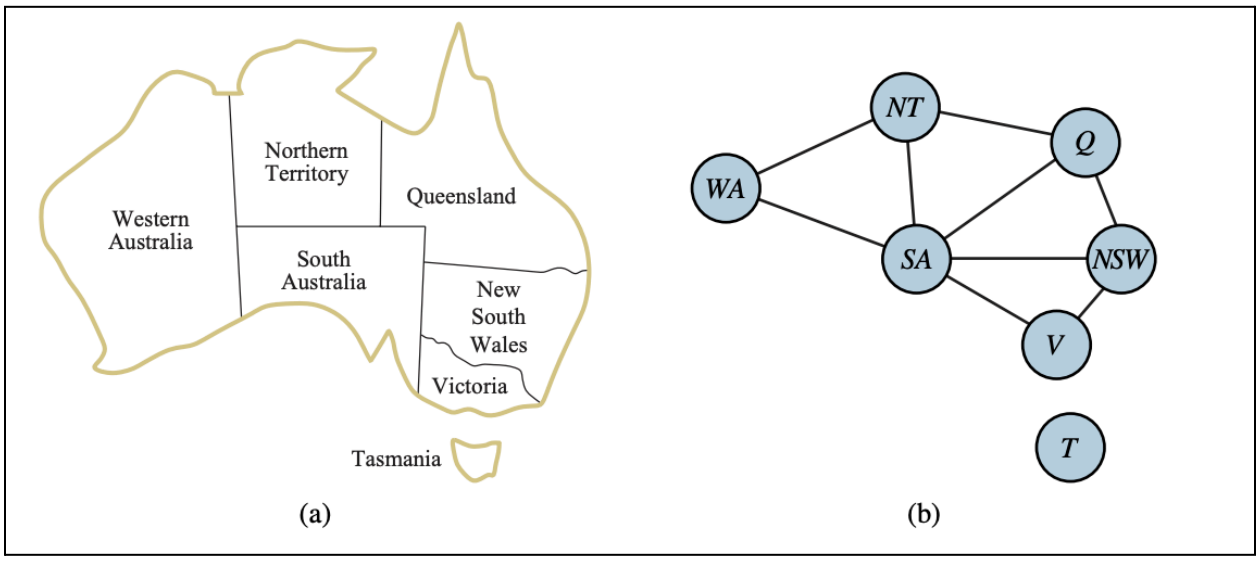
Map Coloring Problem:

---

1. Description

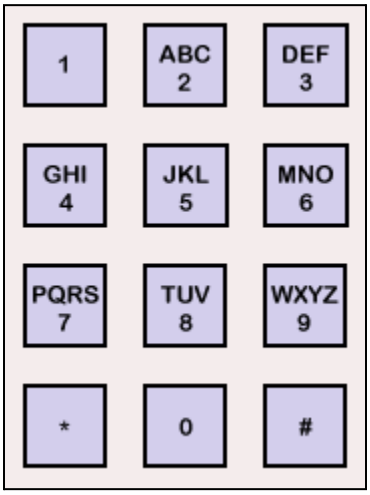
The majority of the work for this PA rides in the general CSP solver; from there, making the tailored CSP problem files, such as the Map Solver, was very straightforward.

The Map Coloring Problem is a classic example of a constraint satisfaction problem (CSP) where the goal is to color regions of a map such that no two adjacent regions share the same color. In my implementation, I specifically focused on the map of Australia as outlined by the textbook, which consists of seven regions: Western Australia (WA), Northern Territory (NT), South Australia (SA), Queensland (QU), New South Wales (NSW), Victoria (VI), and Tasmania (T). Each of these regions needs to be assigned one of three colors: red, green, or blue–making them the variables of this problem.



Graphic taken from AIMA textbook Chapter 6

In my AustralianMapCSP class, I needed a way to represent each region using numerical identifiers in order to ensure the general CSP solver could work solely with numbers, instead of a combination of characters and numbers. Also, I needed this method to be specific enough that there was no risk of regions being classified as the same number. Thus, to convert the region names to numbers, I used the number-to-letter format of a phone dial (e.g., WA is 92, NT is 68, etc.).



Furthermore, the domain for each region is a list of the three possible colors, also represented as integers (1 for red, 2 for green, and 3 for blue). I hard-coded the constraints so that neighboring regions cannot share the same color, and these constraints are enforced using adjacency relationships, such as WA cannot have the same color as NT or SA.

As mentioned, I created an is_constraint_satisfied method specific to the Map Coloring Problem, which checks if two neighboring regions have different colors. Finally, The AustralianMapCSP class includes a format_nums_to_text function that converts numerical assignments into readable region and color names (i.e. moving the results back from number form to text) in order to make the output intuitive and easier to understand.

2. Evaluation

My implementation of the map coloring problem using the general CSP solver is successful in generating valid solutions–upon every run it quickly produces a correct configuration of colors without running into any errors.

Some interesting behavior that the Map Solver demonstrates is in terms of efficiency when heuristics and inferences are applied–the performance metrics, such as the number of states visited and execution time, vary significantly depending on the combination of heuristics and inferences used, in fact increasing when heuristics/inference are used instead of decreasing. When all heuristics and inferences are turned off (use_mrv = False, use_lcv = False, use_degree = False, use_ac3 = False), the solver performs well on the Australian map coloring problem, visiting a relatively small number of states and completing in negligible time. For example, in one run, the solver visited 11 states and completed in 0.0000 seconds (which was expected, given the relatively small size and simplicity of the map coloring problem). However, when all heuristics and inferences are enabled (use_mrv = True, use_lcv = True, use_degree = True, use_ac3 = True), the solver actually visited 15 states and took slightly longer (0.0001 seconds) to complete.

```
Time without heuristics/inference: 0.0000 seconds
{'WA': 'Red', 'NT': 'Green', 'SA': 'Blue', 'Q': 'Red', 'NSW': 'Green', 'V': 'Red', 'T': 'Red'}
States visited:  11


Time with heuristics/inference: 0.0002 seconds
{'SA': 'Red', 'NT': 'Green', 'NSW': 'Green', 'WA': 'Blue', 'Q': 'Blue', 'V': 'Blue', 'T': 'Red'}
States visited:  15
```

I do not feel this is an issue with my code implementation, as my code always outputs a correct answer that abides by the constraints; rather I think the increased number of states visited and the longer runtime suggest that the heuristics and inferences are not always beneficial for this problem. In playing around with the settings, I found that the number of states visited increased when use_mrv was activated (on its own). The MRV heuristic selects the variable with the fewest legal values left, which is intended to minimize future conflicts; however, in the map coloring problem, this approach might

lead to suboptimal decisions early on, causing the solver to explore more states overall–in short, the map's constraints may not be complex enough to benefit from MRV, and the heuristic might lead to unnecessary backtracking. Furthermore, I found the execution time increased when use_ac3 was activated (on its own). As discussed in class, while the AC-3 algorithm can greatly reduce the search space in some cases, it also introduces a significant computational cost/increased time complexity–running AC-3 involves iterating over all arcs and checking for consistency, which can be time-consuming. Use_lcv and use_degree appeared to have no/minimal impact on either testing quantities.

3. Discussion Questions

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

As mentioned above, when heuristics and inference were disabled, the solver performed quite efficiently. The number of states visited was low (11 states in one example), and the execution time was extremely fast (0.0000 seconds). This performance is likely due to the simplicity and small scale of the map coloring problem, which does not necessarily require advanced optimization techniques to solve efficiently (I can do it in my head easily without even needing the code algorithm). When all heuristics and inference were enabled, however, the solver exhibited somewhat counterintuitive behavior. The number of states visited increased to 15, and the runtime increased slightly (0.0001 seconds). While these increases are relatively minor, they show that the heuristics and inference in this case did not improve performance, instead actually making it perform worse; the solver works efficiently without these optimizations, which highlights to me how the effectiveness of heuristics and inference methods is problem-dependent.

Results playing around with the toggling shown below:

```
No Heuristics or Inference: 0.0000 seconds
{'WA': 'Red', 'NT': 'Green', 'SA': 'Blue', 'Q': 'Red', 'NSW': 'Green', 'V': 'Red', 'T': 'Red'}
States visited:  11


All Heuristics and Inference: 0.0001 seconds
{'SA': 'Red', 'NT': 'Green', 'NSW': 'Green', 'WA': 'Blue', 'Q': 'Blue', 'V': 'Blue', 'T': 'Red'}
States visited:  15


Heuristics False, Inference True: 0.0001 seconds
{'WA': 'Red', 'NT': 'Green', 'SA': 'Blue', 'Q': 'Red', 'NSW': 'Green', 'V': 'Red', 'T': 'Red'}
States visited:  11


Heuristics True, Inference False: 0.0000 seconds
{'SA': 'Red', 'NT': 'Green', 'NSW': 'Green', 'WA': 'Blue', 'Q': 'Blue', 'V': 'Blue', 'T': 'Red'}
States visited:  15


MRV + Degree Heuristics True, LCV + Inference False: 0.0000 seconds
{'SA': 'Red', 'NT': 'Green', 'NSW': 'Green', 'WA': 'Blue', 'Q': 'Blue', 'V': 'Blue', 'T': 'Red'}
States visited:  15
```

Ultimately, while the heuristics + inference do not necessarily prove they actually improve performance in a given problem, in all scenarios my program is able to find a correct solution when one exists.

Circuit Board Problem:

---

1. Description

The Circuit Board Problem is a CSP where the objective is to arrange rectangular components on a 2D circuit board in such a way that no components overlap/occupy the same space on the board. Each component is defined by its width and height, and the board itself has a fixed width and height. The variables for this CSP are the (x, y) coordinates of the lower-left corner of each component. The domains for each variable are all possible positions on the board where the component could fit without extending beyond the board's boundaries.

To implement the Circuit Board Problem, I once again utilized my general CSP solver, which uses backtracking and toggled heuristics + inference to explore possible arrangements. However, unlike the simpler constraints of the map coloring problem, the Circuit Board Problem requires a more complex constraint-checking mechanism. Specifically, the is_constraint_satisfied function checks for overlap between

components by comparing the positions and dimensions of two components to ensure they do not occupy the same space, requiring four input variables rather than two.

I also created a function, format_nums_to_text, to visualize the board using ASCII art. Each component is represented by a unique letter, which I determined by adding the index of each particular component in the list of components when converting from number to letter, making it easier to see the layout and determine whether the solution is valid.

2. Evaluation

The Circuit Board Problem provided a more complex and spatially constrained challenge for the CSP solver; still my circuit board testing consistently outputted a correct board with no overlaps quickly every time, which I believe shows its effectiveness and accuracy.

Like with the Map solver, I observed that using heuristics and inference often resulted in worse performance. The Circuit Board solver handled small and medium board cases efficiently, visiting fewer states and solving the problem quickly. However, even for these relatively simple configurations, enabling heuristics and inference increased the number of states visited and the runtime.

For larger boards, the solver generally required significantly more states and time to reach a solution, demonstrating their increased complexity. Yet still with large and ‘huge’ boards, using heuristics and inference drastically increased both the number of states visited and the runtime. In fact, for the huge board, the number of states nearly doubled, and the runtime increased substantially. The consistently worse performance when using heuristics and inference (compared to not using them) ultimately is proving to me the added complexity of computing the heuristics seems to outweigh any benefits they provide.

```
No heuristics/inference -- states visited:  2692


Huge Board with all heuristics/inference: 0.3444 seconds
PPPQQQQQWWWW..FFFFFF
PPPQQQQQ.SS...FFFFFF
PPPQQQQQ.SS...FFFFFF
PPPRRRRNNSS...FFFFFF
PPPRRRRNNGG....JJJJJ
PPPRRRRNNGG....JJJJJ
MCCRRRRNN......JJJJJ
MCCBBBBBUUU....JJJJJ
MCCBBBBB.......JJJJJ
MCCAAAKKKK......EEEE
MCCAAAKKKK......EEEE
MCCAAAIII.......EEEE
M..AAAIII.......EEEE
TTTTTTIII.......EEEE
TTTTTT.LLLLLL..VVVVV
OOOOOOOLLLLLL..VVVVV
OOOOOOOLLLLLL..VVVVV
HHHHHHHHDDDDDDDVVVVV
XXXXXXXXDDDDDDDVVVVV
XXXXXXXXDDDDDDDVVVVV
None
With all heuristics/inference -- states visited:  4611
```

3. Discussion Questions

a. 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.

As I mentioned in Description, in the Circuit Board Problem, 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) consists of all possible positions where the component can be placed and fit completely within the boundaries of the board. The position of a component is represented by the coordinates of its lower left corner; thus, to ensure that the component does not extend beyond the edges of the board, the possible x-coordinates for the lower-left corner range from 0 to n-w, and the possible y-coordinates range from 0 to m-h. You have to subtract w and h in order to account for the right side and the top of the component from crossing the boundaries of the board. Thus:

$$0 <= x <= n-w \text{ and } 0 <= y <= m-h$$

b. *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 way to define a constraint that ensures no two components overlap is to ensure the rightmost side of one board does not overlap with the leftmost side of the other (and vice versa), and the bottom of one board does not overlap with the top of the other. Mathematically, this looks like:

x1 + width1 <= x2 (x1 is to the left of x2)
or x2 + width2 <= x1 (x2 is to the left of x1)
or y1 + height1 <= y2 (y1 is below y2)
or y2 + height2 <= y1 (y2 is below y1)

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

My CSP solver is designed to handle general constraints using integer values, which is why I have character to number conversions for both the Map Coloring and Circuit Board problems. For the circuit board in particular, how I handled ensuring numbers were inputted into the solver was quite simple:

Variables: The solver is passed a list of components (the variables). Thus, the variables are referenced by using the unique integer index of each particular component as it occurs in the list of all components. For example, if there are three components, they are represented as 0, 1, and 2

Domains: The domain of each variable (i.e. component) is a list of all valid (x,y) coordinate pairs, ensuring the component fits on the board. Though coordinates are already integers, the pairs are particularly stored as tuples so that the CSP solver can process

Constraints: The constraints between components are defined also using the integer indices that define the variables. For instance, if component 0 cannot overlap with component 1, this relationship is recorded using the indices 0 and 1