Adam Nelson-Archer

Dr. Eick

COSC4368

20 February 2024

# Generalized Constraint Satisfaction Analysis

The strategy employed for solving the constraint satisfaction problem revolves around a combination of recursive backtracking, forward checking, and domain sorting. The recursive backtracking algorithm explores possible assignments for the variables, checking them against constraints. Forward checking is used alongside backtracking to preemptively eliminate domain values that would lead to a violation of constraints, thereby pruning the search space and reducing the number of necessary variable assignments. Finally, the pruned domains are sorted in by how "constricting" each value is. All of this together creates a program that is highly optimized for finding valid solutions. Because of the hierarchical design of the problem, the program also calculates all possible results for a set of constraints and uses those results as the domain for the following problem.

My program is highly generalized. You can input any set of constraints, any set of variables, and it will run the same way. I specially crafted the program to avoid any hard coding, and a constraint can be added, removed, or modified easily. Information on how to do can can be found in the ReadMe file at the bottom of this report. Realistically, any type of constraint satisfaction problem could be easily implemented, as I did myself with the two different "final version" files. I wanted to create a "layered" solution which utilized different solutions to

generate a new result, and I had to make my program a lot more bulky to do so. So my pain.py file is my main, efficient, optimized program, and ProgramLauncher.py is my version that sets up problems A, B and C to feed into each other. Simply run ProgramLauncher if you would like to see the hierarchical version. I have posted all files to my Github, along with my ReadMe file. If you wish to see them, they are here: link to github.

Pseudocode for my program:

```
def forwardCheck(assignment, domains, constraints, var_assigned):
    # check if the constraint involves the variable that was just assigned
    if var_assigned in vars_involved:
      # Determine the number of unassigned variables involved in this constraint
      unassigned_vars = [var for var in vars_involved if var not in assignment]
          # Check if the constraint is satisfied with this hypothetical assignment
          if not constraint(hypothetical_assignment):
            updated_domains[unassigned_var].remove(value)  # Prune value if constraint fails
        if not updated_domains[unassigned_var]:  # If domain becomes empty, fail
          return None
  return updated_domains

def solve(variables, domains, constraints, assignment, solutions, nva):
    if assignment is complete:
        return solution
    for each var in variables not yet assigned:
        do forward_check, if pass:
            sort domains by LCV #(least constraining variable)
            for each value in domain of var:
                if value is consistent with constraints:
                    add {var: value} to assignment
                    nva += 1
                    result = solve(...)  # Recursive call with updated parameters
                    if result is not failure:
                        return result
                    remove {var: value} from assignment
    return failure

# main method here
variables = a,b,c,d,e,f...
constraints = ['''list of constraints here''']
domains = [...] #set variables with a list of 1 - 120
solutions = []
nva = 0
solve(variables, domains, constraints, {}, solutions, nva)
```

This is a simplified version of my program, but I attempted to preserve as much original functionality as possible. The main method is where all variable modification occurs, as well as the constraint definitions. All of this information is sent to solve(), which represents my main function. Solve() looks at each unassigned variable, picks the next value it can take, and runs it through forward checking. Forward checking determines if an input will lead us to an impossible variable at a later path with a different function. After forward checking, we sort the domain to get the least constraining variables first. To sort this way we run each value through forward checking for each unassigned variable, and if any variable is not usable, it is added to the end of the domain. Once all variables have assignments, the final result is returned to the user.

In my program, I took the results of program A and fed them into the starting domains for problem B. You can see the specifics of this implementation inside of my ProgramLauncher.py, but I basically just stored all results into a set, and used that to populate starting domains.

Here were the final results of my program::

```
Problem A ::
number of constraints: 5
selected variables:  ['A', 'B', 'C', 'D', 'E', 'F']
1st solution found: {'A': 15, 'B': 8, 'C': 7, 'D': 19, 'E': 7, 'F': 38}
NVA at first solution: 1944

Problem B ::
number of constraints: 12
selected variables:  ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
1st solution found: {'A': 111, 'B': 20, 'C': 17, 'D': 67, 'E': 10, 'F': 11, 'G': 1, 'H': 120, 'I': 3, 'J
': 99}
NVA at first solution: 17344

Problem C ::
number of constraints: 17
selected variables:  ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M']
1st solution found: {'A': 111, 'B': 20, 'C': 17, 'D': 67, 'E': 10, 'F': 11, 'G': 1, 'H': 120, 'I': 3, 'J
': 100, 'K': 5, 'L': 50, 'M': 2}
NVA at first solution: 17732
```

(1,944 for Problem A, 17,344 for Problem B, and 17,732 for problem C)

These results were satisfactory, as I had outperformed the brute force method by many orders of magnitude. Note how problem C had only ~300 more assignments than problem B. This is because of the way my forward checking works, I only evaluate a with forward checking if there is some constraint that is defined where the variable in question is solely undefined. Meaning, at the beginning, when we have a lot of undefined variables, I chose not to forward check as it produces minimal optimization and requires a massive computational overhead. The large jump from assignments in problem A to problem B is because we are computing every possible solution for problem A, leaving only a few hundred calculations left for problem B. In my main program, doing problem B alone uses around ~13,000 assignments, which is better, but this is just because of a quirk in the solution. Most of the final variable assignments are low numbers, meaning we access them quickly (we have no real need, in this problem set, to access all solutions for problem A). I chose to implement this mechanic anyways because for most problems this would be heavily optimizing.

I set up my program to do well with a good amount of pre-analyzing. I ran many different models before finding one that was effective with this data set. I only prune domains once we have several variables defined, for example. Given our constraints, it is difficult to prune domains with several undefined variables. In most CSP's, it would be effective to prune variables from a single assignment, but I was able to look at the constraints and modify my problem accordingly. I also chose not to sort variables in order of domain size, because I was sorting domains by LCV, meaning the size of the domain mattered a lot less than the order of its values.

# References

Wikipedia, Constraint Satisfaction -

https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

GeeksForGeeks, Constraint Satisfaction -

www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/

AiSpace, Constraints (good website)-

https://aispace.org/constraint/

KevinBinz, Einstein Constraint Matrices (looked for unique solutions)-

https://kevinbinz.com/2017/08/01/satisfiability-zebra-puzzle/

ChatGPT, GPT4 (I gathered general CSP info., you can read my conversation here) -

https://chat.openai.com/share/a1d38ed3-e40a-49f3-9f33-9368363f485d

README::

**You can find a better, formatted ReadMe on my github, linked [here]**

Generalized Constraint Satisfaction

- This code attempts to solve a highly generalized constraint satisfaction problem by use of recursive backtracking and heavy domain pruning.

The pruning is in two parts - forward looking, and sorting the domains in order of least to most constricting.

Almost all of the program customization is contained in these lines::

```
variables = ['A', 'B', 'C', ...]
Vrange = (1,120)
assignment = {}
constraints = [...]
numConstraints = 5  # Number of constraints to actually use
numVariables = 6   # Number of variables to actually use
```

Variables are characters, assign them into the variables array. Vrange is two numbers, you input the min and max for any value a variable can be. Assignment{} is where the final results are stored, you can input values here if you wish to predetermine a variable value. Example usage: `assignment = {'A': 5, 'B': 5}`

To use this program, constraints need to be coded into the program. They are all in the format of :: `(lambda x: x['A'] == x['B']**2 - x['C']**2, ['A', 'B', 'C'])`. This is a tuple where the first part is the function/constraint itself and the second part names each variable that this constraint acts on.

You can give your variables a min/max by modifying `Vrange(_, _)`, where the first number is the min and the second is the max. Currently the range for all variables is 1-120.

You can then decide how many of your constraints you want to use, and how many variables you select. If a constraint is selected that contains a nonexistent variable, it will not be applied, so you do not need to worry about input validation on that level.

There are two .py files here, "`LauncherVersion.py`" only exists to fulfill specific requirements I was given that this program needed to satisfy. It breaks the problem up into 3 parts, and solves for 4-5 variables at a time (all solutions), and uses those domains to create the next "level"of solutions. This is a good solution, but only works for constraints that are applied hierarchically.