Task 2 - Solving Discrete Constraint Satisfaction Problems TBDL

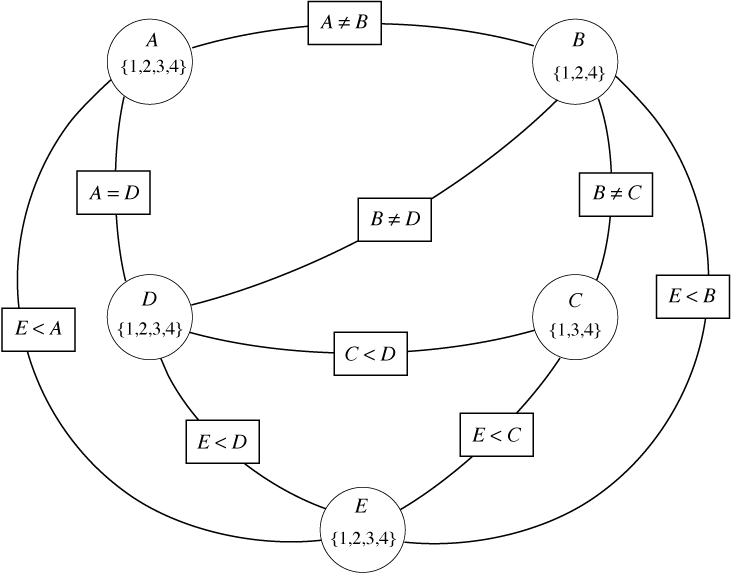


Fig. 2: Example of Constraint Graph

Write a program which finds solution to the following 3 hierarchically organized[[1]](#footnote-1) constraint satisfaction problems, involving 15 variables {A, B, C, …, N, O, P} which can take integer values in {1, …, 125}.

1. Problem A: Find a solution to the constraint satisfaction problem involving the six variables A, B, C, D, E and F and constraints C1, …, C4:
   * (C1) A=B+C+E+F
   * (C2) F> E-B
   * (C3) D=E+F+21
   * (C4) D\*\*2=E\*E\*A + 694
2. Problem B: Find a solution to the constraint satisfaction problem involving ten variables A, …, J which satisfy constrains C1, …, C9:
   * (C5) H\*J+E\*16=(G+**I**)\*\*2 -48
   * (C6) A-C=(H-F)\*\*2+4
   * (C7) 4\*J=G\*\*2+7
   * (C8) H+**I**<D
   * (C9) E\*\*2 < G\*\*2 + J\*\*2
3. Problem C: Find a solution to the constraint satisfaction problem involving 16 variables A, …, P which satisfy constrains C1, ..., C16:
   * (C10) 2\*M=K\*\*2 -A\*8
   * (C11) (N-**O**)\*\*2 = (F-J)\***O**\*2+360
   * (C12) N\*\*2-135=M\*J+A\*\*2
   * (C13) (L+N)\*\*2+445=(B+F)\*(K+M+N+(A\*E))
   * (C14) L\***O**+2250=(A\*\*2)\*(G-E)
   * (C15) K\*\*3-900 = (**O**\*F\*A)+M\*\*2
   * (C16) (P-N)\*\*2-(P-**O**)\*\*2=A\*\*2+K\*L-M\*\*2-1425

Remark: In the above equations the letters ‘I’ and ‘O’ were put into bold face to avoid being mistaken as numbers 0 or 1. Moreover, the letter ‘J’ looks somewhat similar to the letter ‘I’ but to better distinguish the two letters ‘J’ is never in bold face.

Your program should contain a counter **nva** (“number of variable assignments) that counts the number of times an initial integer value is assigned to a variable or the assigned integer to the particular variable is changed; in addition to outputting the solution to the CSV also report the value of this variable at the end of the run, and develop an interface to call your program for CSP Problems A, B, or C. **Your program should return a solution or “no solution exists” and the value of nva after the program terminates**. Moreover, terminate the search as soon as you found a solution—do not search for additional solutions.

Submit a report which

* Gives a brief description of the strategy you used to solve the CSP
* Provides Pseudo Code of your CSP solver
* Explains the Pseudo Code in a paragraph or two
* Describes strategies (if you employed any) you employed to reduce the runtime of your program, measured by the final value of the variable nva.
* Conducting a mathematical pre-analysis to eliminate variables, to obtain additional ‘<’ or ‘>’ constraints to reduce search complexity or developing other problem complexity reduction strategies based on such a pre-analysis, helps to create an efficient solution. Describe the results of the pre-analysis you conducted, and how the results of this pre-analysis were used for reducing the search complexity.
* Explain how your program takes advantage of the hierarchical structure[[2]](#footnote-2) of the three CSP problems.
* Developing a generic program in the sense that its code could be reused to solve other constraint satisfaction problems which have a similar structure, but different constraints is expected. Include a paragraph presenting evidence why your program has this property and what you did to make your program ‘generic’.,

Moreover, submit the Source Code for the implementation in a separate file and instructions on how to run your code in a Readme File. Attach the Readme file as an appendix to your report.

Notes on grading:

* Sophisticated approaches that lead to lower complexities in solving the respective CSPs—measured by the final value of the variable nva—will get up to 30% higher scores compared to programs that use brute force approaches.
* Severe penalties will be assessed if the value of the variable nva is not properly computed.

**Strategy used to solve the CSP**

As we have seen in this course, any CSP problem can be broken down into three components – variables, domain, and constraints. In our program we wish to find what the possible values of our variables {A, B, C, D, …, P} are, given that they are within their domain and follow all the specified constraints.

Firstly, the program defines the variables and their domains for each problem a, b, c. Then I define the constraints, for problem a, C1 – C4, for problem b C5 – C9 and the remaining C10 – C16 respectively for problem c. This is done with the help of a function that defines all the constraints for each problem and returns True/False depending on whether the constraint is satisfied or not

Now, the most important part is to implement backtracking correctly.

The approach I follow is,

* + - Each variable is empty at first, so we must select a variable to assign a value to.
    - To do so, I define a function called *‘select\_unassigned\_variable’* that does what its name suggests by implementing Minimum Remaining Values heuristic approach. This selects the next unassigned variable with the smallest domain.
    - We now order the values in the domain of the current variable using least constraining value heuristic approach. This allows us to select the value of the variable that eliminates the least values in the remaining values’ domains.
    - Now we assign the selected variable to the ordered values, one at a time, recursively calling the backtracking algorithm with the new variable assignment. If a solution is found, return it, if no solution found, backtrack to the last variable assignment and check if the next value in the ordered domain is valid.

**Provide Pseudo Code of your CSP solver**

**domain** = range(1, 126)

**define** constraints for the CSP problem

**backtrack**(assignment, variables):

if len(assignment) == len(variables):

return assignment

**select\_unassigned\_variable**(variables, assignment):

for var in variables:

if var not in assignment:

return var

**order\_domain\_values**(var, assignment):

values = [i for i in range(1, 126)]

return values

**is\_consistent**(assignment):

for constraint in constraints:

if not constraint(assignment):

return False

return True

variables\_A = ['A', 'B', 'C', 'D', 'E', 'F'] //can fill this according to each problem as needed

assignment\_A = {}

solution\_A = **backtrack**(assignment\_A, variables\_A)

if solution\_A is not None:

print("Solution found:")

print(solution\_A)

print("Number of variable assignments:", len(solution\_A))

else:

print("No solution found.")

**Explain the Pseudo Code in a paragraph or two**

* I first define the domain for each variable to be integers from 1 – 125.
* Then, the ‘select\_unassigned\_variable’ function selects the next unassigned variable in the order A, B, C, D, E, …, P.
* The function ‘is\_consistant’ takes each of our variables with its value, and checks if it satisfies all the constraints relating to that variable.
* The backtrack function takes in a list of variable assignments and a counter ‘nva’ to account for the total number of variable assignments. If all variables have been assigned successfully, store that as a solution and increase nva by 1. Else, select the next unassigned variable (with the help of our ‘select\_unassigned\_variable’ function) and for its domain, check if the value satisfies all the constraints. If the value is satisfiable, I add this variable and its value to my list, and lastly, recursively call backtrack with the new list and nva.
* After the recursive call, if a solution is produced – output the solution. If no solution produced, remove the variable and its value from the list as these are incorrect.
* Call the backtrack function again with an empty list and nva is now 0.

To use this code for other CSP problems, all I would need to do is change the ‘is\_consistant’ function to take in different constraints relating to the problem(s) needed to be solved.

**Strategies employed to reduce the runtime of the program.**

At first, I tried using brute force to solve the homework. I started working on problem (a) and my approach was to use nested for loops checking if the variables are within their domains and if they satisfy all the constraints. When I ran the program, I quickly realized that what I implemented was quite inefficient as it took forever to run, more than 5 - 7 minutes, which is an extremely long run time. I realized that my program was wasting time iterating over all the possible combination of values for the variables within the given domain even when the constraints were not satisfied. I quickly discarded of that code and started again. This time I made use of generative recursion and backtracking, to implement my program as explained above. Having a validator (checking if the constraints are satisfied) in place with backtracking allows us to reduce the number of possible solutions (assignments) by eliminating any assignments that do not follow the constraints. Moreover, the program computes the solutions in a hierarchical manner which allows us to make use of our earlier computed solutions as constraints in the next problem we wish to find the solution for, reducing the search space at each level. This topic is further explained in this report.

**Mathematical Pre-analysis**

To mathematically obtain the solution to these constraints we can make use of simple algebraic manipulation and substitution.

C1: A = B + C + E + F

C2: F > E – F = **F + B > E**

Now, we want to substitute A into C4 -> D^2 = E^2\*(B + C + E + F) + 694

= E^3 + F*E^2 + (B + C)E^2 + E(B + C)F + EB*F + E*C*F + E^2*B + E^2*C - D^2 - 694 = 0

Now we have a cubic equation where we can find the values of its variables by applying the rational root theorem. The roots obtained are (+/-1, +/-2, +/- 347, +/-694) which relate to the possible values of E. Substituting E into the third constraint gives D = F + E + 21, substituting A and D into constraint 1 gives, **B + C + 2F + 21 = A**

Performing a mathematical pre-analysis, we have successfully reduced the search complexity of the problem since now we only need to find values for variables B, C, and F that will satisfy our newfound constraints -> **F > E – B** and **A + B + C + D + E + F = 237**.

Some of the solutions found for problem A are,

A = 30, B = 1, C = 3, E = 1, F = 25, D = 47

A = 29, B = 1, C = 4, E = 1, F = 23, D = 45

A = 28, B = 2, C = 5, E = 2, F = 21, D = 43

A = 27, B = 2, C = 6, E = 2, F = 19, D = 41

A = 26, B = 3, C = 7, E = 3, F = 17, D = 39…

Now if we try to perform a pre-analysis for problem b, we will find that it is quite difficult to eliminate variables and reduce the search complexity manually. Trying to derive a simpler relationship between the variables results in,

C5: H\*J+E\*16=(G+**I**)\*\*2 -48 -> H\*J + E\*16 + 48 = (G+I)^2 -> sqrt(H\*J + E\*16 + 48) = G + I

C5\_new: G = sqrt(H\*J + E\*16 + 48) – I

Substituting the above value of G in constraint (C7) allows us to obtain:

4J = (sqrt(HJ + E16 + 48) - I)\*\*2 + 7

-> 4J = HJ + E16 + 48 - 2Isqrt(HJ + E16 + 48) + I2 + 7 -> 3J - E16 - 55 = 2Isqrt(HJ + E16 + 48) - I2

= (3J - E16 - 55)2 = 4\*I2\*(HJ + E16 + 48) - 4I(3J - E16 - 55)sqrt(HJ + E16 + 48) + I\*\*4

= (3J - E16 - 55)\*\*2 - 4I2\*(HJ + E16 + 48) + I4

= 4\*I\*\*2(3\*J – E\*16 - 55 - sqrt(H\*J + E\*16 + 48))(3\*J – E\*16 - 55 + sqrt(H\*J + E\*16 + 48))

Moving on, we would now have to do further substitutions and simplifications to find the range of values that satisfy the above constraint. If we look at problem c, it will be an even longer process to find all the solutions as this problem contains even more complex constraints. Overall, we can see that it becomes very complicated and time consuming to compute the solutions in this manner, or even try to reduce the constraints. Therefore, it is not always best to try any solve these problems manually.

For problems with numerous and complex constraints we must make use of efficient constraint-solving algorithm involving backtracking, to generate the possible solutions to these CSP problems.

**Explain** **how the program takes advantage of the hierarchical structure of the three CSP problems.**

The program takes advantage of the hierarchical structure of the 3 CSP problems by solving them one at a time in a hierarchical manner. The solution found in problem (a) is used as constraints in problem (b), similarly, the solutions found for problem (b) are used as constraints for problem (c). As mentioned above, this allows us to reduce the search space at each step.

For example,

Say we find the values of A, B, C, D, E, F after solving problem (a). We can use these same values as constraints in our next problem (b), eliminating the need for us to compute values for variables A - F, reducing the search space at each step whilst also computing the value of the rest of the variables. We can use the same approach for problem (c). Additionally, this in turn helps increase the efficiency at which our program runs.

**Include a paragraph presenting evidence why your program has this property and what you did to make your program ‘generic’.**

As mentioned earlier in this report, my program takes advantage of the hierarchical structure of the CSP problem, and this is seen in the program’s implementation. The solutions from the previous problem are being used as constraints in the next problem. As long the problems are hierarchically structured, the program will be able to efficiently find solutions to each problem by repeatedly iterate over the list of problems solving them one at a time, hierarchically. With our backtracking algorithm, we can prune the search space by checking if the constraints are satisfied and backtrack to the place of conflict, that is, where a solution cannot be found. This makes the program ‘generic’ as it allows any number of hierarchically structured CSP problems to be solved in this manner. Moreover, using the previous problems solutions as constraints, with the backtracking algorithm, and the overall flexible implementation allows for us to use this program to solve any hierarchically designed constraint satisfactory problems.

In addition, I make use of a modular design approach. Here, each CSP is represented as a module taking in input, performing calculations, and producing output. The input are our variables, their domains and constraints and the output are the possible solutions. The user can easily make modifications and manipulate the modules according to the hierarchy of the CSP problem.

**Moreover, submit the Source Code for the implementation in a separate file and instructions on how to run your code in a Readme File. Attach the Readme file as an appendix to your report**

**Appendix**

READme.txt

This file is meant as instructions for my code. Unfortunately, I was unable to come up with a solution that produces the correct output. The main.py file runs but does not produce the desired output. I tried implementing the reported algorithm and this is what I came up with.

The file I submitted has 3 python files inside it,

1. main.py - with my entire source code for problem a, b, c

2. problem\_a.py - a second implementation for problem a

3. problme\_b.py - a second implementation for problem b

1. A solution of the higher numbered problem also represents a solution of the lower numbered problem! [↑](#footnote-ref-1)
2. If your approach uses solutions of a lower problem to solve the higher problem, e.g. uses solutions of problem A to solve problem B then the proper value for the variable nva should be computer by adding the cost of creating the solutions for A and the cost of finding a single solution for B based on the solutions obtained for A. [↑](#footnote-ref-2)