5 Constraint Satisfaction Problem

Constraint Satisfaction Problems(CSP) uses a factored representation of each state, with each state associated with a set of variables(X), each variable containing a value from its domain(D). The objective of the problem to find the value for each of the variable that satisfy all the constraints(C) of the problem.

A CSP comprise of three components:

1. Set of variables: $X: \{x_1, x_2, \ldots, x_n\}$
2. Set of domains(D) belonging to each variable $\{D_1, D_2, \ldots, D_n\}$ where $D_i = domain(x_i)$
3. Set of constraints(C) that specifies the valid combinations of values assigned to each of the variable.

In CSP the order of series of action has no effect on possible outcome, hence CSP problems are *commutative*. Let us consider an algorithm to solve CSP.

5.1 Backtracking Search

One useful algorithm to solve CSP is the Backtracking Search. Let us try to represent $4 - Queen$ puzzle as CSP, for that first we have to find variable, possible domain for each of the variable, and constraint to solve.

We can describe the CSP as follows:

1. Variables: $\{x_1, x_2, x_3, x_4\}$, where each x_i represent the variable for i^{th} column in 4×4 grid.
2. For each variable x_i: $D_{x_i} = \{1, 2, 3, 4\}$, D_{x_i} represent the value of the row, i.e. each variable x_i can possibly take a value of the row.
3. Constraints: Let NoAttack(x_i, x_j) return True if the queen at x_i cannot attack the queen at x_j, and False otherwise.
   (a) NoAttack(x_1, x_2)
   (b) NoAttack(x_1, x_3)
   (c) NoAttack(x_1, x_4)
   (d) NoAttack(x_2, x_3)
   (e) NoAttack(x_2, x_4)
   (f) NoAttack(x_3, x_4)

We can specify the constraints fully for each NoAttack(x_i, x_j) in a huge truth table as seen in Table 1.

Table 1: Specifying constraints on a huge truth table

x_1	x_2	NoAttack(x_1,x_2)
1	1	False
1	2	False
1	3	True
1	4	True
2	1	False
...	...	...

Our goal is to choose the values of all x so that none of the queens in their respective positions can attack each other.

Backtracking Search Algorithms

We first consider a naive algorithm that does not check for consistency of the variables in the process of assigning values to them. Algorithm 1 represents a naive Backtracking Approach for CSP.

Algorithm 1 BACKTRACKINGSEARCHNAIVE($prob, assign$)

```
if AllVarsAssigned(prob, assign) then
    if isConsistent(assign) then
        return assign
    else
        return failure
var ← PickUnassignedVar(prob, assign)
for value ∈ OrderDomainValue(var, prob, assign) do
    assign ← assign ∪ (var = value)
    result ← BacktrackingSearch(prob, assign)
    if result! = failure then return result
    assign ← assign \ (var = value)
return failure
```

Let us try to execute the Algorithm 1 step by step for $4 - Queen$ puzzle. Figure 1 represents the board overview of backtracking algorithm to solve $4 - Queen$ puzzle.

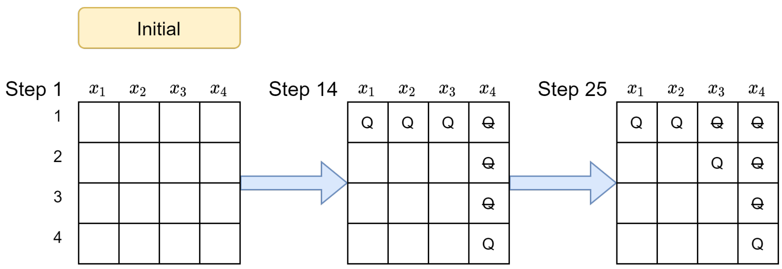

Figure 1: Board state at various steps of naive Backtracking Search.

Let $\underline{i.j}$ be the i^{th} functional call, and j^{th} operation in that functional call. The execution of the algorithm goes as follows:

1. <u>1.1</u> Pick variable x_1; <u>1.2</u> Set $x_1 = 1$
2. <u>2.1</u> Pick variable x_2; <u>2.2</u> Set $x_2 = 1$
3. <u>3.1</u> Pick variable x_3; <u>3.2</u> Set $x_3 = 1$
4. <u>4.1</u> Pick variable x_4; <u>4.2</u> Set $x_4 = 1$
5. <u>5.1</u> isConsistent(assign)=False, thus back track; <u>4.3</u> Set $x_4 = 2$
6. <u>5.1</u> isConsistent(assign)=False, thus back track; <u>4.4</u> Set $x_4 = 3$
7. <u>5.1</u> isConsistent(assign)=False, thus back track; <u>4.5</u> Set $x_4 = 4$
8. <u>5.1</u> isConsistent(assign)=False, thus back track

9. 4.6 Finished ‘for’ loop, thus back track; 3.3 Set $x_3 = 2$
10. 4.1 Pick variable x_4; 4.2 Set $x_4 = 1$
11. 5.1 isConsistent(assign)=False, thus back track; 4.3 Set $x_4 = 2$
12. 5.1 isConsistent(assign)=False, thus back track; 4.4 Set $x_4 = 3$
13. 5.1 isConsistent(assign)=False, thus back track; 4.5 Set $x_4 = 4$
14. 5.1 isConsistent(assign)=False, thus back track
15. 4.6 Finished ‘for’ loop, thus back track
16. $\cdots$

As shown in the above execution of the Algorithm 1 for $4 - Queen$ puzzle, the main drawback of the algorithm is to check at the end if the assignment is consistent or not. It assigns a possible value to each of the four variables, and then check at the end if the assignment is consistent, and if not, algorithm changes the value of one of the variable and repeat again. Therefore, Algorithm 1 would simply check for consistency with respect to all possible permutation of values 1 to 4 assigned to each of the four variables x_1 to x_4.

Improved Backtracking Search

To overcome the issue of Algorithm 1, a better approach would be every time the algorithm has to assign a value to a variable, it first checks to ensure that the assigned value is consistent with the previous assignments. Therefore, the improved Algorithm 2 assigns a position to the queen only if it can not be attacked by other queen.

Algorithm 2 BacktrackingSearch($prob, assign$)

```
1: if AllVarAssigned(prob, assign) then return assign
2: var ← PickUnassignedVar(prob, assign)
3: for value in OrderDomainValue(var, prob, assign) do
4:     if ValIsConsistentWithAssignment(value, assign) then
5:         assign ← assign ∪ (var = value)
6:         result ← BacktrackingSearch.(prob, assign)
7:         if result!=failure then return result
8:         assign ← assign \ (var = value)
9: return failure
```

Let us try to execute the Algorithm 2 step by step for $4 - Queen$ puzzle. Figure 2 represents the board overview of backtracking algorithm to solve $4 - Queen$ puzzle. The execution of the algorithm goes as follows:

1. 1.1 Pick x_1; 1.2 Set $x_1 = 1$
2. 2.1 Pick x_2; 2.2 Set $x_2 = 3$ as $\{1, 2\}$ is attackable
3. 3.1 Pick x_3; 3.2 Return $failure$ as $\{1, 2, 3, 4\}$ are attackable
4. 2.3 Set $x_1 = 4$ as $\{1, 2\}$ is attackable and $\{3\}$ had been assigned

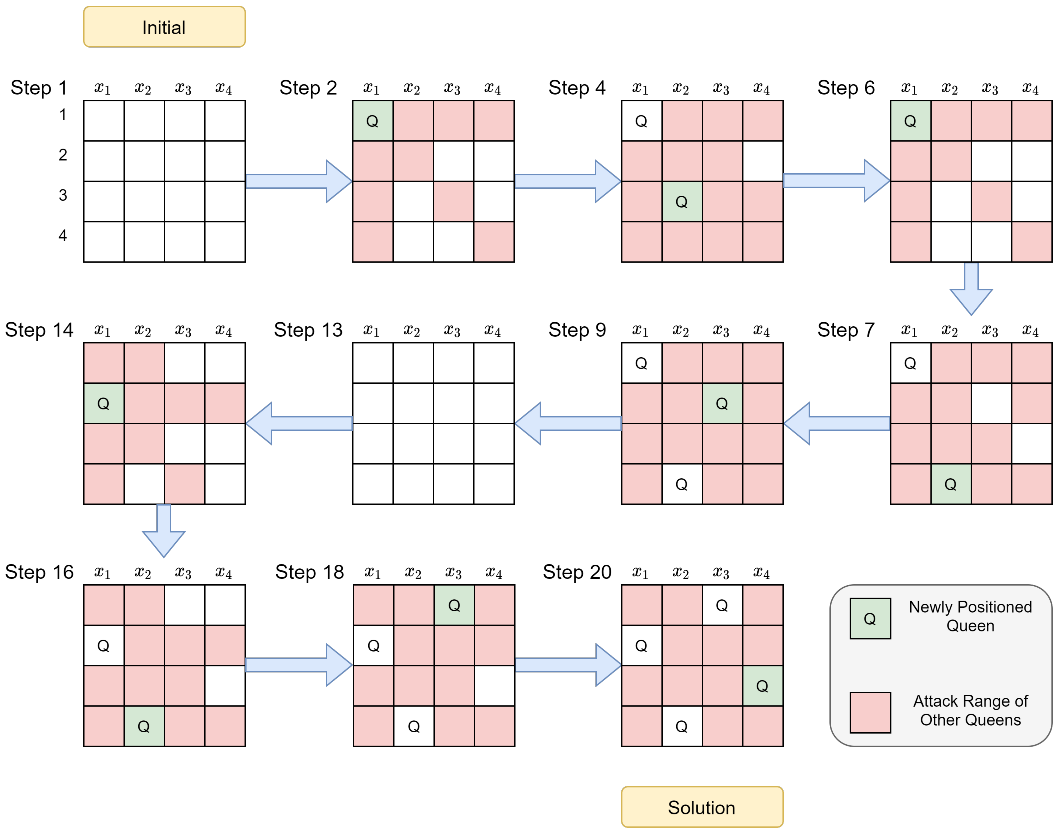

Figure 2: Board state at various steps of improved Backtracking Search

5. 3.1 Pick x_3; 3.2 Set $x_3 = 2$ as $\{1, 3, 4\}$ are attackable
6. 4.1 Pick x_4
7. 4.2 Return $failure$ as $\{1, 2, 3, 4\}$ are attackable
8. 3.3 Return $failure$ as $\{1, 3, 4\}$ are attackable and $\{2\}$ had been assigned
9. 2.4 Return $failure$ as $\{1, 2\}$ are attackable and $\{3, 4\}$ had been assigned
10. 1.3 Set $x_1 = 2$
11. 2.1 Pick x_2; 2.2 Set $x_2 = 4$ as $\{1, 2, 3\}$ are attackable
12. 3.1 Pick x_3; 3.2 Set $x_3 = 1$ as $\{2, 3, 4\}$ are attackable
13. 4.1 Pick x_4; 4.2 Set $x_4 = 3$ as $\{1, 2, 4\}$ are attackable
14. 5.1 Return $assignments$ because all variables are assigned

As seen from the execution of the Algorithm 2, checking consistency every time a value is assigned to variable improves the efficiency, and we reach goal state in less number of steps than with Algorithm 1.

5.2 Backtracking Search with Inference

An another idea to improve the efficiency of the Backtracking Search is that once the algorithm assigns a value to a variable x_i, it checks for all the constraints that x_i appears to *infer* the restrictions on rest of the variables and avoid picking values that are not consistent with previous assignments and possible future assignments.

We have a new function $\textsc{Infer}(prob, var, assign)$, that output a set of assignments that can be inferred based on the constraints of the problem, whenever a *value* is assigned to *variable*. If it is inferred that we can not assign a value to all variables consistently, the function returns a $failure$. In other words, the $\textsc{Infer}$ function helps to make inferences on what values other unassigned variables can take. In CSPs, assigning a value to a variable could lead to restrictions the value that other variables can take, which in turn could propagate the restrictions to other variables. Infer function aims to identify these restrictions and to use them to further restrict the search space.

Algorithm 3 represents the improved backtrack algorithm with inference.

Algorithm 3 BacktrackingSearch_with_Inference($prob, assign$)

if $\textsc{AllVariablesAssigned}(prob, assign)$ **then return** $assign$
$var \leftarrow \textsc{PickUnassignedVar}(prob, assign)$
for $value$ in $\textsc{OrderDomainValue}(var, prob, assign)$ **do**
    if $\textsc{ValIsConsistentWithAssignment}(value, assign)$ **then**
        $assign \leftarrow assign \cup (var = value)$
        $inference \leftarrow \textsc{Infer}(prob, var, assign)$
        $assign \leftarrow assign \cup inference$
        if $inference!=failure$ **then**
            $result \leftarrow \textsc{BacktrackingSearch.}(prob, assign)$
            if $result!=failure$ **then return** $result$
        $assign \leftarrow assign \setminus \{(var = value) \cup inference\}$
return $failure$

Below is the execution of Algorithm 3 for $4 - Queen$ problem:

1. <u>1.1</u> Pick x_1; <u>1.2</u> Set $x_1 = 1$
2. <u>1.3</u> $inference$ returned as $failure$, remove assignment $x_1 = 1$
3. <u>1.4</u> Set $x_1 = 2$
4. <u>1.5</u> $inference$ returned as $\{x_2 = 4, x_3 = 1, x_4 = 3\}$
5. <u>2.1</u> Return *assignments* because all variables are assigned

Inference Function Logic at Step 3 At step 3, because $x_1 = 1$, the inference function is able to deduce that the red squares in the Figure 3 are within a queens attacking range and no other queens can be placed there.

Then, the inference function looks at each and every constraint to infer restrictions on rest of the variables. (Note that there are many possible different orders in which constraints are considered).

- <u>Constraint 1:</u> $\textsc{NoAttack}(x_1, x_2)$
  - Observation from Figure 3: $x_1 = 1 \Rightarrow x_2 \notin \{1, 2\}$

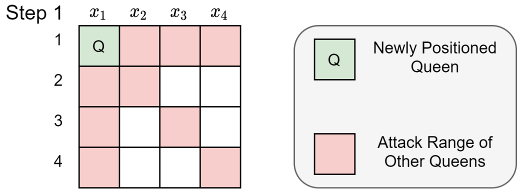

Figure 3: Board state at first step of Backtracking Search with inference

- Constraint 2: NoAttack(x_1, x_3)
  - Observation from Figure 3: $x_1 = 1 \Rightarrow x_3 \notin \{1, 3\}$
- Constraint 3: NoAttack(x_1, x_4)
  - Observation from Figure 3: $x_1 = 1 \Rightarrow x_4 \notin \{1, 4\}$
- Constraint 4: NoAttack(x_2, x_4)
  - Because $x_2 \in \{3, 4\}$,
  - If $x_2 = 3 \Rightarrow x_4 = 2$
  - If $x_2 = 4 \Rightarrow x_4 = 3$

  Note that we were not able to infer any restriction on x_4 or x_2 from the above step.
- Constraint 5: NoAttack(x_2, x_3)
  - Deduction: $x_2 \in \{3, 4\} \Rightarrow x_3 \notin \{3, 4\}$
  - Deduction: $x_3 \notin \{3, 4\} \wedge x_3 \notin \{1, 3\} \Rightarrow x_3 \notin \{1, 3, 4\} \Rightarrow x_3 = 2$
- Constraint 6: NoAttack(x_3, x_4)
  - Deduction: $x_3 = 2 \Rightarrow x_4 \notin \{1, 2, 3\}$
  - Deduction: $x_4 \notin \{1, 2, 3\} \wedge x_4 \notin \{1, 4\} \Rightarrow x_4 \notin \{1, 2, 3, 4\}$
  - Therefore,with no possible value to assign to x_4, $x_1 = 1$ is not going to lead to a solution, return failure and backtrack

Inference Function Logic at Step 5 At step 5, because $x_1 = 2$, the inference function is able to deduce that the red squares in the Figure 4 are within a queens attacking range and no other queens can be placed there.

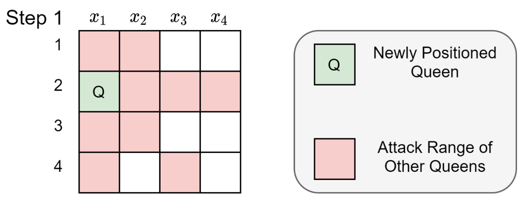

Figure 4: Board state at first step of Backtracking Search

Then, the inference function looks at each and every constraint to infer restrictions on rest of the variables.

- Constraint 1: $\textsc{NoAttack}(x_1, x_2)$
  - Observation from Figure 4: $x_1 = 2 \Rightarrow x_2 \notin \{1, 2, 3\} \Rightarrow x_2 = 4$
- Constraint 2: $\textsc{NoAttack}(x_1, x_3)$
  - Observation from Figure 4: $x_1 = 2 \Rightarrow x_3 \notin \{2, 4\}$
- Constraint 3: $\textsc{NoAttack}(x_1, x_4)$
  - Observation from Figure 4: $x_1 = 2 \Rightarrow x_4 \notin \{2\}$
- Constraint 4: $\textsc{NoAttack}(x_2, x_4)$
  - Deduction: $x_2 = 4 \Rightarrow x_4 \notin \{2, 4\}$
  - Deduction: $x_4 \notin \{2, 4\} \wedge x_4 \notin \{4\} \Rightarrow x_4 \notin \{2, 4\}$
- Constraint 5: $\textsc{NoAttack}(x_2, x_3)$
  - Deduction: $x_2 = 4 \Rightarrow x_3 \notin \{3, 4\}$
  - Deduction: $x_3 \notin \{3, 4\} \wedge x_3 \notin \{2, 4\} \Rightarrow x_3 \notin \{2, 3, 4\} \Rightarrow x_3 = 1$
- Constraint 6: $\textsc{NoAttack}(x_3, x_4)$
  - Deduction: $x_3 = 1 \Rightarrow x_4 \notin \{1, 2\}$
  - Deduction: $x_4 \notin \{1, 2\} \wedge x_4 \notin \{2, 4\} \Rightarrow x_4 \notin \{1, 2, 4\} \Rightarrow x_4 = 3$
  - Therefore,the INFER function returns $\{x_2 = 4, x_3 = 1, x_4 = 3\}$

The Backtracking search with an inference function, Algorithm 3, therefore becomes much more efficient. The overall efficiency however, is now strongly dependent on the efficiency of the INFER function which can be very costly to implement.

5.3 Implementation of Infer

Now, the important question is *How to implement the Infer functions?* Let us discuss important questions related to INFER functions:

Representation of data-structure It stores each of the variable x_i, and set of values that x_i is not supposed to take, such a set of values grows as INFER makes deductions based on the current assignment and constraints. For example: $x_1 \notin \{1, 2\}, x_2 \notin \{1, 3, 4\} \ldots$

Store the constraints The naive approach to store the constraints could be storing them as a truth-table described in Table 1, but we know that it is not efficient to use truth-tables for larger data; hence the more efficient way of storing the constraints is to come-up with some function that can returns true/false depending on the current values of the argument variables. Definitions of such a function depend on the application, for example for $n - Queen$ problem, such a function can be $f(x_1, x_2)\{if\{|x_1 - x_2| > 1\}$ return True else False$\}$.

ComputeDomain($\boldsymbol{x, assign, inference}$) In addition to above discussed questions, we also need an way to compute the (effective) domain of a variable x given an assignment and inference. We use a function $\textsc{ComputeDomain}(x, assign, inference)$ to return the domain of variable x given assignment and inference. Again, the definition of function COMPUTEDOMAIN is not fixed and mainly depends on the application and inference definition.

For example: if we have the assignment, $\{x_1 = 1\}$ and the inference, $\{x_1 \notin \{2,3\}\}$, then the ComputeDomain($x_1, assign, inference$) should output the domain of x_1 as $\{1\}$. In another example for the assignment, $\{x_2 = 1\}$ and the inference, $\{x_1 \notin \{2,3\}\}$ the ComputeDomain($x_1, assign, inference$) should output the domain of x_1 as $\{1,4\}$.

Algorithm 4 represents Infer function of Algorithm 3.

Algorithm 4 Infer($prob, var, assign$) function of Algorithm 3

```
inference ← ∅
varQueue ← [var]
while varQueue is not empty do
    y ← varQueue.pop()
    for each constraint C in prob where y ∈ Vars(C) do
        for all x ∈ Vars(C) \ y do
            S ← ComputeDomain(x, assign, inference)
            for each value v in S do
                if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ⊢ v] is satisfied then
                    inference ← inference ∪ (x ∉ {v})
            T ← ComputeDomain(x, assign, inference)
            if T = ∅ then return failure
            if S ≠ T then
                varQueue.add(x)
return inference
```

Example in N-queen problem Recall in the N-queen problem for Algorithm 3, at the start of the execution of the algorithm, the assignment $x_1 = 1$ was made and the inference function is then called on this assignment.

x_1 is pushed and popped, and all the constraints in the problem involving variable x_1 is iterated over a for loop, namely: NoAttack(x_1, x_2),NoAttack(x_1, x_3),NoAttack(x_1, x_4).

For the first constraint NoAttack(x_1, x_2) to be iterated, the inner for loop calls upon x in the constraint that is not x_1, which in this case is only x_2. The initial domain for x_2 stored in S is $\{1,2,3,4\}$. For each value in S, we check if there is a valid value for x_2 so that NoAttack(x_1, x_2) is satisfied. In this case the domain of x_1 is 1 because it has already been assigned that value. For example, when $x_2 = 2$, no valid value for $x_1 \in \{1\}$ exists so that the constraint is satisfied, therefore, $x_2 \notin \{1\}$ is added into inference. The same is done $x_2 = 2$ and $x_2 \notin \{2\}$ is added to inference. After that, the domain for x_2 is computed again with the updated inference, which returns set T as $x_2 \in \{3,4\}$, which is not an empty set, but is also not the same as S. Thus, x_2 is added into $varQueue$.

The same process is iterated for the other remaining constraints to add in values of x_3 and x_4 into $inference$.

When x_2 is popped from the $varQueue$, all the constraints in the problem involving variable x_2 is iterated over a for loop, namely: NoAttack(x_1, x_2), NoAttack(x_2, x_3), NoAttack(x_2, x_4).

For example sake, let us consider constraint NoAttack(x_2, x_3). The inner for loop then calls upon x in the constraint that is not x_2, which in this case is only x_3. The initial domain for x_3 stored in S is $\{2,4\}$. For each value in S, we check if there is a valid value for x_3 so that NoAttack(x_2, x_3) is satisfied. For example, when $x_3 = 4$, no valid value of $x_2 \in \{3,4\}$ exists so that NoAttack(x_2, x_3) is satisfied, so $x_3 \notin \{4\}$ is added into the constraint. After that, the domain for x_3 is computed again with the updated inference, which returns T as $x_3 \in \{2\}$, which is not an empty set, but is also not the same as S. Thus, x_3 is added into *varQueue*.

The algorithm continues on until it reaches constraint NoAttack(x_3, x_4), where the T for x_4 is am empty set, thus returning failure to the main backtracking algorithm.

Alternative Implementation of Inference

The computation required for INFER function can be expensive, and may potentially slow down the search. Some alternative implementation attempts to limit the depth of the inference in order to reduce computational cost. However, one important point to note here is this would also mean that the backtracking algorithm does not benefit from information that can be derived from deeper levels of inference.

The alternative implementations are as follows:

1. Don't add any variable to *varQueue* at each iteration. This is the equivalent of deleting Lines 13 and 14 of the Infer Algorithm 4.
    (a) This ensures that the inference will rule out only some of the values of variable.
    (b) This variant of inference is also known as *Forward Checking*.
    (c) Although forward checking can rule out many inconsistencies, it does not infer further ahead to ensure arc consistency for all the other variables.
2. Replace the $S \neq T$ condition in line 13, replace with $|T| = 1$.
    (a) This goes one step further from forward checking, and allows for further inferences for the variables that have one valid value in the domain.
    (b) Depending on the complexity of the application, the condition can be set to any particular value, e.g $|T| \leq 3$.

In addition to aforementioned different variants of implementation, there are many heuristics that are proved to efficient in backtracking-search algorithms:

Minimum Remaining Value Heuristic : In PickUnassignedVar function of the backtracking search always choose the next unassigned variable that have the smallest domain size. The idea behind this heuristic is to assign values to variables that is most likely to cause failure quickly, thus allowing for pruning of larger branches in the search tree.

Least Constraining Value Heuristic In OrderDomainValue function of the backtracking search always pick a value in the domain that rules out the least domain values of other neighboring variables in the constraint. This allows for the maximum flexibility for subsequent variable assignments. This heuristic is only relevant if we are interested in a single solution.

How Hard are Constraint Satisfaction Problems? Nevertheless, the ways described are only heuristics and not guaranteed to perform well in any kind of problem. In fact, CSPs are very hard to solve, belonging to a set of problems known in Computer Science as NP-Complete for which there are no known Polynomial time solutions to solve these problems but the answer to these problems can be checked in polynomial time. There are multiple variants of CSPs, and while most are NP-Complete, some can be solved in Polynomial time. Below lists some of the variants of CSPs.

There are some special variants of CSP as listed below:

1. Binary CSP: Every constraint is defined over two variables(NP-complete).

2. Boolean CSP(SAT): The domain of every variable is 0,1(NP-complete).

3. 2-SAT: Combination of Binary CSP and Boolean CSP, that is, every constraint is defined over two variables and the domain of every variable is 0,1(Polynomial time(P)-solvable).