Deploying a Constraint Satisfaction Framework

- Constraints are defined using a `Constraint` class.

- Each `Constraint` consists of the `variables` it constrains and a method that checks whether it's `satisfied ()`.

  The determination of whether a constraint is satisfied is the main logic that goes into defining a specific constraint-satisfaction problem.

- The default implementation must be overridden because we're defining our `Constraint` class as an abstract base class. Abstract base classes aren't meant to be instantiated; only the subclasses that override and implement their `@abstractmethod`s are used.
- Please refer:
  Information on typing module at :
  https://docs.python.org/3/library/typing.html

```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod
V = TypeVar('V') # variable type
D = TypeVar('D') # domain type
# Base class for all constraints
class Constraint(Generic[V, D], ABC):      # The variables that
the constraint is between
 def __init__(self, variables: List[V]) -> None:
 self.variables = variables
# Must be overridden by subclasses
   @abstractmethod
 def satisfied(self, assignment: Dict[V, D]) -> bool:
```

- The center piece of our constraint-satisfaction framework is a class called `CSP`.

- `CSP` is the gathering point for variables, domains, and constraints.

- In terms of its type hints, it uses generics to make itself flexible enough to work with any kind of variables and domain values (`V` keys and `D` domain values).

- Within `CSP`, the definitions of the collections `variables`, `domains`, and `constraints` are of types that you'd expect.

- The `variables` collection is a `list` of variables,

- `domains` is a `dict` mapping variables to lists of possible values (the domains of those variables), and

- `Constraints` is a `dict` that maps each variable to a `list` of the constraints imposed on it.

```
# A constraint satisfaction problem consists of variables of type V  # that  have  ranges  of  values  known  as  domains  of  type  D  and constraints  # that determine whether a particular variable's domain selection is valid

class CSP(Generic[V, D]):

  def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:

     self.variables:  List[V]  =  variables  #  variables  to  be constrained

self.domains:  Dict[V,  List[D]]  =  domains  #  domain  of  each variable

 self.constraints: Dict[V, List[Constraint[V, D]]] = {}

  for variable in self.variables:

     self.constraints[variable] = []

       if variable not in self.domains:

       raise LookupError("Every variable should have a domain assigned to it.")

 def add_constraint(self, constraint: Constraint[V, D]) -> None:

   for variable in constraint.variables:

   if variable not in self.variables:

   raise LookupError("Variable in constraint not in CSP")

  else:

     self.constraints[variable].append(constraint)
```

- The `__init__ ()` initializer creates the `constraints dict`.

- The `add constraint ()` method goes through all of the variables touched by a given constraint and adds itself to the `constraints` mapping for each of them.

- Both methods have basic error-checking in place, and raise an exception when a `variable` is missing a domain or a `constraint` is on a non-existent variable.

- A given configuration of variables and selected domain values satisfy the constraints The configuration is called `assignment`.

- We need a function that checks every constraint for a given variable against an assignment to see if the variable's value in the assignment works for the constraints. Here we implement a `consistent()` function as a method on `CSP`.

```
Check if the value assignment is consistent by checking all
constraints  # for the given variable against it
 def consistent(self, variable: V, assignment: Dict[V, D]) ->
bool:
 for constraint in self.constraints[variable]:
    if not constraint.satisfied(assignment):
        return False
 return True
```

- `consistent()` goes through every constraint for a given variable (it's always a variable that was newly added to the assignment) and checks if the constraint is satisfied, given the new assignment.
- If the assignment satisfies every constraint, `True` is returned. If any constraint imposed on the variable isn't satisfied, `False` is returned.
- This constraint-satisfaction framework uses a simple backtracking search to find solutions to problems.
- *Backtracking* is the idea that once you hit a wall in your search, you go back to the last known point where you made a decision before the wall, and choose a different path.
- The backtracking search implemented in the following `backtracking_search ()` function is a kind of recursive depth-first search.
- This function's added as a method to the `CSP` class.

```
def backtracking_search(self, assignment: Dict[V, D] = {}) ->
Optional[Dict[V, D]]:
 # assignment is complete if every variable is assigned (our
base case)
if len(assignment) == len(self.variables):
  return assignment
 #  get  all  variables  in  the  CSP  but  not  in  the
assignment       unassigned: List[V] = [v for v in self.variables
if v not in assignment]
 # get the every possible domain value of the first unassigned
variable
first: V = unassigned[0]
 for value in self.domains[first]:
local_assignment=assignment.copy()
local_assignment[first] = value
# if we're still consistent, we recurse (continue)
   if self.consistent(first, local_assignment):
                          result:    Optional[Dict[V,    D]]    =
self.backtracking_search(local_assignment)                # if we
didn't find the result, we will end up backtracking
            if result is not None:
```

```
return result
return None
```

- Let's check `backtrackingSearch()`, line by line

```
if len(assignment) == len(self.variables):
    return assignment
```

- The base case for the recursive search is finding a valid assignment for every variable. Once we have, we return the first instance of a solution that was valid (we stop searching).

```
unassigned: List[V] = [v for v in self.variables if v not in assignment]
first: V = unassigned[0]
```

- To select a new variable whose domain we can explore, we go through all of the variables and find the first that doesn't have an assignment.
- To do this, we create a `list` of variables in `self.variables` but not in `assignment` through a list comprehension, and call it `unassigned`. Then we pull out the first value in `unassigned`.

```
for value in self.domains[first]:
local_assignment = assignment.copy()
local_assignment[first] = value
```

- We try assigning every possible domain value for that variable, one at a time. The new assignment for each is stored in a local dictionary called `local_assignment`.
- If the new assignment in `local_assignment` is consistent with all of the constraints (which is what `consistent()` checks for), we continue recursively searching with the new assignment in place. If the new assignment turns out to be complete (the base case), we return the new assignment up the recursion chain.

```
return None  # no solution
```

- Finally, if we've gone through every possible domain value for a particular variable, and there's no solution utilizing the existing set of assignments, we return `None`, indicating no solution.
- This leads to backtracking up the recursion chain to the point where a different prior assignment could have been made.