Utkarsh_Gupta_Lab7(1)

September 9, 2020

Lab 7: Constraint Satisfaction Problems

In this lab assignment, we are solving the map coloring problem and crypto-arithmetic problem using constraint satisfaction problem.

[1]:
```
#Import the necessary libraries
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod
```

[2]:
```
#Declares a type variable V as variable type and D as domain type
V = TypeVar('V') # variable type
D = TypeVar('D') # domain type
```

[3]:
```
#This is a 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

    # This is an abstract method which must be overridden by subclasses
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        ...
```

[4]:
```
# 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:
        # variables to be constrained
        self.variables: List[V] = variables
        # domain of each variable
        self.domains: Dict[V, List[D]] = domains
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        for variable in variables:
            self.constraints[variable] = []
            if variable not in self.domains:
```

```
                raise LookupError("Every variable should have a domain assigned␣
↪to it.")
    #This method add constraint to variables as per their domains
    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)

    # 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

    #This method is performing the backtracking search to find the result
    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
```

[5]:
```
#MapColoringConstraint is a subclass of Constraint class
class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
```

```
        self.place2: str = place2
    #Define the abstract method satisfied in subclass
    def satisfied(self, assignment: Dict[str, str]) -> bool:
        # If either place is not in the assignment then it is not
        # yet possible for their colors to be conflicting
        if self.place1 not in assignment or self.place2 not in assignment:
            return True
        # check the color assigned to place1 is not the same as the
        # color assigned to place2
        return assignment[self.place1] != assignment[self.place2]
```

[6]:
```
#Main starts
if __name__ == "__main__":
    #Initializes the variables as per the regions of the graph
    variables: List[str] = ["BOX_1", "BOX_2", "BOX_4",
                            "BOX_3", "BOX_5", "BOX_6", "BOX_7"]
    domains: Dict[str, List[str]] = dict()
    for variable in variables:
        #Initialize the domain of each variable
        domains[variable] = ["red", "green", "blue"]
    #Instantiate the object of CSP
    csp: CSP[str, str] = CSP(variables, domains)
    #Add constraints to the given MAP problem
    csp.add_constraint(MapColoringConstraint("BOX_1", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_1", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_4", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_5"))
    csp.add_constraint(MapColoringConstraint("BOX_5", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_5"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_7"))
    #Finding the solution to the problem by calling the backtracking_search()␣
 ↪method
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

```
{'BOX_1': 'red', 'BOX_2': 'green', 'BOX_4': 'blue', 'BOX_3': 'red', 'BOX_5':
'green', 'BOX_6': 'red', 'BOX_7': 'green'}
```

[7]:
```
#SendMoreMoneyConstraint is a subclass of Constraint class
class SendMoreMoneyConstraint(Constraint[str, int]):
    def __init__(self, letters: List[str]) -> None:
```

```
        super().__init__(letters)
        self.letters: List[str] = letters

    def satisfied(self, assignment: Dict[str, int]) -> bool:
        # if there are duplicate values then it's not a solution
        if len(set(assignment.values())) < len(assignment):
            return False

        # if all variables have been assigned, check if it adds correctly
        if len(assignment) == len(self.letters):
            s: int = assignment["S"]
            e: int = assignment["E"]
            n: int = assignment["N"]
            d: int = assignment["D"]
            m: int = assignment["M"]
            o: int = assignment["O"]
            r: int = assignment["R"]
            y: int = assignment["Y"]
            send: int = s * 1000 + e * 100 + n * 10 + d
            more: int = m * 1000 + o * 100 + r * 10 + e
            money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
            return send + more == money
        return True # no conflict
```

[8]:
```
if __name__ == "__main__":
    letters: List[str] = ["S", "E", "N", "D", "M", "O", "R", "Y"]
    possible_digits: Dict[str, List[int]] = {}
    for letter in letters:
        possible_digits[letter] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    possible_digits["M"] = [1]  # so we don't get answers starting with a 0
    csp: CSP[str, int] = CSP(letters, possible_digits)
    csp.add_constraint(SendMoreMoneyConstraint(letters))
    solution: Optional[Dict[str, int]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

```
{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}
```

BONUS QUESTIONS: 1. Build your own map and set the constraint as shown in above example 2. Solve the following crypto-arithmetic problem: CROSS + ROADS = DANGER using constraint satisfaction

[9]:
```
#Main starts
if __name__ == "__main__":
    #Initializes the variables as per the regions of the graph
    variables: List[str] = ["UP", "UK", "BIHAR",
```

```
                              "NCR", "CHANDIGARH", "KERALA", "ANDRA PRADESH"]
    #TODO: Initialize the domain as empty dictionary
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        #Initialize the domain of each variable
        domains[variable] = ["red", "green", "blue"]
    #Instantiate the object of CSP
    csp: CSP[str, str] = CSP(variables, domains)
    #Add constraints to the given MAP problem
    csp.add_constraint(MapColoringConstraint("UP", "UK"))
    csp.add_constraint(MapColoringConstraint("UP", "BIHAR"))
    csp.add_constraint(MapColoringConstraint("NCR", "UK"))
    csp.add_constraint(MapColoringConstraint("BIHAR", "KERALA"))
    csp.add_constraint(MapColoringConstraint("BIHAR", "NCR"))
    csp.add_constraint(MapColoringConstraint("BIHAR", "CHANDIGARH"))
    csp.add_constraint(MapColoringConstraint("CHANDIGARH", "NCR"))
    csp.add_constraint(MapColoringConstraint("KERALA", "ANDRA PRADESH"))
    csp.add_constraint(MapColoringConstraint("KERALA", "CHANDIGARH"))
    csp.add_constraint(MapColoringConstraint("KERALA", "UK"))
    #Finding the solution to the problem by calling the backtracking_search()␣
↪method
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

```
{'UP': 'red', 'UK': 'green', 'BIHAR': 'green', 'NCR': 'red', 'CHANDIGARH':
'blue', 'KERALA': 'red', 'ANDRA PRADESH': 'green'}
```

[10]:
```
# Question 2
#SendMoreMoneyConstraint is a subclass of Constraint class
class CrossRoadsDanger(Constraint[str, int]):
    def __init__(self, letters: List[str]) -> None:
        super().__init__(letters)
        self.letters: List[str] = letters

    def satisfied(self, assignment: Dict[str, int]) -> bool:
        # if there are duplicate values then it's not a solution
        if len(set(assignment.values())) < len(assignment):
            return False

        # if all variables have been assigned, check if it adds correctly
        if len(assignment) == len(self.letters):
            c: int = assignment["C"]
            r: int = assignment["R"]
            o: int = assignment["O"]
```

```
            s: int = assignment["S"]
            a: int = assignment["A"]
            d: int = assignment["D"]
            n: int = assignment["N"]
            g: int = assignment["G"]
            e: int = assignment["E"]
            cross: int = c * 10000 + r * 1000 + o * 100 + s * 10 + s
            roads: int = r * 10000 + o * 1000 + a * 100 + d * 10 + s
            danger: int = d * 100000 + a * 10000 + n * 1000 + g * 100 + e * 10␣
↪+ r
            return cross + roads == danger
        return True
```

[11]:
```
if __name__ == "__main__":
    possible_letters: List[str] = ["A", "C", "D", "E", "G", "N", "O", "R", "S"]
    digits: Dict[str, List[int]] = {}
    for i in possible_letters:
        digits[i] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    digits["D"] = [1]
    csp: CSP[str, int] = CSP(possible_letters, digits)
    csp.add_constraint(CrossRoadsDanger(possible_letters))
    solution: Optional[Dict[str, int]] = csp.backtracking_search()
    if solution is None:
        print("Sorry, couldn't find anything")
    else:
        print(solution)
```

```
{'A': 5, 'C': 9, 'D': 1, 'E': 4, 'G': 7, 'N': 8, 'O': 2, 'R': 6, 'S': 3}
```