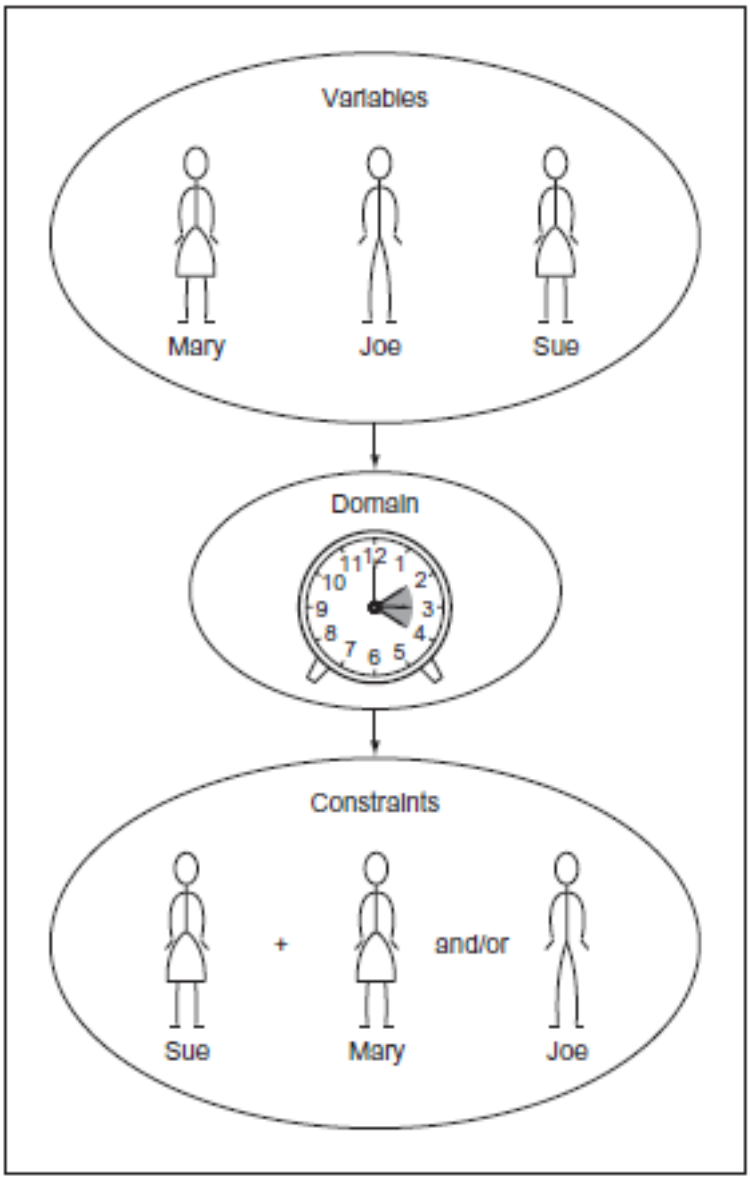
csp

May 31, 2022

1 Constraint Satisfaction Problems

A large number of problems that computational tools are used to solve can be broadly categorized as constraint-satisfaction problems (CSPs). CSPs are composed of variables with possible values that fall into ranges known as domains. Constraints between the variables must be satisfied in order for constraint-satisfaction problems to be solved. Those three core concepts—variables, domains, and constraints—are simple to understand, and their generality underlies the wide applicability of constraint-satisfaction problem solving.

Let's consider an example problem. Suppose you are trying to schedule a Friday meeting for Joe, Mary, and Sue. Sue has to be at the meeting with at least one other person. For this scheduling problem, the three people—Joe, Mary, and Sue—may be the variables. The domain for each variable may be their respective hours of availability. For instance, the variable Mary has the domain 2 p.m., 3 p.m., and 4 p.m. This problem also has two constraints. One is that Sue has to be at the meeting. The other is that at least two people must attend the meeting. A constraint satisfaction problem solver will be provided with the three variables, three domains,and two constraints, and it will then solve the problem without having the user explain exactly how. Following figure illustrates this example.



1.1 Building a constraint-satisfaction problem framework

Constraints will be defined using a Constraint class. Each Constraint consists of the variables it constrains and a method that checks whether it is 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 should be overridden. In fact, it must be, because we are defining our Constraint class as an abstract base class. Abstract base classes are not meant to be instantiated. Instead, only their subclasses that override and implement their @abstractmethods are for actual use.

[1]:
```
from abc import ABC, abstractmethod

# Base class for all constraints
```

```
class Constraint(ABC):
    # The variables that the constraint is between
    def __init__(self, variables):
        self.variables = variables

    # Must be overridden by subclasses
    @abstractmethod
    def satisfied(self, assignment):
        ...
```

The centerpiece of our constraint-satisfaction framework will be 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 variables, domains, and constraints collections are of types that you would 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.

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 will raise an exception when a variable is missing a domain or a constraint is on a nonexistent variable. How do we know if a given configuration of variables and selected domain values satisfies the constraints? We will call such a given configuration an "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.

consistent() goes through every constraint for a given variable (it will always be the variable that was just 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 is not satisfied, False is returned.

This constraint-satisfaction framework will use 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. If you think that sounds like depth-first search from chapter 2, you are perceptive. The backtracking search implemented in the following backtracking_ search() function is a kind of recursive depth-first search

[2]:
```
# 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():
    def __init__(self, variables, domains):
        self.variables = variables  # variables to be constrained
        self.domains = domains  # domain of each variable
```

```
        self.constraints = {}
        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):
        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, assignment):
        for constraint in self.constraints[variable]:
            if not constraint.satisfied(assignment):
                return False
        return True

    def backtracking_search(self, assignment={}):
        # 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 = [v for v in self.variables if v not in assignment]

        # get the every possible domain value of the first unassigned variable
        first = 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 = 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 walk through backtracking_search(), line by line.

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

The base case for the recursive search is having found a valid assignment for every variable. Once

we have, we return the first instance of a solution that was valid. (We do not keep 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 will explore, we simply go through all of the variables and find the first that does not 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 all possible domain values for that variable, one at a time. The new assignment for each is stored in a local dictionary called local_assignment. if self.consistent(first, local_assignment):

```
result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)
if result is not None:
return result
```

If the new assignment in local_assignment is consistent with all of the constraints (that 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.

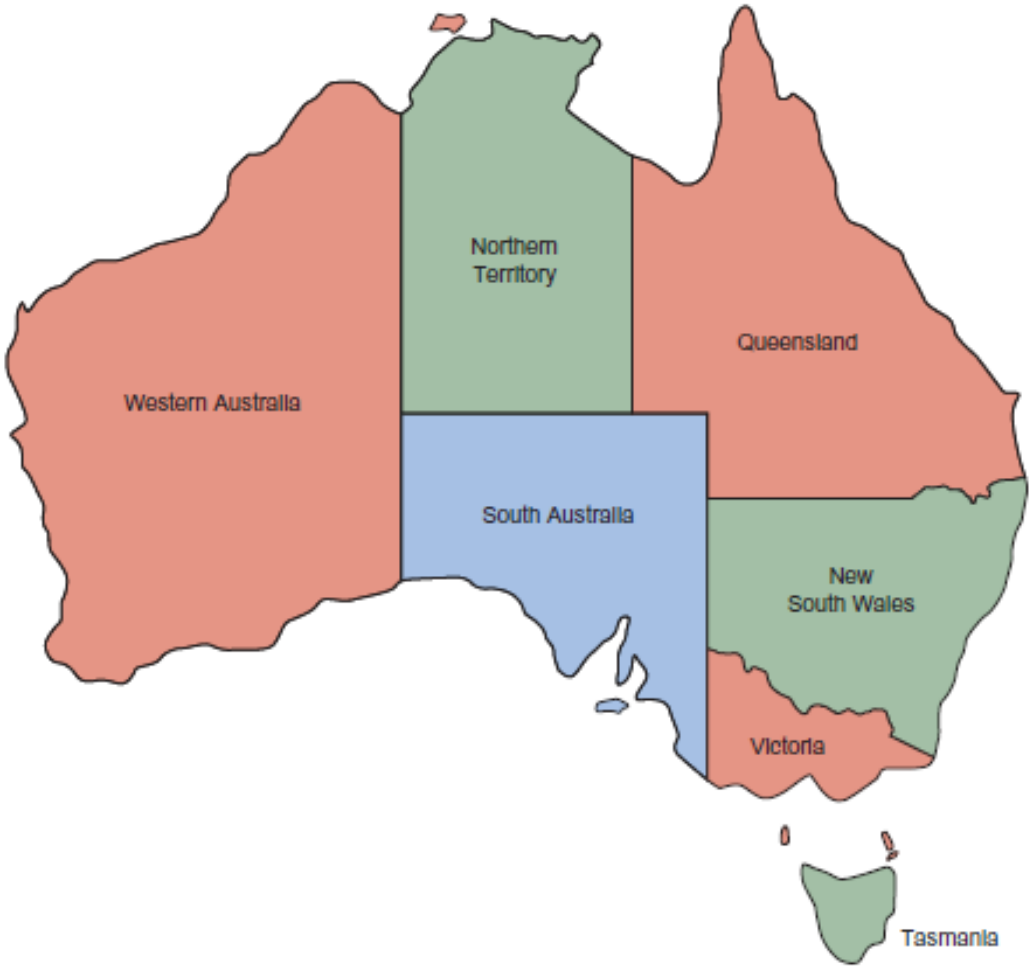
```
return None # no solution
```

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

1.2 The Australian map-coloring problem

Imagine you have a map of Australia that you want to color by state/territory (which we will collectively call "regions"). No two adjacent regions should share a color. Can you color the regions with just three different colors?

The answer is yes. Try it out on your own. (The easiest way is to print out a map of Australia with a white background.) As human beings, we can quickly figure out the solution by inspection and a little trial and error. It is a trivial problem, really, and a great first problem for our backtracking constraint-satisfaction solver. The problem is illustrated in figure:



[3]:
```
class MapColoringConstraint(Constraint):
    def __init__(self, place1, place2):
        super().__init__([place1, place2])
        self.place1 = place1
        self.place2 = place2

    def satisfied(self, assignment):
        # 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]
```

Lets run the code:

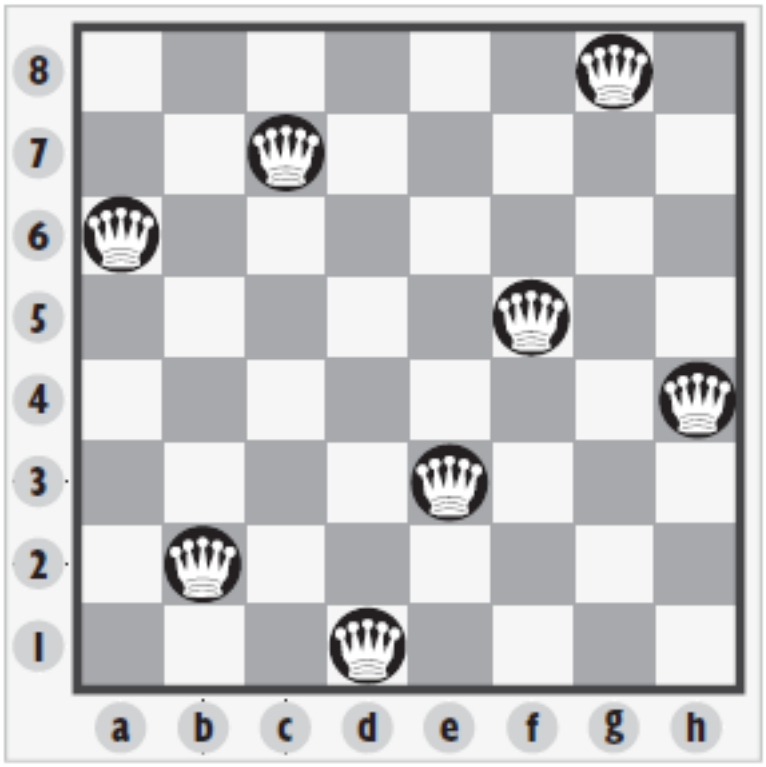
[4]:
```
variables = ["Western Australia", "Northern Territory", "South Australia",
             "Queensland", "New South Wales", "Victoria", "Tasmania"]
domains = {}
for variable in variables:
    domains[variable] = ["red", "green", "blue"]
csp = CSP(variables, domains)
csp.add_constraint(MapColoringConstraint(
    "Western Australia", "Northern Territory"))
csp.add_constraint(MapColoringConstraint(
    "Western Australia", "South Australia"))
csp.add_constraint(MapColoringConstraint(
    "South Australia", "Northern Territory"))
csp.add_constraint(MapColoringConstraint("Queensland", "Northern Territory"))
csp.add_constraint(MapColoringConstraint("Queensland", "South Australia"))
csp.add_constraint(MapColoringConstraint("Queensland", "New South Wales"))
csp.add_constraint(MapColoringConstraint("New South Wales", "South Australia"))
csp.add_constraint(MapColoringConstraint("Victoria", "South Australia"))
csp.add_constraint(MapColoringConstraint("Victoria", "New South Wales"))
csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))

solution = csp.backtracking_search()
if solution is None:
    print("No solution found!")
else:
    print(solution)
```

```
{'Western Australia': 'red', 'Northern Territory': 'green', 'South Australia':
'blue', 'Queensland': 'red', 'New South Wales': 'green', 'Victoria': 'red',
'Tasmania': 'green'}
```

1.3 The eight queens problem

A chessboard is an eight-by-eight grid of squares. A queen is a chess piece that can move on the chessboard any number of squares along any row, column, or diagonal. A queen is attacking another piece if in a single move, it can move to the square the piece is on without jumping over any other piece. (In other words, if the other piece is in the line of sight of the queen, then it is attacked by it.) The eight queens problem poses the question of how eight queens can be placed on a chessboard without any queen attacking another queen. The problem is illustrated in the figure:



To represent squares on the chessboard, we will assign each an integer row and an integer column. We can ensure each of the eight queens is not on the same column by simply assigning them sequentially the columns 1 through 8. The variables in our constraint-satisfaction problem can just be the column of the queen in question. The domains can be the possible rows (again, 1 through 8).

To solve the problem, we will need a constraint that checks whether any two queens are on the same row or diagonal. (They were all assigned different sequential columns to begin with.) Checking for the same row is trivial, but checking for the same diagonal requires a little bit of math. If any two queens are on the same diagonal, the difference between their rows will be the same as the difference between their columns. Can you see where these checks take place in QueensConstraint?

[5]:
```
class QueensConstraint(Constraint):
    def __init__(self, columns):
        super().__init__(columns)
        self.columns = columns

    def satisfied(self, assignment):
        # q1c = queen 1 column, q1r = queen 1 row
        for q1c, q1r in assignment.items():
            # q2c = queen 2 column
            for q2c in range(q1c + 1, len(self.columns) + 1):
                if q2c in assignment:
                    q2r = assignment[q2c]  # q2r = queen 2 row
```

```
                    if q1r == q2r:  # same row?
                        return False
                    if abs(q1r - q2r) == abs(q1c - q2c):  # same diagonal?
                        return False
        return True  # no conflict
```

Notice that we were able to reuse the constraint-satisfaction problem-solving framework that we built for map coloring fairly easily for a completely different type of problem. This is the power of writing code generically! Algorithms should be implemented in as broadly applicable a manner as possible unless a performance optimization for a particular application requires specialization.

Lets run the code

[8]:
```
columns = [1, 2, 3, 4, 5, 6, 7, 8]
rows = {}

for column in columns:
    rows[column] = [1, 2, 3, 4, 5, 6, 7, 8]
csp = CSP(columns, rows)
csp.add_constraint(QueensConstraint(columns))

solution = csp.backtracking_search()
if solution is None:
    print("No solution found!")
else:
    print(solution)
```

```
{1: 1, 2: 5, 3: 8, 4: 6, 5: 3, 6: 7, 7: 2, 8: 4}
```

A correct solution will assign a column and row to every queen.