The Dogs

BERECZKI NORBET CRISTIAN
GR. 256

Statement:

Mr. Adams, Mrs. Barber, Mr. Cox, Miss Duke, and Miss Evans live in a row of houses in the same block. Each owns a dog. The dogs they own are a beagle, a collie, a dachshund, a poodle, and a retriever.

- Mr. Cox and Miss Duke live at the ends of the row of houses.
- A woman owns the retriever.
- The collie lives in the middle house.
- Miss Duke does not own the poodle.
- The dachshund was a gift from the owner's husband.
- The retriever lives between the collie and the beagle.

Which breed of dog does each person own?

Search Space:

Let P = {"Mr. Adams", "Mrs. Barber", "Mr. Cox", "Miss Duke", "Miss Evans"}
Let D = {"beagle", "collie", "dachshund", "poodle", "retriever"}
Let V = {(p,d) for each p in P and each d in D}

Seach Space would be D: V x V x V x V x V

The depth of the search tree should be 5.

Solution:

Code in Python 3.7.4.

Naive Solution

We consider a list of people and a list of dogs. We make a list of permutation of the people and one for the dogs. We try to combine any permutation of the people's list with any permutation of the dogs. Once we have this pair(of a permutation of people and a permutation of the dogs) we check if this solution is satisfied.

Improved backtracking

1. We will consider as a variable each pair of (person & dog) and for each such variable we will consider the domain as being the number of possible house numbers.
2. We will reduce the restrictions to unary constraints.
3. After reducing the constraints we are able to enforce node consistency and reduce the search space.
4. After doing this we apply the backtracking algorithm to find the final solution.

A comparison follows:

Naive solution

(we note here that an iterative approach was used and not a recursive one):

```
python dogs.py  0.03s user 0.01s system 88% cpu 0.045 total
```

Improved backtracking:

```
python lab2_dogs.py  0.66s user 0.01s system 98% cpu 0.685 total
```

Sample solution:

---------- SOLUTION ----------

Person: Mr. Adams
Dog: collie

House number: 3

Person: Mrs. Barber
Dog: dachshund

House number: 2

Person: Mr. Cox
Dog: poodle

House number: 1

Person: Miss Evans
Dog: retriever

House number: 4

Person: Miss Duke
Dog: beagle

House number: 5
---------- END SOLUTION ----------

Code:

```python
import sys
from copy import deepcopy


# ------------------------------- CLASSES -------------------------------


class Variable:

    def __init__(self, person, dog):
        self._person = person
        self._dog = dog

    def get_person(self):
        return self._person

    def get_dog(self):
        return self._dog

    def __str__(self):
        return f"""
        -----
        Person: {self._person}
        Dog: {self._dog}
        -----"""


class BaseRestriction:
    # each function
    # returns True if restriction is violated
    # returns False otherwise
    def restrict(self, variable, value):
        raise Exception("Method not implemented")


class Restriction1(BaseRestriction):

    def restrict(self, variable, value):
        p = variable.get_person()
        v = value

        if p == "Mr. Cox" or p == "Miss Duke":
            return v not in [1, 5]
        else:
            return v not in [2, 3, 4]


class Restriction2(BaseRestriction):

    def restrict(self, variable, value):
        p = variable.get_person()
        d = variable.get_dog()

        if d != "retriever":
            return False
        if not ("Miss" in p or "Mrs." in p):
            return True
        return False
```

```python


class Restriction3(BaseRestriction):

    def restrict(self, variable, value):
        d = variable.get_dog()
        v = value

        if d != "collie":
            return v == 3
        else:
            return v != 3


class Restriction4(BaseRestriction):

    def restrict(self, variable, value):
        d = variable.get_dog()
        p = variable.get_person()

        if d == "poodle" and p == "Miss Duke":
            return True
        return False


class Restriction5(BaseRestriction):

    def restrict(self, variable, value):
        d = variable.get_dog()
        p = variable.get_person()

        if d != "dachshund":
            return False
        if not ("Miss" in p or "Mrs." in p):
            return True
        return False


class Restriction6(BaseRestriction):

    def restrict(self, variable, value):
        d = variable.get_dog()
        v = value

        if d != "retriever":
            return False
        # should be retriever
        return v not in [2, 4]


class Restriction7(BaseRestriction):

    def restrict(self, variable, value):
        d = variable.get_dog()
        v = value

        if d != "beagle":
            return False
```

```python
        # should be beagle
        return v not in [1, 5]

# ____________________ DEFINE NODE CONSISTENCY ENFORCER ________________________

class NodeConsistencyEnforcer:

    def __init__(self, variables, restrictions, domain):
        self.variables = variables
        self.restrictions = restrictions
        self.domain = domain

    def enforce(self):
        nodes = []
        for variable in self.variables:
            new_vals = []
            for val in self.domain:
                violates = False
                for restriction in self.restrictions:
                    violates = violates or restriction.restrict(variable, val)

                if not violates:
                    new_vals.append(val)
            if len(new_vals) > 0:
                nodes.append([variable, new_vals])
        return nodes

# -------------------------------- MAIN --------------------------------------

people = ["Mr. Adams", "Mrs. Barber", "Mr. Cox", "Miss Duke", "Miss Evans"]
dogs = ["beagle", "collie", "dachshund", "poodle", "retriever"]

variables = []
for p in people:
    for d in dogs:
        variables.append(Variable(person=p,dog=d))

unary_restrictions = [
    Restriction1(),
    Restriction2(),
    Restriction3(),
    Restriction4(),
    Restriction5(),
    Restriction6(),
    Restriction7(),
]

enforcer = NodeConsistencyEnforcer(
    variables=variables,
    restrictions=unary_restrictions,
    domain=list(range(1,6))
)

nodes = enforcer.enforce()

for variable, domain in nodes:
    print(variable)
    print(domain)
```

```python
# _____________________ BACKTRACKING ____________________


class Linearizer:
    @staticmethod
    def linearize(nodes):
        linearized = []
        for variable, vals in nodes:
            for v in vals:
                linearized.append([variable, v])
        return linearized


class BacktrackingSolver:

    def __init__(self, candidates):
        self.candidates = candidates

    def solve(self):
        self.sols = []
        self._back([])
        return self.sols


    def _check(self, partial_solution):
        ps = [variable.get_person() for variable, value in partial_solution]
        ds = [variable.get_dog() for variable, value in partial_solution]
        vs = [value for variable, value in partial_solution]

        return (
            len(ps) == len(set(ps)) and
            len(ds) == len(set(ds)) and
            len(vs) == len(set(vs))
        )

    def _back(self, partial_solution):

        if len(partial_solution) == 5:
            self.sols.append(deepcopy(partial_solution))
            return

        for candidate in self.candidates:
            partial_solution.append(candidate)
            if self._check(partial_solution):
                self._back(partial_solution)
            partial_solution.pop()


candidates = Linearizer().linearize(nodes)


solver = BacktrackingSolver(candidates)
sols = solver.solve()
```