Implement Crossword Functions

crossword.py
Python

generate.py
Python

There are two Python files in this project: crossword.py and generate.py. The first has been entirely written for you, the second has some functions that are left for you to implement.

First, let's take a look at crossword.py. This file defines two classes, Variable (to represent a variable in a crossword puzzle) and Crossword (to represent the puzzle itself).

Notice that to create a Variable, we must specify four values: its row i, its column j, its direction (either the constant Variable.ACROSS or the constant Variable.DOWN), and its length.

The Crossword class requires two values to create a new crossword puzzle: a structure_file that defines the structure of the puzzle (the _ is used to represent blank cells, any other character represents cells that won't be filled in) and a words_file that defines a list of words (one on each line) to use for the vocabulary of the puzzle. Three examples of each of these files can be found in the data directory of the project, and you're welcome to create your own as well.

Note in particular, that for any crossword object crossword, we store the following values:

crossword.height is an integer representing the height of the crossword puzzle.
crossword.width is an integer representing the width of the crossword puzzle.
crossword.structure is a 2D list representing the structure of the puzzle. For any valid row i and column j, crossword.structure[i][j] will be True if the cell is blank (a character must be filled there) and will be False otherwise (no character is to be filled in that cell).
crossword.words is a set of all of the words to draw from when constructing the crossword puzzle.
crossword.variables is a set of all of the variables in the puzzle (each is a Variable object).
crossword.overlaps is a dictionary mapping a pair of variables to their overlap. For any two distinct variables v1 and v2, crossword.overlaps[v1, v2] will be None if the two variables have no overlap, and will be a pair of integers (i, j) if the variables do overlap. The pair (i, j) should be interpreted to mean that the ith character of v1's value must be the same as the jth character of v2's value.
Crossword objects also support a method neighbors that returns all of the variables that overlap with a given variable. That is to say, crossword.neighbors(v1) will return a set of all of the variables that are neighbors to the variable v1.

Next, take a look at generate.py. Here, we define a class CrosswordCreator that we'll use to solve the crossword puzzle. When a CrosswordCreator object is created, it gets a crossword property that should be a Crossword object (and therefore has all of the properties described above). Each CrosswordCreator object also gets a domains property: a dictionary that maps variables to a set of possible words the variable might take on as a value. Initially, this set of words is all of the words in our vocabulary, but we'll soon write functions to restrict these domains.

We've also defined some functions for you to help with testing your code: print will print to the terminal a representation of your crossword puzzle for a given assignment (every assignment, in this function and elsewhere, is a dictionary mapping variables to their corresponding words). save, meanwhile, will generate an image file corresponding to a given assignment (you'll need to pip3 install Pillow if you haven't already to use this function). letter_grid is a helper function used by both print and save that generates a 2D list of all characters in their appropriate positions for a given assignment: you likely won't need to call this function yourself, but you're welcome to if you'd like to.

Finally, notice the solve function. This function does three things: first, it calls enforce_node_consistency to enforce node consistency on the crossword puzzle, ensuring that every value in a variable's domain satisfy the unary constraints. Next, the function calls ac3 to enforce arc consistency, ensuring that binary constraints are satisfied. Finally, the function calls backtrack on an initially empty assignment (the empty dictionary dict()) to try to calculate a solution to the problem.

The functions enforce_node_consistency, ac3, and backtrack, though, are not yet implemented (among other functions). That's where you come in!

Specification
Complete the implementation of enforce_node_consistency, revise, ac3, assignment_complete, consistent, order_domain_values, selected_unassigned_variable, and backtrack in generate.py so that your AI generates complete crossword puzzles if it is possible to do so.

The enforce_node_consistency function should update self.domains such that each variable is node consistent.

Recall that node consistency is achieved when, for every variable, each value in its domain is consistent with the variable's unary constraints. In the case of a crossword puzzle, this means making sure that every value in a variable's domain has the same number of letters as the variable's length.
To remove a value x from the domain of a variable v, since self.domains is a dictionary mapping variables to sets of values, you can call self.domains[v].remove(x).
No return value is necessary for this function.
The revise function should make the variable x arc consistent with the variable y.

x and y will both be Variable objects representing variables in the puzzle.
Recall that x is arc consistent with y when every value in the domain of x has a possible value in the domain of y that does not cause a conflict. (A conflict in the context of the crossword puzzle is a square for which two variables disagree on what character value it should take on.)
To make x arc consistent with y, you'll want to remove any value from the domain of x that does not have a corresponding possible value in the domain of y.
Recall that you can access self.crossword.overlaps to get the overlap, if any, between two variables.
The domain of y should be left unmodified.
The function should return True if a revision was made to the domain of x; it should return False if no revision was made.
The ac3 function should, using the AC3 algorithm, enforce arc consistency on the problem. Recall that arc consistency is achieved when all the values in each variable's domain satisfy that variable's binary constraints.

Recall that the AC3 algorithm maintains a queue of arcs to process. This function takes an optional argument called arcs, representing an initial list of arcs to process. If arcs is None, your function should start with an initial queue of all of the arcs in the problem. Otherwise, your algorithm should begin with an initial queue of only the arcs that are in the list arcs (where each arc is a tuple (x, y) of a variable x and a different variable y).
Recall that to implement AC3, you'll revise each arc in the queue one at a time. Any time you make a change to a domain, though, you may need to add additional arcs to your queue to ensure that other arcs stay consistent.
You may find it helpful to call on the revise function in your implementation of ac3.
If, in the process of enforcing arc consistency, you remove all of the remaining values from a domain, return False (this means it's impossible to solve the problem, since there are no more possible values for the variable). Otherwise, return True.
You do not need to worry about enforcing word uniqueness in this function (you'll implement that check in the consistent function.)
The assignment_complete function should (as the name suggests) check to see if a given assignment is complete.

An assignment is a dictionary where the keys are Variable objects and the values are strings representing the words those variables will take on.
An assignment is complete if every crossword variable is assigned to a value (regardless of what that value is).
The function should return True if the assignment is complete and return False otherwise.
The consistent function should check to see if a given assignment is consistent.

An assignment is a dictionary where the keys are Variable objects and the values are strings representing the words those variables will take on. Note that the assignment may not be complete: not all variables will necessarily be present in the assignment.
An assignment is consistent if it satisfies all of the constraints of the problem: that is to say, all values are distinct, every value is the correct length, and there are no conflicts between neighboring variables.
The function should return True if the assignment is consistent and return False otherwise.
The order_domain_values function should return a list of all of the values in the domain of var, ordered according to the least-constraining values heuristic.

var will be a Variable object, representing a variable in the puzzle.
Recall that the least-constraining values heuristic is computed as the number of values ruled out for neighboring unassigned variables. That is to say, if assigning var to a particular value results in eliminating n possible choices for neighboring variables, you should order your results in ascending order of n.
Note that any variable present in assignment already has a value, and therefore shouldn't be counted when computing the number of values ruled out for neighboring unassigned variables.
For domain values that eliminate the same number of possible choices for neighboring variables, any ordering is acceptable.
Recall that you can access self.crossword.overlaps to get the overlap, if any, between two variables.
It may be helpful to first implement this function by returning a list of values in any arbitrary order (which should still generate correct crossword puzzles). Once your algorithm is working, you can then go back and ensure that the values are returned in the correct order.
You may find it helpful to sort a list according to a particular key: Python contains some helpful functions for achieving this.
The select_unassigned_variable function should return a single variable in the crossword puzzle

that is not yet assigned by assignment, according to the minimum remaining value heuristic and then the degree heuristic.

An assignment is a dictionary where the keys are Variable objects and the values are strings representing the words those variables will take on. You may assume that the assignment will not be complete: not all variables will be present in the assignment.
Your function should return a Variable object. You should return the variable with the fewest number of remaining values in its domain. If there is a tie between variables, you should choose among whichever among those variables has the largest degree (has the most neighbors). If there is a tie in both cases, you may choose arbitrarily among tied variables.
It may be helpful to first implement this function by returning any arbitrary unassigned variable (which should still generate correct crossword puzzles). Once your algorithm is working, you can then go back and ensure that you are returning a variable according to the heuristics.
You may find it helpful to sort a list according to a particular key: Python contains some helpful functions for achieving this.
The backtrack function should accept a partial assignment assignment as input and, using backtracking search, return a complete satisfactory assignment of variables to values if it is possible to do so.

An assignment is a dictionary where the keys are Variable objects and the values are strings representing the words those variables will take on. The input assignment may not be complete (not all variables will necessarily have values).
If it is possible to generate a satisfactory crossword puzzle, your function should return the complete assignment: a dictionary where each variable is a key and the value is the word that the variable should take on. If no satisfying assignment is possible, the function should return None.
If you would like, you may find that your algorithm is more efficient if you interleave search with inference (as by maintaining arc consistency every time you make a new assignment). You are not required to do this, but you are permitted to, so long as your function still produces correct results. (It is for this reason that the ac3 function allows an arcs argument, in case you'd like to start with a different queue of arcs.)
You should not modify anything else in generate.py other than the functions the specification calls for you to implement, though you may write additional functions and/or import other Python standard library modules. You may also import numpy or pandas, if familiar with them, but you should not use any other third-party Python modules. You should not modify anything in crossword.py.

Hints
For order_domain_values and select_unassigned_variable, it may be helpful to implement them first without worrying about the heuristics, and then add heuristics later. Your algorithm will still work: it just may end up exploring more assignments than it needs to before finding a solution.
To run your program, you can run a command like python generate.py data/structure1.txt data/words1.txt, specifying a structure file and a words file. If an assignment is possible, you should see the resulting assignment printed. You may also add an additional command-line argument for an image file, as by running python generate.py data/structure1.txt data/words1.txt output.png, to generate an image representation of the resulting crossword puzzle as well.
The Crossword class has a neighbors function you can use to access all of the neighbors (i.e., overlapping variables) of a particular variable. Feel free to use that any time you need to determine the neighbors of a particular variable!

Let's begin by implementing the functions in `generate.py` according to the specifications. We'll start with the `enforce_node_consistency` function.

`enforce_node_consistency`

This function ensures that each value in a variable's domain satisfies the unary constraints (i.e., the length of the word matches the length of the variable).
Here is the implementation for `enforce_node_consistency`:

```python
def enforce_node_consistency(self):
    for var in self.crossword.variables:
        for word in set(self.domains[var]):
            if len(word) != var.length:
                self.domains[var].remove(word)
```

`revise`

This function makes a variable arc consistent with another variable.
Here is the implementation for `revise`:

```python
def revise(self, x, y):
    revised = False
    overlap = self.crossword.overlaps[x, y]
    if overlap:
        i, j = overlap
        for word_x in set(self.domains[x]):
            satisfies_constraint = False
            for word_y in self.domains[y]:
                if word_x[i] == word_y[j]:
                    satisfies_constraint = True
                    break
            if not satisfies_constraint:
                self.domains[x].remove(word_x)
                revised = True
    return revised
```

`ac3`

This function enforces arc consistency using the AC3 algorithm.
Here is the implementation for `ac3`:

```python
def ac3(self, arcs=None):
    if arcs is None:
        arcs = [(x, y) for x in self.crossword.variables for y in
self.crossword.neighbors(x)]

    queue = collections.deque(arcs)

    while queue:
        (x, y) = queue.popleft()
```

```
            if self.revise(x, y):
                if not self.domains[x]:
                    return False
                for z in self.crossword.neighbors(x) - {y}:
                    queue.append((z, x))
        return True
```

`assignment_complete`

This function checks if the assignment is complete.
Here is the implementation for **`assignment_complete`**:

python

```
    def assignment_complete(self, assignment):
        return set(assignment.keys()) == self.crossword.variables
```

`consistent`

This function checks if the assignment is consistent.
Here is the implementation for **`consistent`**:

python

```
    def consistent(self, assignment):
        for var, word in assignment.items():
            if len(word) != var.length:
                return False
            for neighbor in self.crossword.neighbors(var):
                if neighbor in assignment:
                    i, j = self.crossword.overlaps[var, neighbor]
                    if word[i] != assignment[neighbor][j]:
                        return False
        if len(set(assignment.values())) != len(assignment.values()):
            return False
        return True
```

`order_domain_values`

This function orders the domain values according to the least-constraining values heuristic.
Here is the implementation for **`order_domain_values`**:

python

```
    def order_domain_values(self, var, assignment):
        def count_conflicts(value):
            conflicts = 0
            for neighbor in self.crossword.neighbors(var):
                if neighbor not in assignment:
                    i, j = self.crossword.overlaps[var, neighbor]
                    for neighbor_val in self.domains[neighbor]:
                        if value[i] != neighbor_val[j]:
                            conflicts += 1
            return conflicts
```

```
        return sorted(self.domains[var], key=count_conflicts)
```

`select_unassigned_variable`

This function selects an unassigned variable using MRV and degree heuristics.
Here is the implementation for `select_unassigned_variable`:

```python
    def select_unassigned_variable(self, assignment):
        unassigned = [v for v in self.crossword.variables if v not in assignment]

        def mrv_heuristic(var):
            return (len(self.domains[var]), -len(self.crossword.neighbors(var)))

        return min(unassigned, key=mrv_heuristic)
```

`backtrack`

This function uses backtracking search to find a complete assignment.
Here is the implementation for `backtrack`:

```python
    def backtrack(self, assignment):
        if self.assignment_complete(assignment):
            return assignment

        var = self.select_unassigned_variable(assignment)
        for value in self.order_domain_values(var, assignment):
            new_assignment = assignment.copy()
            new_assignment[var] = value
            if self.consistent(new_assignment):
                result = self.backtrack(new_assignment)
                if result is not None:
                    return result
        return None
```

Updated `generate.py`

Here is the complete `generate.py` with the implementations:

```python
    import collections
    from crossword import *


    class CrosswordCreator:

        def __init__(self, crossword):
            """
            Create new CSP crossword generator.
            """
            self.crossword = crossword
            self.domains = {
```

```
            var: self.crossword.words.copy()
            for var in self.crossword.variables
        }

    def print(self, assignment):
        """
        Print crossword assignment to the terminal.
        """
        letters = self.letter_grid(assignment)
        for i in range(self.crossword.height):
            for j in range(self.crossword.width):
                if self.crossword.structure[i][j]:
                    print(letters[i][j] or " ", end="")
                else:
                    print("", end="")
            print()

    def save(self, assignment, filename):
        """
        Save crossword assignment to an image file.
        """
        from PIL import Image, ImageDraw, ImageFont

        # Create a blank canvas
        cell_size = 100
        cell_border = 2

        img = Image.new(
            "RGBA",
            (self.crossword.width * cell_size,
             self.crossword.height * cell_size),
            "white"
        )
        draw = ImageDraw.Draw(img)

        # Load a font
        try:
            font = ImageFont.truetype("arial.ttf", 80)
        except IOError:
            font = ImageFont.load_default()

        letters = self.letter_grid(assignment)
        for i in range(self.crossword.height):
            for j in range(self.crossword.width):
                rect = [
                    (j * cell_size + cell_border, i * cell_size + cell_border),
                    ((j + 1) * cell_size - cell_border, (i + 1) * cell_size - cell_border)
                ]
                if self.crossword.structure[i][j]:
                    draw.rectangle(rect, fill="white")
                    if letters[i][j]:
                        w, h = draw.textsize(letters[i][j], font=font)
                        draw.text(
                            (rect[0][0] + (cell_size - w) / 2, rect[0][1] + (cell_size -
h) / 2 - 10),
                            letters[i][j], fill="black", font=font
                        )
                else:
                    draw.rectangle(rect, fill="black")

        img.save(filename)

    def letter_grid(self, assignment):
        """
        Return 2D array representing a given assignment.
        """
        letters = [
            [None for _ in range(self.crossword.width)]
            for _ in range(self.crossword.height)
        ]
        for variable, word in assignment.items():
            direction = variable.direction
            for k in range(len(word)):
                i = variable.i + (k if direction == Variable.DOWN else 0)
                j = variable.j + (k if direction == Variable.ACROSS else 0)
```

```
                letters[i][j] = word[k]
        return letters

    def solve(self):
        """
        Enforce node and arc consistency, and then solve the CSP.
        """
        self.enforce_node_consistency()
        self.ac3()
        return self.backtrack(dict())

    def enforce_node_consistency(self):
        """
        Update `self.domains` such that each variable is node-consistent.
        (Remove any values that are inconsistent with a variable's unary
        constraints; in this case, the length of the word.)
        """
        for var in self.crossword.variables:
            for word in set(self.domains[var]):
                if len(word) != var.length:
                    self.domains[var].remove(word)

    def revise(self, x, y):
        """
        Make variable `x` arc consistent with variable `y`.
        To do so, remove values from `self.domains[x]` for which there is no
        possible corresponding value in `self.domains[y]` that does not
        conflict. Return True if a revision was made to the domain of
        `x`; return False if no revision was made.
        """
        revised = False
        overlap = self.crossword.overlaps[x, y]
        if overlap:
            i, j = overlap
            for word_x in set(self.domains[x]):
                satisfies_constraint = False
                for word_y in self.domains[y]:
                    if word_x[i] == word_y[j]:
                        satisfies_constraint = True
                        break
                if not satisfies_constraint:
                    self.domains[x].remove(word_x)
                    revised = True
        return revised

    def ac3(self, arcs=None):
        """
        Update `self.domains` such that each variable is arc consistent.
        If `arcs` is None, begin with initial list of all arcs in the problem.
        Otherwise, use `arcs` as the initial list of arcs to make consistent.
        Return True if arc consistency is enforced and no domains are empty;
        return False if one or more domains end up empty.
        """
        if arcs is None:
            arcs = [(x, y) for x in self.crossword.variables for y in
self.crossword.neighbors(x)]

        queue = collections.deque(arcs)

        while queue:
            (x, y) = queue.popleft()
            if self.revise(x, y):
                if not self.domains[x]:
                    return False
                for z in self.crossword.neighbors(x) - {y}:
                    queue.append((z, x))
        return True

    def assignment_complete(self, assignment):
        """
        Return True if `assignment` is complete (i.e., assigns a value to each
        crossword variable); return False otherwise.
        """
        return set(assignment.keys()) == self.crossword.variables
```

```
    def consistent(self, assignment):
        """
        Return True if `assignment` is consistent (i.e., words fit in crossword
        puzzle without conflicting characters); return False otherwise.
        """
        for var, word in assignment.items():
            if len(word) != var.length:
                return False
            for neighbor in self.crossword.neighbors(var):
                if neighbor in assignment:
                    i, j = self.crossword.overlaps[var, neighbor]
                    if word[i] != assignment[neighbor][j]:
                        return False
        if len(set(assignment.values())) != len(assignment.values()):
            return False
        return True

    def order_domain_values(self, var, assignment):
        """
        Return a list of values in the domain of `var`, in order by the number
        of values they rule out for neighboring variables.
        The first value in the list, for example, should be the one that rules
        out the fewest values among the neighbors of `var`.
        """
        def count_conflicts(value):
            conflicts = 0
            for neighbor in self.crossword.neighbors(var):
                if neighbor not in assignment:
                    i, j = self.crossword.overlaps[var, neighbor]
                    for neighbor_val in self.domains[neighbor]:
                        if value[i] != neighbor_val[j]:
                            conflicts += 1
            return conflicts

        return sorted(self.domains[var], key=count_conflicts)

    def select_unassigned_variable(self, assignment):
        """
        Return an unassigned variable not already part of `assignment`.
        Choose the variable with the minimum number of remaining values in its
        domain. If there is a tie, choose the variable with the highest degree.
        If there is a tie in both cases, any of the tied variables are acceptable
        return values.
        """
        unassigned = [v for v in self.crossword.variables if v not in assignment]

        def mrv_heuristic(var):
            return (len(self.domains[var]), -len(self.crossword.neighbors(var)))

        return min(unassigned, key=mrv_heuristic)

    def backtrack(self, assignment):
        """
        Using Backtracking Search, take as input a partial assignment for the
        crossword and return a complete assignment if possible to do so.
        `assignment` is a mapping from variables (keys) to words (values).
        If no assignment is possible, return None.
        """
        if self.assignment_complete(assignment):
            return assignment

        var = self.select_unassigned_variable(assignment)
        for value in self.order_domain_values(var, assignment):
            new_assignment = assignment.copy()
            new_assignment[var] = value
            if self.consistent(new_assignment):
                result = self.backtrack(new_assignment)
                if result is not None:
                    return result
        return None
```

Now, you should be able to test the program by running the command `python generate.py data/structure1.txt data/words1.txt` with appropriate structure and words files. If an assignment is possible, the resulting assignment should be printed. You may also save an image of the puzzle by running `python generate.py data/structure1.txt data/words1.txt output.png`.
Feel free to modify or add any helper functions or further refine the existing implementations as needed.