Lab 3: Constraint Satisfaction Problems

519021910702 游克垚

1 Exercise 1: Rearrange the Seats

1.1 __classroom__conflict()

This function is to check if $var1 \leftarrow val1$ and $var2 \leftarrow val2$ will cause conflicts. There are two conditions to cause conflicts. One is that $val1 == val2$ and another is that $var1$ and $var2$ is adjacent to each other and they are friends. So my implementation is as follows:

```
def _classroom_conflict(self, var1, val1, var2, val2):
        return val1 != val2 and (not self._is_adjacent(var1, var2) or not self.
            _is_friend(val1, val2))
```

1.2 backtracking()

I use nconflicts() to check consistency, assign() to assign a value to a variable and unassign() to unassign a value from a variable. My implementation is as follows:

```
def backtracking(
    csp,
    select_unassigned_variable=mrv,
    order_domain_values=lcv
):
    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if csp.nconflicts(var, value, assignment) == 0:
                csp.assign(var, value, assignment)
                result = backtrack(assignment)
                if result != None:
                    return result
                csp.unassign(var, assignment)
        return None
    result = backtrack({})
    assert result is None or csp.goal_test(result)
    return result
```

1.3 result

The result of easy_classroom is as follows:

```
Time consumption: 0.0056s
Result: ⍰ Success
Solution:
[[4 2]
 [3 1]
 [5 6]
 [7 8]]
```

The result of fail_classroom is as follows:

```
Time consumption: 0.0001s
Result: ⍰ Fail (no solution found).
```

2 Exercise 2: Sudoku (Filtering)

2.1 forward_checking()

The forward_checking() is to prune neighbor values inconsistent with var=value. My implementation is as follows:

```python
def forward_checking(csp, var, value, assignment, removals):
    csp.support_pruning()  # It is necessary for using csp.prune()
    for v in csp.neighbors[var]:
        for val2 in csp.curr_domains[v]:
            if not csp.constraints(var, value, v, val2):
                csp.prune(v, val2, removals)
    return True
```

2.2 AC3()

The AC3() is to make sure all arcs are consistent after assigning a variable. My implementation is as follows:

```python
def AC3(csp, removals=None):
    def revise(Xi, Xj):
        removed = False
        for i in csp.curr_domains[Xi]:
            consistent = False
            for j in csp.curr_domains[Xj]:
                if csp.constraints(Xi, i, Xj, j):
                    consistent = True
            if not consistent:
                csp.prune(Xi, i, removals)
                removed = True
        return removed

    queue = {(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]}
    csp.support_pruning()  # It is necessary for using csp.prune()
    while queue:
        Xi, Xj = queue.pop()
        if revise(Xi, Xj):
            for Xk in csp.neighbors[Xi]:
                queue.add((Xk, Xi))

    return True  # CSP is satisfiable
```

2.3 backtracking_with_inference()

The backtracking_with_inference() is to use filtering strategies in backtracking algorithm. My implementation is as follows:

```python
def backtracking_with_inference(
    csp,
    inference,
    select_unassigned_variable=mrv,
    order_domain_values=lcv
):
    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if csp.nconflicts(var, value, assignment) == 0:
                removals = csp.suppose(var, value)
                csp.assign(var, value, assignment)
                inference(csp, var, value, assignment, removals)
                result = backtrack(assignment)
                if result != None:
                    return result
                csp.unassign(var, assignment)
                csp.restore(removals)
        return None

    result = backtrack({})
    assert result is None or csp.goal_test(result)
    return result
```

2.4 result

The result of forward_checking() on easy_sudoku is as follows:

```
Time consumption: 0.0053s
Result: ? Success
Solution:
4 8 3 | 9 2 1 | 6 5 7
9 6 7 | 3 4 5 | 8 2 1
2 5 1 | 8 7 6 | 4 9 3
------+-------+------
5 4 8 | 1 3 2 | 9 7 6
7 2 9 | 5 6 4 | 1 3 8
1 3 6 | 7 9 8 | 2 4 5
------+-------+------
3 7 2 | 6 8 9 | 5 1 4
8 1 4 | 2 5 3 | 7 6 9
6 9 5 | 4 1 7 | 3 8 2
```

The result of AC3() on easy_sudoku is as follows:

```
Time consumption: 0.0620s
Result: ? Success
Solution:
4 8 3 | 9 2 1 | 6 5 7
9 6 7 | 3 4 5 | 8 2 1
2 5 1 | 8 7 6 | 4 9 3
------+-------+------
5 4 8 | 1 3 2 | 9 7 6
7 2 9 | 5 6 4 | 1 3 8
1 3 6 | 7 9 8 | 2 4 5
------+-------+------
3 7 2 | 6 8 9 | 5 1 4
8 1 4 | 2 5 3 | 7 6 9
6 9 5 | 4 1 7 | 3 8 2
```

The result of forward_checking() on harder_sudoku is as follows:

```
Time consumption: 0.0082s
Result: ? Success
Solution:
4 1 7 | 3 6 9 | 8 2 5
6 3 2 | 1 5 8 | 9 4 7
9 5 8 | 7 2 4 | 3 1 6
------+-------+------
8 2 5 | 4 3 7 | 1 6 9
7 9 1 | 5 8 6 | 4 3 2
3 4 6 | 9 1 2 | 7 5 8
------+-------+------
```

```
2 8 9 | 6 4 3 | 5 7 1
5 7 3 | 2 9 1 | 6 8 4
1 6 4 | 8 7 5 | 2 9 3
```

The result of AC3() on harder_sudoku is as follows:

```
Time consumption: 0.1205s
Result: ? Success
Solution:
4 1 7 | 3 6 9 | 8 2 5
6 3 2 | 1 5 8 | 9 4 7
9 5 8 | 7 2 4 | 3 1 6
------+-------+------
8 2 5 | 4 3 7 | 1 6 9
7 9 1 | 5 8 6 | 4 3 2
3 4 6 | 9 1 2 | 7 5 8
------+-------+------
2 8 9 | 6 4 3 | 5 7 1
5 7 3 | 2 9 1 | 6 8 4
1 6 4 | 8 7 5 | 2 9 3
```

We can find that forward_checking() is quicker than AC3() because forward_checking() just needs to check neighbours of this variable while AC3() needs to check all arcs.

3 Exercise 3: N-Queens (Hill Climbing)

3.1 min_conflicts()

I choose stochastic to implement min_conflicts() which means I randomly choose conflicted variables instead of choosing max_conflicts variable and assign min_conflicts_value to it. My implementation is as follows:

```python
def min_conflicts(csp, max_steps=100000):
    assignment = {}
    for v in csp.variables:
        csp.assign(v, min_conflicts_value(csp, v, assignment), assignment)

    for i in range(max_steps):
        vars = csp.conflicted_vars(assignment)
        if not vars:
            return assignment
        var = random.choice(vars)
        val = min_conflicts_value(csp, var, assignment)
        csp.assign(var, val, assignment)

    return None
```

Its result on 8_nqueens is as follows:

```
Time consumption: 0.0001s
Result: ▯ Success
Solution:
. - . - . - Q -
- Q - . - . - .
. - . - . Q . -
- . Q . - . - .
Q - . - . - . -
- . - Q - . - .
. - . - . - . Q
- . - . Q . - .
```

3.2 min_conflicts() for Sudoku

The result of min_conflicts() on easy_sudoku is as follows:

```
Time consumption: 46.5223s
Result: ▯ Fail (no solution found).
```

It failed to find the solution because it will easily get stuck in a local optima because there are too many variables in sudoku problems and stochastic will not work well.