Lab 3

姓名：游灏溢
班级：F1903302
时间：3/11/2021

Abstract

This lab is mainly about solving constraint satisfaction problems. We mainly use backtracking method to solve it step by step or hillclimbing method. To accelarate the algorithm, we also add inference function to help reduce the branches in the method.

Content

Exercise1Backtracking Search

Rearrange the Seats

consider the situation that will cause conflict

1. assign 2 people at the same seat
2. assign 1 person at more than 1 seats
3. assign 2 people at the adjacent seats and they are friend.

So I finish the _classroom_conflict() that

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

Backtracking search

I use the induce methods. When I have assigned a series of assignment, I need to care about how to assign another variable. Then I randomly choose a variable that has not been assigned and assign it with a value causing no conflicts to get a new assignment. After that, I test whether there is solution satisfying new assignment. If so, it is also the solution satisfying current assignment, otherwise I test another value until all the values cannot get the solution.

So the implement code is:

```
    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)
                new_assignment=backtrack(assignment)
                if new_assignment is not None:
                    return new_assignment
                else:csp.unassign(var,assignment)
        return None
```

Exercise2

Inference function

Forward Checking

As I tend to assign a value to a variable, I will prune the value in the variable's neighbors and add them into the removals.

And if there is a variable its domains becoming empty during the pruning, return False.

```
def forward_checking(csp, var, value, assignment, removals):
```

```
"""Prune neighbor values inconsistent with var=value."""

csp.support_pruning()  # It is necessary for using csp.prune()

for item in csp.neighbors[var]:

 if value in csp.curr_domains[item]:

    csp.prune(item,value,removals)

 if len(csp.curr_domains[item])==0:

    return False

 return True
```

AC3

Every pair x_i, x_j in the queue means the arc $x_j \to x_i$. Every time pop one pair, I judge whether it is consistent. If so, I prune the domain and return **True** in the revise() , which means I should push all the arc pointing to x_j.

```
def revise(Xi, Xj):
    """Return true if we remove a value."""
    if len(csp.curr_domains[Xi])==1 and csp.curr_domains[Xi][0] in csp.curr_domains[Xj]:
        csp.prune(Xj,csp.curr_domains[Xi][0],removals)
        return True
    else:return False

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 item in csp.neighbors[Xj]:
            queue.add((Xj,item))
return True  # CSP is satisfiable
```

Backtrapping with Inference

When I assign a variable, I will use the inference function to prune the branch and record the pruning in the removals.

When I find assigning the value causes conflicts, I will restore the removals and unassign the variable.

```
def backtrack(assignment):
    if len(assignment) == len(csp.variables):return assignment
    var=select_unassigned_variable(assignment,csp)
    var = select_unassigned_variable(assignment, csp)
    domains=order_domain_values(var, assignment, csp)
    for value in domains:
        removals=[]
        if csp.nconflicts(var,value,assignment)>0:continue
        if not inference(csp,var,value,assignment,removals):
            csp.restore(removals)
            removals.clear()
            continue
        for i in domains:
            if i!=value and i in csp.curr_domains[var]:
                csp.prune(var,i,removals)
        csp.assign(var,value,assignment)
        new_assignment=backtrack(assignment)
        if new_assignment is not None:return new_assignment
        else:
            csp.unassign(var,assignment)
        csp.restore(removals)
    return None
```

Performance Analysis

```
PS C:\上交电院\大三上\ai\lab3\lab3> python main.py --algo backtrack+ac3 --layout easy_sudoku
Time consumption: 0.1187s
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
```

```
PS C:\上交电院\大三上\ai\lab3\lab3> python main.py --algo backtrack+ac3 --layout harder_sudoku
Time consumption: 0.1326s
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
```

```
PS C:\上交电院\大三上\ai\lab3\lab3> python main.py --algo backtrack+fc --layout easy_sudoku
Time consumption: 0.0153s
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
```

```
PS C:\上交电院\大三上\ai\lab3\lab3> python main.py --algo backtrack+fc --layout harder_sudoku
Time consumption: 0.0339s
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
```

After comparation, we can get 2 conclusions.

1. The algorithm of forward checking has better performance than ac3. When I trace the running process of two inference function respectively, I find the ac3 costs lots time at the inference function. Every time I enter it, it need to push all the x_i and its neigbors as pairs into the queue, and call the revise function of these pairs, which costs plenty of time.
2. The layout easy_sudoku costs less time than harder_sudoku for both two algorithm. That's because the domains of the variables in former one are much more limited.

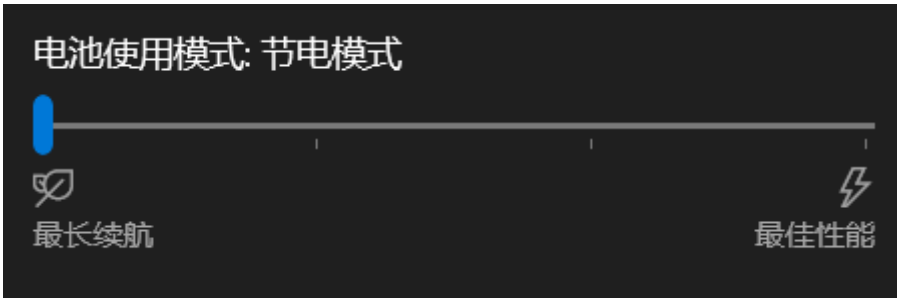
More discussion

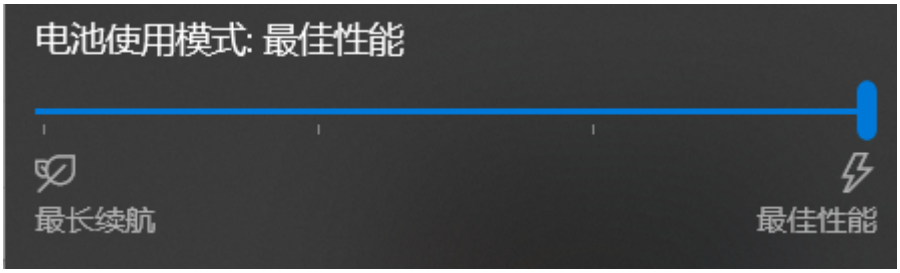
Although this is not included in the homework, I must need to discuss more about this exercise.

Firstly, performance testing method is not rational. It uses the running time of the program to represent the algorithm performance.

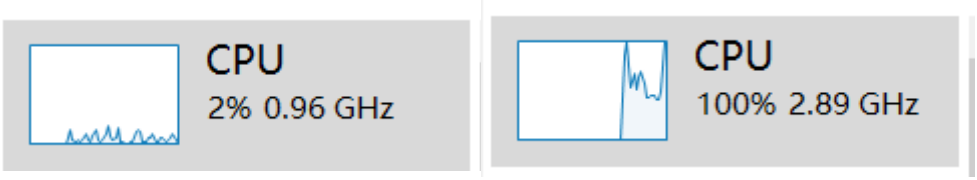
```
start = time.time()
results = algorithm(problem)
print(f"Time consumption: {time.time() - start:.4f}s")
```

However, the running time will be influence by lots of things, such as





or the number of the process holded by the cpu.



Secondly, the api of the program is not well written.

Such as regarding the ac3 function, we can change

```
queue = {(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]}
```

into

```
queue = {(Xi, Xk) for Xi in csp.variables if len(csp.curr_domains[Xi])==1 for Xk in csp.neighbors[Xi]}
```

then the program can be accelarated

```
PS C:\上交电院\大三上\ai\lab3\lab3> python main.py --algo backtrack+ac3 --layout harder_sudoku
Time consumption: 0.0648s
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
```

Exercise 3

Variable Choosing Function

If assignment is not full, we choose a variable that has not been assigned, otherwise, random choose a variable.

```
from random import choice
from .variable_order import mrv
def random_choose(assignment):
    l=csp.variables
    if len(l)==len(assignment):
        return choice(l)
    else:
        return mrv(assignment,csp)
    assignment=dict()
```

Hill Climbing

Assign the variable the value with min conflicts.

The loop ends until it finds the solution or exceeds the limitation.

```
    assignment=dict()
    step=0
    while(len(assignment)<len(csp.variables) or not csp.goal_test(assignment)) and
step<max_steps:
        var=random_choose(assignment)
        val=min_conflicts_value(csp,var,assignment)
        csp.assign(var,val,assignment)
        step+=1
    if csp.goal_test(assignment):
        return assignment
    else: return None
```

Performance

It has worse performance than inference

```
PS C:\上交电院\大三上\ai\lab3\lab3> python main.py --algo hill_climbing --layout easy_sudoku
Time consumption: 0.2354s
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
```

That's because I need to trials are very random. I need to iterate to improve my assignment until it satisfies, but I do not know which variable I need to improve, so I can only randomly choose until I fetch it.