Lab 3: Constraint Satisfaction Problems

> *CS410: Artificial Intelligence*
> *Shanghai Jiao Tong University, Fall 2021*
> *Due Time: 2021.11.04 23:59*

Assignment

In this lab, we are going to implement some algorithms for solving CSPs. Keep in mind that these algorithms are general enough, and the only things you deal with are the variables, domains and constraints. We define these in `csp.py`. Please check how each method of class `CSP` works before you get started.

Exercise 1: Rearrange the Seats (Backtracking Search)

The first exercise is to use backtracking search to rearrange the seats of students. The input is the initial seats of the students in the ''classroom'' and the output is the new seats of students, where the maze in the environment is treated as a "classroom" with $n \times m$ seats (n rows and m columns) and the seats in each line are separated by spaces. In each seat, there is a student represented by a unique integer in the range of $[1, n \times m]$ and there are total $n \times m$ students in this classroom. Student A and student B are adjacent to each other if student B could be reached from student A in any direction of North, South, West, or East. In the initial seats, two students are friends if they are adjacent to each other. We now want to rearrange the seats of these $n \times m$ students such that none of these students are adjacent to his/her friends.

For example, the initial seats of students

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

could be rearranged to

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

1. Implement method `_classroom_conflict()` of class `Classroom` in `classroom.py` to check whether any two pairs of `(variable, value)` will cause conflicts.

   > *Hint: You may want to use* `_is_adjacent()` *and* `_is_friend()` *to implement* `_classroom_conflict()`.

2. Implement method `backtracking()` in `algorithm/backtracking.py` to solve the problem. Its input contains a CSP, a method to order unassigned variables, and a method to order domain values. If a solution is found, `backtracking()` outputs the final assignment stored in a dictionary where each key is a variable and the corresponding value is the assigned value. Otherwise, `backtracking()` returns `None`. Notice that the feasible solution may not be unique and any feasible solution will be accepted.

   > *Hint:*
   >
   > a. *You may want to use* `nconflicts()`, `assign()` *and* `unassign()` *of class* `CSP` *to implement* `backtracking()`.
   > b. *The default variable order and value order is defined in* `mrv()` *in* `algorithm/variable_order.py` *and* `lcv()` *in* `algorithm/value_order.py` *respectively.*

   You can evaluate your algorithm with

   ```
   python main.py --algo backtrack --layout easy_classroom
   python main.py --algo backtrack --layout fail_classroom
   ```

Exercise 2: Sudoku (Filtering)

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

The second exercise is to use backtracking search together with two filtering strategies to fill the Sudoku puzzle. As the given problem has a list of constraints to be satisfied, CSP solutions work efficiently for the Sudoku problem as well.

1. Implement method `forward_checking()` in `algorithm/inference.py`.

   > *Hint: You may want to use* `constraints()`, `prune()` *and* `curr_domains` *of class* `CSP` *to implement* `forward_checking()`.

2. Implement method `AC3()` in `algorithm/inference.py`.

   > *Hint: You may want to use* `constraints()`, `prune()`, `curr_domains` *and* `neighbors` *of class* `CSP` *to implement* `AC3()`.

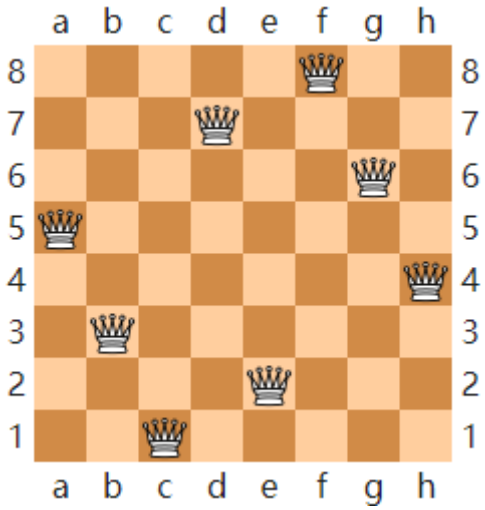
3. Improve your backtracking algorithm with filtering strategies *forward checking* and *constraint propagation* by implementing `backtracking_with_inference()` in `algorithm/backtracking.py`. You can evaluate your algorithm with

   ```
   python main.py --algo backtrack+fc --layout easy_sudoku
   python main.py --algo backtrack+ac3 --layout easy_sudoku
   python main.py --algo backtrack+fc --layout harder_sudoku
   python main.py --algo backtrack+ac3 --layout harder_sudoku
   ```

   > *Hint:*
   >
   > a. *To use the filtering strategies, you only need to do minor adjustments over* `backtracking()`.
   > b. *Method* `forward_checking()` *and* `mac`
   > c. *You may want to use* `suppose()` *and* `restore()` *method of class* `CSP` *to assume a single assignment and restore it.*

4. Compare the efficiency of two filtering strategies for solving Sudoku. Discuss and explain your findings.

Exercise 3: N-Queens (Hill Climbing)



The last exercise is to use hill climbing algorithm to solve the n-queens problem.

1. Implement method `min_conflicts()` in `algorithm/hillclimbing.py`, which uses `min_conflicts_value()` as the value function. You can choose any of the variants (random-restart/stochastic/first-choice) to implement `min_conflicts()`. You can evaluate your algorithm with

   ```
   python main.py --algo hill_climbing --layout 8_nqueens
   ```

   > *Hint: You may want to use* `assign()` *and* `conflicted_vars()` *class* `CSP` *to implement* `min_conflicts()`.

2. Try using `min_conflicts()` for Sudoku problems. Discuss and explain your findings. You can evaluate it with

   ```
   python main.py --algo hill_climbing --layout easy_sudoku
   ```

Submission

Here are the files you need to submit (please do **NOT** rename):

- `classroom.py`
- `backtracking.py`
- `inference.py`
- `hillclimbing.py`
- `report.pdf` for your **brief** report.

Name your **zip** file containing the above files as `xxxxxxxxxxx.zip` with your student ID, and submit it on Canvas.