Problem Set 2: Constraint Satisfaction & Games

The goal of this problem set is to implement and get familiar with CSP and adversarial search algorithms.

To run the autograder, type the following command in the terminal:

```
python autograder.py
```

If you wish to run a certain problem only (e.g. problem 1), type:

```
python autograder.py -q 1
```

where 1 is the number of the problem you wish to run.

To debug your code through the autograder, you should disable the timeout functionality. This can be done via the `debug` flag as follow:

```
python autograder.py -d -q 1
```

Instructions

In the attached python files, you will find locations marked with:

```
#TODO: ADD YOUR CODE HERE
NotImplemented()
```

Remove the `NotImplemented()` call and write your solution to the problem. **DO NOT MODIFY ANY OTHER CODE**; The grading of the assignment will be automated, and any code written outside the assigned locations will not be included during the grading process.

IMPORTANT: Starting from this problem set, you must document your code (explain the algorithm you are implementing in your own words within the code) to get the full grade. Undocumented code will be penalized.

For this assignment, you should submit the following files only:

- `student_info.json`
- `CSP_solver.py`
- `search.py`

Put these files in a compressed zip file named `solution.zip` which you should submit to Google Classroom.

File Structure

There are 5 files that you can run from the terminal:

- `play_sudoku.py`: where you can play a sudoku puzzle or let the solver play it. This is useful for debugging.
- `play_tree.py`: where you can play a tree game or let an agent play it. This is useful for debugging.
- `play_dungeon.py`: where you can play a dunegon level or let an agent play it. This is useful for debugging.
- `autograder.py`: where you test your code and get feedback about your results for the test cases.
- `speed_test.py`: where you can check your computer's speed, which is useful to predict how long your code will take to run on the grading machine.

For Part 1 (CSP), these are the files relevant to the requirements:

- `CSP.py` **[IMPORTANT]**: This is where the generic CSP problem and the constraints are defined. You should understand the code written in it.
- `sudoku.py`: This is where the sudoku problem is defined. You should not need to understand it to finish the requirements, since your code should work for any CSP problem in general as defined in `CSP.py`.
- `CSP_solver.py` **[IMPORTANT + REQUIREMENT]**: This is where you should write the code for the requirements in Part 1.

For Part 2 (Games), these are the files relevant to the requirements:

- `mathutils.py` **[IMPORTANT]**: This contains some useful math utilities that are used in the game. You should understand how to use the classes and functions written in it.
- `game.py` **[IMPORTANT]**: This is where the generic game is defined. You should understand the code written in it.
- `agents.py` **[IMPORTANT]**: This is where game agents are defined. It is recommended that you understand the code written in it.
- `tree.py`: This is where the tree game is defined. You should not need to understand it to finish the requirements, since your code should work for any game in general as defined in `game.py`.
- `dungeon.py`: This is where the dungeon game is defined. You should not need to understand it to finish the requirements, since your code should work for any game in general as defined in `game.py`.
- `search.py` **[IMPORTANT + REQUIREMENT]**: This is where you should write the code for the requirements in Part 2.

There are some files defined in the folder `helpers` that are used for testing. You should not need to understand how to use them, but it won't harm to know the following information about them:

- `globals.py`: This only contains some imports that should be seen by all the testing code, so they are defined here to be imported in `autograder.py`.
- `mt19937.py`: This is a pseudo-random number generator. We define our own instead of use the builtin `random` module to ensure that the results are reproduceable regardless of the Python version.
- `pruned_tree.py`: This contains some utility functions to print the pruned tree onto the console.
- `test_tools.py`: This is where most of the testing code lies. It has pairs of functions `run_*` and `compare_*` to run and check the results of different functions in the requirements. It is relatively complex, and error messages may point you towards this file, if your code returns something wrong that also leads the testing code to crash.

- `utils.py`: This contains some classes and functions that are used by `autograder.py` and `test_tools.py`. This is where the `load_function` lies which is used to dynamically load your solutions from their python files at runtime without using `import`. This ensures that having an error in one file does not stop the autograder from grading the other files.

Part 1: Constraint Satisfaction

Problem Definitions

There is one problem defined in this part of the problem set which is **Sudoku**. In this single-player game, you are given a square grid of size `NxN` where every cell should contain a number in the range `[1, N]`. The grid is divided into `N` non-overlapping square subgrids where each subgrid has an area `N`. Usually, the puzzle will contain some cells which contain fixed values, called clues, and you must fill the rest of the grid such that the value within each cell is unique relative to its row, column and subgrid. If you want to read more about Sudoku, you can look at the game's wikipedia page: https://en.wikipedia.org/wiki/Sudoku.

You can play a sudoku game by running:

```
python play_sudoku.py sudoku\sudoku_9x9_2.txt
```

You can also let the "backtracking search" agent play the game in your place as follow:

```
python play_sudoku.py sudoku\sudoku_9x9_2.txt -a backtrack
```

Problem 1: 1-Consistency

Inside `CSP_solve.py`, modify the function `one_consistency` to implement 1-Consistency. This functio should modify the domains to only include values that satisfy their variables' unary constraints. Then all unary constraints should be removed from the problem (they are no longer needed). The function should return `False` if any domain becomes empty. Otherwise, it should return `True`.

Note: The test code for the next problems relies on the function `one_consistency`, so make sure it works correctly before proceeding to the next problems.

Problem 2: Forward Checking

Inside `CSP_solve.py`, modify the function `forward_checking` to implement Forward Checking. Given the problem, the variable to be assigned, its assigned value and the domains of the unassigned values,this function should return `False` if it is impossible to solve the problem after the given assignment, and `True` otherwise. In general, the function should do the following:

- For each binary constraints that involve the assigned variable:
    - Get the other involved variable.

    - If the other variable has no domain (in other words, it is already assigned a value), skip this constraint.
    - Update the other variable's domain to only include the values that satisfy the binary constraint with the assigned variable.
- If any variable's domain becomes empty, return `False`. Otherwise, return `True`.

Problem 3: Least Restraining Value

Inside `CSP_solve.py`, modify the function `least_restraining_values` to implement the "Least Restraining Value" heuristic. Generally, this function is very similar to the forward checking function, but it differs as follows:

- You are not given a value for the given variable, since you should do the process for every value in the variable's domain to see how much it will restrain the neigbors domain.
- Here, you do not modify the given domains. But you can create and modify a copy.

Problem 4: Minimum Remaining Values

Inside `CSP_solve.py`, modify the function `minimum_remaining_values` to implement the MRV "Minimum Remaining Values" heuristic.

Problem 5: Backtracking Search

Inside `CSP_solve.py`, modify the function `solve` to implement the Backtracking Search algorithm with forward checking. The variable ordering should be decided by the MRV heuristic. The value ordering should be decided by the "least restraining value" heurisitc. Unary constraints should be handled using 1-Consistency before starting the backtracking search. This function should return the first solution it finds (a complete assignment that satisfies the problem constraints). If no solution was found, it should return `None`.

The autograder will track the calls to `problem.is_complete` to check the pruning (forward checking) and compare with the expected output.To get the correct result for the explored nodes, you should check if that assignment is complete **ONLY ONCE** using `problem.is_complete` for every assignment including the initial empty assignment, **EXCEPT** for the assignments pruned by the forward checking. Also, if 1-Consistency deems the problem unsolvable, you shouldn't call `problem.is_complete` at all.

Part 2: Games

Problem Definitions

There are two problems defined in this part of the problem set:

1. **Tree Problem**: where the environment is a tree where the states are nodes, and the actions are edges to the children states. The problem definition is implemented in `tree.py` and the problem instances are included in the `trees` folder.
2. **Dungeon Crawling**: where the environment is a 2D grid in which the player `'@'` has to find a key `'K'` then reach the exit door `'E'`. The dungeon contains monsters `'M'` which will kill the player and the player can collect daggers `'~'` to kill monsters. When the player and a monster meet in the same tile, if

the player has a dagger, it will kill the monster, otherwise the monster kills the player. A dagger can only be used once. The player can also collect coins `'$'` but they are not essential to win the game. Both the player and the monsters cannot stand in a wall tile `'#'` so they have to find a path that consists of empty tiles `'.'`. In addition, no more than one monster can stand on the same tile. The problem definition is implemented in `dungeon.py` and the problem instances are included in the `dungeons` folder.

You can play a tree or a dungeon scavenging game by running:

```
# For playing a tree (e.g. tree1.json)
python play_tree.py tree\tree1.json

# For playing a dungeon (e.g. dungeon1.txt)
python play_dungeon.py dungeons\dungeon1.txt
```

You can also let a search agent play the game in your place (e.g. a Minimax Agent) as follow:

```
python play_dungeon.py dungeons\dungeon1.txt -a minimax
```

For trees, you have to specify the 2nd player (adversary) too:

```
python play_tree.py tree\tree1.json -a minimax -adv random
```

The agent options are:

- `human` where the human play via the console
- `random` where the computer plays randomly
- `greedy` where the computer greedly selects the action that increases the heuristic value (1-step look-ahead).
- `minimax` where the computer uses Minimax search to select the action.
- `alphabeta` where the computer uses Alpha-beta pruning to select the action.
- `alphabeta_order` where the computer uses Alpha-beta pruning with move ordering to select the action.
- `expectimax` where the computer uses Expectimax search to select the action.

To get detailed help messages, run `play_dungeon.py` and `play_graph.py` with the `-h` flag.

Important Notes

The autograder will track the calls to `game.is_terminal` to check the pruning and compare with the expected output. Therefore, **ONLY CALL** `game.is_terminal` **once on each unpruned node in the game tree** in all algorithms. During expansion, make sure to loop over the actions in same order as returned by `game.get_actions` except in `alphabeta_with_move_ordering`. If two actions have the same value, pick the action that appears first in the `game.get_actions`.

Problem 6: Minimax Search

Inside `search.py`, modify the function `minimax` to implement Minimax search. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

Problem 7: Alpha Beta Pruning

Inside `search.py`, modify the function `alphabeta` to implement Alpha Beta pruning. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

Problem 8: Alpha Beta Pruning with Move Ordering

Inside `search.py`, modify the function `alphabeta_with_move_ordering` to implement Alpha Beta pruning with move ordering. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

IMPORTANT NOTE: If two children have the same heuristic value, they should be explored in the same order as they were in the original order. In other words, the sorting must be stable.

Problem 9: Expectimax Search

Inside `search.py`, modify the function `expectimax` to implement Minimax search. The return value is a tuple containing the expected value of the game tree and the best action based on the algorithm. If the given state is terminal, the returned action should be `None`.

IMPORTANT NOTE: For the chance nodes, assume that all the children has the same probability. In other words, the probability of a child is 1 / (the number of children).

Time Limit

In case your computer has a different speed compared to mine (which I will use for testing the submissions), you can use the following information as a reference: `On my computer, running speed_test.py takes ~17 seconds`. You can measure the run time for this operation on computer by running:

```
python speed_test.py
```

If your computer is too slow, you can increase the time limit in the testcases files.

Delivery

The delivery deadline is `Monday December 5th 2022 23:59`. It should be delivered on **Google Classroom**. This is an individual assignment. The delivered code should be solely written by the student who delivered it.

Any evidence of plagiarism will lead to receiving **zero** points.