Computer Science 384 Thursday, October 7, 2021
St. George Campus University of Toronto

Homework Assignment #2: Constraint Satisfaction Problems
Due: Thursday, November 04, 2021 by 11:00 PM

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due, i.e. no question about this assignment will be answered, whether it is asked on the discussion board, via email or in person.*

Late Policy: 10% per day after the use of 3 grace days.

Total Marks: This assignment represents 11% of the course grade.

Handing in this Assignment

What to hand in on paper: Nothing.

What to hand in electronically: You must submit your assignment electronically. Download the zip file that is called `csp.zip` from Quercus, which contains the files `backtracking.py`, `csp_problems.py`, `constraints.py`, `csp.py`, `util.py`, `nqueens.py`, `sudoku.py`, `plane_scheduling.py`, and `autograder.py`. You must modify the files `backtracking.py`, `csp_problems.py` and `constraints.py` so that they solve the problems specified in this document. Then, submit your modified `backtracking.py`, `csp_problems.py` and `constraints.py` files and your signed acknowledgment form using MarkUs. Your login to MarkUs is your teach.cs username and password. You can submit a new version of any file at any time, though the lateness penalty applies if you submit after the deadline. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission.

We will test your code electronically. You will be supplied with a testing script that will run a **subset** of the tests. If your code fails all of the tests performed by the script (using Python version 3.9.7), you will receive a failing grade on the assignment.

When your code is submitted, we will run a more extensive set of tests which will include the tests run in the provided testing script and a number of other tests. You have to pass *all* of these more elaborate tests to obtain full marks on the assignment.

Your code will not be evaluated for partial correctness; it either works or it doesn't. It is your responsibility to hand in something that passes at least some of the tests in the provided testing script.

- *Make certain that your code runs on teach.cs using python3 (version 3.9.7) using only standard imports.* This version is installed as "python3" on teach.cs. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- *Do not add any non-standard imports from within the python file you submit (the imports that are already in the template files must remain).* Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- *Do not change the supplied starter code.* Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks.

Clarifications: Important corrections (hopefully few or none) and clarifications to the assignment will be announced on Quercus. *It is your responsibility to monitor for any clarifications or corrections to the assignment.*

Help Sessions: There will be several help sessions for this assignment. Dates and times for these sessions will be posted to Quercus ASAP.

Questions: Questions about the assignment should be asked on Piazza.

If you have a question of a personal nature, please email your course instructors (csc384-2021-09 at cs.toronto.edu). Make sure to place [CSC384] and A2 in the subject line of your message.

1 Introduction and Starter Code

In this assignment, you will implement constraints and backtracking search algorithms that will be used to solve the nQueens problem and sudoku puzzles. Additionally, you will encode a CSP to solve a problem related to scheduling airplanes (described in Sec. 2.6).

The code for this assignment contains the following files:

Files you will edit:

- `backtracking.py` - File where all of the code related to backtracking search is located. You will implement forward checking and GAC search in this file.
- `csp_problems.py` - File where all of the code related implementing different CSP problems is located. You will implement a new version of the nQueens CSP and a CSP to solve the plane scheduling problem (defined below) in this file.
- `constraints.py` - File where all of the code related implementing various constraints is located. You will implement the NValues constraint in this file.

Files you will *not* edit:

- `csp.py` - File containing the definitions of Variables, Constraints, and CSP classes.
- `util.py` - File containing basic utility functions.
- `nqueens.py` - File for solving nQueens problems.
- `sudoku.py` - File for solving sudoku problems.
- `plane_scheduling.py` - File for solving plane scheduling problems
- `autograder.py` - Program for evaluating your solutions. As always, your solutions will also be evaluated with additional tests besides those performed by the autograder.

Files to Edit and Submit: You will fill in portions of `backtraking.py`, `constraints.py`, and `csp_problems.py` during the assignment. You may also add other functions and code to these file so as to create a modular implementation. Do **NOT** add code to other files and import them. Those files cannot be submitted so your solution relying on them will fail our testing. Please do not change the other files in the distribution.

2 Problems

2.1 Implementing a Table Constraint (4 points)

The file `backtracking.py` already contains an implementation of BT (plain backtracking search) while `csp_problems.py` contains an implementation of the nQueens problem. Try running the following command to solve the 8 queens problem

```
python3 nqueens.py 8
```

If you run the following

```
python3 nqueens.py -c 8
```

the program will find all solutions to the 8-Queens problem. To see the other arguments you can use, run the following command:

```
python3 nqueens.py --help
```

Note that FC and GAC will be implemented in subsequent questions, so these algorithms will not be functional yet.

For this question, you will implement the class `QueensTableConstraint` inside the file `constraint.py`. This class creates table constraints representing the constraints between queens in two different rows for the `nQueens` problem. Do not change the existing function signatures. Once implemented, you can solve the `nQueens` CSP using your table constraint implementation by running

```
python3 nqueens.py -t 8
```

Check a number of sizes and '-c' options: you should get the same solutions returned irrespective of whether or not you use '-t'. That is, your table constraint should yield the same behavior as the original `QueensConstraint`.

2.2 Forward Checking (5 points)

In `backtracking.py`, you will find the unfinished function FC. For this question, you must complete this function. The essential subroutine `FCCheck` has already been implemented for you. Note that your implementation must deal correctly with finding one or all solutions. You can check how this is done in the already implemented BT algorithm (be sure that you restore all pruned values even if FC is terminating after one solution).

After implementing FC you will be able to run the following

```
python3 nqueens.py -a FC 8
```

to solve 8-Queens with forward checking. Solve some different sizes and check how the number of nodes explored differs from when BT is used.

Also, try solving the sudoku puzzle using the command

```
python3 sudoku.py 1
```

This command will solve sudoku board #1 using Forward Checking.

To see the other arguments you can use, run the command

```
python3 sudoku.py --help
```

To see how BT performs compared to FC, try the command

```
python3 sudoku.py -a 'BT' 1
```

Finally, to find all solutions using FC, use the command

`python3 sudoku.py -a 'FC' -c 1`

Check if any of the boards 1-7 have more than one solution.

Note also that if you have a sudoku board you would like to solve, you can easily add it into `sudoku.py` and solve it. Look at the code in this file to see how input boards are formatted and placed in the list boards. Once a new board is added to the list boards it can be solved with the command `python3 sudoku.py -a 'FC' k` where k is the position of the new board in the list `boards`.

2.3 GacEnforce and GAC (7 points)

In `backtracking.py`, you will find unfinished `GacEnforce` and `GAC` routines. For this question, you must complete these functions.

After finishing these routines you will be able to run the command

`python3 nqueens.py -a GAC 8`

Try different numbers of Queens and see how the number of nodes explored differs from when you run FC. Does GAC also take less time than FC on sudoku? What about on nQueens?

Now try running

`python3 sudoku.py -e 1`

which will not do any backtracking search, it will only run `GacEnforce` at the root.

Try running only `GacEnforce` on each board to see which ones are solved by only doing `GacEnforce`.

2.4 AllDiff for Sudoku (2 points)

In `csp_problems.py`, you will find the function `sudokuCSP`. This function takes a model parameter that is either `'neq'` or `'alldiff'`. When `model == 'neq'`, the returned CSP contains many binary not-equals constraints. But when `model == 'alldiff'` the model should contain 27 `allDifferent` constraints.

For this question, complete the implementation of `sudokuCSP` so it properly handles the case when `model == 'alldiff'` using allDifferent constraints instead of binary not-equals.

Note that this question is straightforward as you can use the class `AllDiffConstraint(Constraint)` that is already implemented in `constraints.py`. However, you must successfully complete Question 3 to get any marks on this question.

2.5 NValues Constraint (4 points)

The `NValues` Constraint is a constraint over a set of variables that places a lower and an upper bound on the number of those variables taking on value from a specified set of values.

In `constraints.py` you will find an incomplete implementation of the class `NValuesConstraint`. In particular, the function `hasSupport` has not yet been implemented. For this question, complete the implementation of `NValuesConstraint`.

2.6 Plane Scheduling (10 points)

For this question, implement a solver for the plane scheduling problem described below by encoding the problem as a CSP and using your already developed code to find solutions.

Plane scheduling problem: You have a set of planes, and a set of flights each of which needs to be flown by some plane. The task is to assign to each plane a sequence of flights so that:

1. Each plane is only assigned flights that it is capable of flying (e.g., small planes cannot fly trans-Atlantic flights).
2. Each plane's initial flight can only be a flight departing from that plane's initial location.
3. The sequence of flights flown by every plane must be feasible. That is, if F2 follows F1 in a plane's sequence of flights then it must be that F2 can follow F1 (normally this would mean that F1 arrives at the same location that F2 departs).
4. Certain flights terminate at a maintenance location. All planes must be serviced with a certain minimum frequency. If this minimum frequency is K then in the sequence of flights assigned to a plane at least one out every subsequence of K flights must be a flight terminating at a maintenance location. Note that if a plane is only assigned J flights with $J < K$, then it satisfies this constraint.
5. Each flight must be scheduled (be part of some plane's sequence of flights). And no flight can be scheduled more than once.

Example:
Say we have two planes AC-1 and AC-2, and 5 flights AC001, AC002, AC003, AC004, and AC005. Further:

1. AC-1 can fly any of these flights while AC-2 cannot fly AC003 (but can fly the others).
2. AC-1 can start with flight AC001, while AC-2 can start with AC004 or AC005.
3. AC002 can follow AC001, AC003 can follow AC002, and AC001 can follow AC003 (these form a one-way circuit since we can't, e.g., fly AC003 first then AC002). In addition, AC004 can follow AC003, AC005 can follow AC004, and AC004 can follow AC005.
4. AC003 and AC005 end at a maintenance location.
5. The minimum maintenance frequency is 3.

In this case a legal solution would be for AC-1's schedule to be the sequence of flights [AC001, AC002, AC003, AC004] while AC-2's schedule is [AC005] (notice that for AC-1 every subsequence of size 3 at least one flight ends at a maintenance location). Another legal schedule would be for AC-1 to fly [AC001, AC002, AC003, AC004, AC005] and AC-2 to fly [] (i.e., AC-2 does do any flights).

Your task is to take a problem instance, where information like that given in the above example is specified, and build a CSP representation of the problem. Then, solve the CSP using any of the search algorithms, and from the solution, extract a legal schedule for each plane. Note that the set of constraints you have (and have built in the previous questions) are sufficient to model this problem (but feel free to implement further constraints if you need them for the CSP model you develop).

See `plane_scheduling.py` for the details of how problem instances are specified; `csp_problems.py` contains the class `PlaneProblem` for holding a specific problem.

You are to complete the implementation of `solve_planes` in the file `csp_problems.py`. This function takes a `PlaneProblem`, constructs a CSP, solves that CSP with backtracking search, converts the solutions of the CSP into the required format (see the `solve_planes` starter code for a specification of the output format) and then returns the solutions.

You can test your code with `plane_scheduling.py`. The command

```
python3 plane_scheduling.py -a GAC -c K
```

where K is the problem number, will invoke your code (from `csp_problems.py`) on the specified problem. (Use the command ‘ `python3 plane_scheduling.py --help`’ for further information). It can be particularly useful to test your code on problems 1-4 as these problems only test one of the constraints you have to satisfy.

A Few Hints:
First, you should decide on the variables and the domain of values for these variables that you want to use in your CSP model. You should design your variables so that it makes it easy to check the constraints. Avoid variables that require an exponential number of values: performing GAC on such constraints will be too expensive. A number of values equal to the number of flights times number of planes values would be ok.

Try not to use table constraints over large numbers of variables. Table constraints over two or three variables are fine: performing GAC on table constraints with large numbers of variables becomes very expensive.

In some models it is useful to observe that if plane P can fly up to K different flights, then the length of its sequence of flights is at most K. For example, in the example above, AC-1 can fly 5 different flights while AC-2 can fly 4 different flights. So clearly, the sequence of flights flown by AC-1 can’t be more than 5 long, and for AC-2 in sequence can’t be more than 4 long.

As an example of a set of variables and values that would be inadequate, consider having a variable for every flight with values being the set of planes that can fly that flight. This a reasonable number of variables, and it makes the last constraint (i.e., that every flight is scheduled only once) automatically satisfied (since every variable can only have one value). However, these variables by themselves will not be sufficient, as we won’t be able to determine the sequencing of the set of flights assigned to a particular plane. Potentially, such variables might be useful, but other variables would have to be added to model the sequencing part of the CSP.

3 Autograder

You can run the autograder with the following command:

```
python3 autograder.py -q qn
```

where `qn` is one of `q1`, `q2`, `q3`, `q4`, `q5` or `q6`. You can also run the grader on all questions together with the following command:

```
python3 autograder.py
```

The autograder does not generate any graphical output.

GOOD LUCK!