Computer Science 384 February 25, 2019 (UPDATED MARCH 1)
St. George Campus University of Toronto

Homework Assignment #3: Constraint Satisfaction
Due: March 12, 2019 by 10:00 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 part of the 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 assignment files from the A3 web page. Modify `backtracking.py`, `csp_problems.py`, and `constraints.py` appropriately so that they solve the problems specified in this document. **Submit your modified** `backtracking.py`**,** `csp_problems.py`**, and** `constraints.py` **files.** You must also fill in and submit the acknowledgment form, `acknowledgment_form.pdf`.

How to submit: If you submit before you have used all of your grace days, you will submit your assignment using MarkUs. Your login to MarkUs is your teach.cs username and password. It is your responsibility to include all necessary files in your submission. 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. More detailed instructions for using Markus are available at:
`http://www.teach.cs.toronto.edu/~csc384h/winter/markus.html`.

Extra Information

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 3 Clarification page.[1] You are responsible for monitoring the A3 Clarification page.

Questions: Questions about the assignment should be asked on Piazza.[2] If you have a question of a personal nature, please email one of the A3 TAs, Bryan Chan, at **chanb** at cs dot toronto dot edu or Maayan Shvo at **maayanshvo** at cs dot toronto dot edu. You can also email an instructor. In any case, please place 384 and A3 in the subject line of your message.

---

[1] `http://www.teach.cs.toronto.edu/~csc384h/winter/Assignments/A3/a3_faq.html`.
[2] `https://piazza.com/utoronto.ca/winter2019/csc384`.

Evaluation Details

Your code will be autograded for technical correctness. The tests in `autograder.py` will be run. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. If your code fails all of the tests performed by the script (using Python version 3.7), you will receive zero marks. **It's up to you to create test cases to further test your code—that's part of the assignment!**

When your code is submitted, **we will run a more extensive set of 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 3.7) 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.

Introduction

There are six questions in this assignment. Two problems will ask you to implement two constraint propagators—a Forward Checking constraint propagator, and a Generalized Arc Consistence (GAC) constraint propagator. Three problems will ask you to encode constraints. The final problem will ask you to encode a CSP using some of your encoded constraints.

What is supplied

Files you will modify:

- **backtracking.py**. 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**. Where all of the code related to implementing different CSP problems is located. You will implement a new version of the nQueens CSP and a CSP to solve the class scheduling problem in this file.
- **constraints.py**. Where all of the code related implementing various constraints is located. You will implement an NValues constraint in this file.

Files you can ignore:

- **csp.py**. File containing the definitions of Variables, Constraints, and CSP classes.
- **util.py**. Some basic utility functions.

- **nqueens.py**. Solves nQueens problems.
- **class_scheduling.py**. Solves class scheduling problems.
- **autograder.py**. Program for evaluating your solutions. As always your solution might also be evaluated with additional tests besides those performed by the autograder.

Question 1: Implementing a Table Constraint (worth 10/100 marks)

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:

```
python3 nqueens.py 8
```

to solve the 8 queens problem using BT. If you run:

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

the program will find all solutions to the 8-Queens problem. Try:

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

to see the other arguments you can use. (However, you haven't implemented FC nor GAC yet, so you can't use these algorithms yet.) Try some different small numbers with the '-c' option, to see how the number of solutions grows with the number of Queens. Also observe that even numbered queens are generally faster to solve, and the time to find a single solution for 'BT' grows quite quickly. Observe the number of nodes explored. Later once you have FC and GAC implemented you will see that they explore fewer nodes.

For this question look at *constraints.py*. There you will find the class QueensTableConstraint that you have to implement for this question. This class creates a table constraint to capture the nQueens constraint. This table constraint will explicitly store sets of all satisfying tuples of values that can be assigned to the variables. Once you have implemented QueensTableConstraint you can run:

```
python3 nqueens.py -m "table" 8
```

to solve the nQueens CSP using your table constraint implementation.

For nQueens problem, if '-c' is provided, then all solutions will be generated. It generates only one solution by default. The '-m' option specifies which constraint setting to use (row, table, alldiff). We can also change the algorithm used by providing '-a' option (BT, FC, GAC). Check a number of sizes using the '-c' option: you should get the same solutions returned irrespective of whether or not you use the '-m' option. That is, your table constraint should yield the same behavior as the original QueensConstraint.

Question 2: Forward Checking **(worth 20/100 marks)**

In *backtracking.py* you will find the unfinished function **FC**. You have to complete this function. Note that the essential subroutine FCCheck has already been implemented for you. Note that your implementation must deal correctly with finding one or all solutions. Check how this is done in the already implemented BT algorithm. Be sure that you restore all pruned values even if **FC** terminates after one solution.

After implementing **FC** you will be able to run:

```
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.

Question 3: GacEnforce and GAC **(worth 20/100 marks)**

In *backtracking.py* you will also find unfinished **GacEnforce** and **GAC** routines. Complete these functions.

After finishing these routines you will be able to run:

```
python 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 take less time than FC on nqueens?

Question 4: AllDiff and Neq for nQueens **(worth 5/100 marks)**

In *csp_problems.py* you will find the function **nQueens**. This function takes a model parameter that is either 'row', 'table' or 'alldiff'. When model == 'alldiff' the returned CSP should contain a single n-ary all different constraint and many binary not-equals constraints.

Complete the implementation of **nQueens** so it properly handles the case when model == 'alldiff' using allDifferent constraints and binary not-equals constraints.

Note that this question should not take much time as you can use the class *AllDiffConstraint(Constraint)* and the class *NeqConstraint(Constraint)*; these are already implemented in *constraints.py*.

Do you see a performance difference between 'alldiff' and 'row'? Why is one better than the other?

Question 5: NValues Constraint (worth 10/100 marks)

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. For example, if we have 3 variables V1, V2, V3, each with domain [1, 2, 3, 4], then the call `NValuesConstraint('test_nvalues', [V1, V2, V3], [1, 3], 1, 2)` will only be satisfied by assignments such that at least 1 of the V1, V2, V3 is assigned the value 1 or 3, and at most 2 of them have been assigned the value 1 or 3.

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

Question 6: Class Scheduling (worth 35/100 marks)

Implement a solver for the following class scheduling problem by encoding the problem as a CSP and using your already developed code to find solutions.

A student is struggling to schedule her course schedule. She has a list of courses that she wishes to take. Her schedule, like every student's schedule, has some arbitrary number of available time slots (e.g. if a student can only have classes from 9am to 5pm, there will 40 time possible slots during the week). Classes on her list, like every class, are represented by strings that conform to the following pattern:

```
<course_code>-<building>-<time_slot>-<type>-<section_number>
```

The 'type' of class can be either a lecture or a tutorial. Your task is to assign this student a schedule of lectures and tutorials such that:

1. The student takes all the courses on her list.
2. For any given course code, the student can only enrol in one lecture and one tutorial. In addition, the student's tutorial for a given course must be **after** the lecture (e.g. if the lecture is at time slot 1, the tutorial must be after time slot 1).
3. The student must have time to get from one class to another. However, some buildings are too far apart so the student cannot make it under 10 minutes (e.g. FG to FE). The student is therefore not allowed to put these two classes in consecutive time slots. Note that buildings in close proximity to one another will be encoded in an adjacency list that will accompany each scheduling problem (refer to the starter code).
4. Since the student is human, she must be rested with a certain minimum frequency. If this minimum frequency is K, then in her schedule, at least one in every consecutive set of K time slots must not have a class. If the number of courses the student must take is less than K, the constraint will be satisfied trivially.

As an example, suppose there are 6 time slots in the student's schedule. There are 3 buildings: BA, SF, MP. Within 10 minutes the student can go between BA and MP, between BA and SF, but not between SF and MP. The student's minimum rest frequency is 5. There are two courses she must take: CSC148, and CSC165, with the following sections:

1. CSC148-SF-3-LEC-01

2. CSC148-BA-4-TUT-01
3. CSC165-BA-2-LEC-01
4. CSC165-MP-2-LEC-02
5. CSC165-MP-5-TUT-01
6. CSC165-SF-6-TUT-02

We will represent breaks in the schedule using the string `NOCLASS`. A valid solution, then, will be:

[NOCLASS, CSC165-BA-2-LEC-01, CSC148-SF-3-LEC-01, CSC148-BA-4-TUT-01, CSC165-MP-5-TUT-01, NOCLASS]

Another valid solution is:

[NOCLASS, CSC165-BA-2-LEC-01, CSC148-SF-3-LEC-01, CSC148-BA-4-TUT-01, NOCLASS, CSC165-SF-6-TUT-02]

Your task is to build a CSP representation of this problem. You will then solve the CSP using any of the search algorithms you have implemented and from the solution extract a legal schedule for any given student. Note that the set of constraints you have (and have built as per previous questions) are sufficient to model this problem.

See *class_scheduling.py* for the details of how problem instances will be specified; *csp_problems.py* contains the class **ScheduleProblem** that will hold a specific problem.

You are to complete the implementation of **solve_schedules** in the file csp_problems.py. This function takes a **ScheduleProblem**, constructs a CSP, solves that CSP with backtracking search, converts the solutions of the CSP into the required format (see the **solve_schedules** starter code for a specification of the output format) and then returns solutions. You can also test your code with *class_scheduling.py*. The command:

```
python3 class_scheduling.py -a GAC -c N
```

where N is a problem number (ranging from 1 to 7), will invoke your code (from *csp_problems.py*) on the specified problem. (Use `python3 class_scheduling.py --help` for further information). It can be particularly useful to test your code on problems 1-4 in the starter code, as these problems each only test one of the constraints you have to satisfy.

HAVE FUN and GOOD LUCK!