Assignment 3: CSPs

Posted Oct 31st, due Friday Nov 16th @ 11:59pm

Table of Contents

Introduction

In this project, you will implement some new constraints and backtracking search algorithms.

Note that this code base is unrelated to the Berkeley pacman code base. So you will not need any of the files from A1 nor A2. (If anyone has a good idea as to how to use CSPs within the pacman framework please let us know).

The code for this project contains the following files, available as a zip archive.

Files you'll edit:

`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 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 in this file.
`constraints.py`	Where all of the code related implementing various constraints is located. You will implement the 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`	Solve nQueens problems.
`sudoku.py`	Solve sudoku problems.
`plane_scheduling.py`	Solve plane 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.

Files to Edit and Submit: You will fill in portions of `backtraking.py`, `csp.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. You will submit these file with your modifications. Please *do not* change the other files in this distribution.

Evaluation: 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.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. There will be scheduled help sessions (to be announced), the piazza discussion forum will be monitored and questions answered, and you can also ask questions about the assignment during office hours. These things are for your support; please take advantage of them. If you can't make our office hours, let us know and we will arrange a different appointment. We want the assignment to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Piazza Discussion: Please be careful not to post spoilers.

What to Submit

You will be using MarkUs to submit your assignment. You will submit three files:

1. Your modified `backtracking.py`
2. Your modified `csp_problems.py`
3. Your modified `constraints.py`
4. A signed copy of the following acknowledgment

Note: In the various parts below we ask a number of questions. You do not have to hand in answers to these questions, rather these questions are

designed to help you understand the material.

AutoGrader

`autograder.py` is not the same as the Berkeley autograder. You can only run the command

```
python autograder.py -q qn
```

where `qn` is one of `q1`, `q2`, `q3`, `q4`, or `q5`.

Or you can run the grader on all questions together with the command

```
python autograder.py
```

CSPs

Question 1 (4 points): Implementing a Table Constraint

`backtracking.py` already contains an implementation of BT (plain backtracking search) while `csp_problems.py` contains an implementation of the nQueens problem. Try running

```
python nqueens.py 8
```

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

```
python nqueens.py -c 8
```

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

```
python 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 `constraint.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. Once you have that implemented you can run

```
python nqueens.py -t 8
```

to solve the nQueens CSP using your table constraint implementation. 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`

Question 2 (5 points): Forward Checking

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...just 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

```
python 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 sudoku using the command

```
python sudoku.py 1
```

Which will solve board #1 using Forward Checking. Try other boards (1 to 7). Use

```
python sudoku.py --help
```

to see the other arguments you can use.

Also try

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

to see how BT performs compared to FC. Finally try

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

To find all solutions using FC. 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 `python`

`sudoku.py -a 'FC' k` where `k` is the position of the new board in the list`boards`

Question 3 (7 points): GacEnforce and GAC

In `backtracking.py` you will 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` also take less time than `FC` on `sudoku`? What about on `nqueens`?

Now try running

```
python 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`.

Question 4 (2 points): AllDiff for Sudoku

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.

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 **very easy** 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.

Question 5 (4 points): NValues Constraint

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 `class NValuesConstraint`. In particular, the function `hasSupport` has not yet been implemented. Complete this implementation.

Question 6 (10 points): Plane Scheduling

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

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

```
python 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 `python 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, 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.

Submission

You're not done yet! You will also need to submit your code and signed acknowledgment to MarkUs.