Artificial Intelligence	**Assignment 4**
comp440/comp557 Fall 2018	**8 October 2018**

The assignment is due October 21 at 8 pm on Canvas. Download `assignment4.zip` *from Canvas. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. All code should be included in* `submission.py` *at the labeled points. For Problem 3, please run the auto grader using the command line* `python grader.py` *and report the results in your writeup. Problem 5 is optional for comp440 and required for comp557. Upload* `writeup.pdf` *and* `submission.py` *and* `profile.txt` *onto Canvas by the due date and time. One submission per group, please.*

1 Sudoku and constraint satisfaction (20 points)

The game of Sudoku consists of filling a 9x9 grid with digits 1 through 9. The 9x9 grid is divided into nine 3x3 grids shown with the darker borders in Figure 1. Each row, each column, and each of the nine 3x3 grids should contain each of the nine digits exactly once. A specific instance of the Sudoku game is one in which certain digits have already been placed as in the previous figure. The object of this exercise is to use backtracking depth-first search with forward checking and arc-consistency to solve Sudoku puzzles. Assume we formulate 9x9 Sudoku as a CSP with 81 variables, $X_{ij}, 1 \leq i \leq 9, 1 \leq j \leq 9$, where X_{ij} denotes the grid cell at row i and column j. The domain of each of these variables is the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The top left corner is (1,1).

Figure 1: An easy NYT Sudoku puzzle

- (5 points) Using the variable names above, write down all the constraints defining any partially filled Sudoku board (not just the one in Figure 1 which is offered as an example only). You can use constraints of the form $V = v$, $V \neq v$, as well as the *AllDiff(S)* constraint. V stands for a variable and v for a value. S is a subset of variables and *AllDiff* takes a set of variables S and enforces the fact that they are all distinct; it is a compact packaging of all pairwise inequality constraints. *AllDiff*(V_1, V_2, V_3) is short for $\{V_1 \neq V_2, V_1 \neq V_3, V_2 \neq V_3\}$.
- (5 points) What does forward checking with an initial board do? Answer this question in general and then illustrate your answer using the variable X_{64} in the board in Figure 1.
- (5 points) Variable ordering is an important part of solving CSPs in general, and Sudoku puzzles in particular. Explain how the most-constrained variable heuristic would work in general for Sudoku puzzles. Then, use your heuristic to determine which squares(s) will be assigned first in the example board in Figure 1.
- (5 points) Consider variable X_{48} in Figure 1. Why can we assign it the value of 7? Can forward checking and arc consistency derive this from the given constraints? Explain why or why not.

2 Constraint satisfaction with n-ary factors (20 points)

(Problem 6.6 from your textbook)

- (10 points) Show how a single ternary constraint such as $A + B = C$ can be turned into three binary constraints using an auxiliary variable. You may assume finite domains. Hint: Consider a new variable that takes on values that are pairs of other values, and consider constraints such as "X is the first element of the pair Y". Define the new variable, its domains and the binary factors introduced.
- (7 points) Show how constraints with more than three variables can be treated similarly. For example, consider a constraint like $A + B = C + D$. Again, assume finite domains. Define the new variable(s), domains and binary factors.
- (3 points) Show how unary constraints can be eliminated by altering the domains of variables.

Now you have demonstrated that any CSP can be transformed into a CSP with only binary constraints.

3 Constraint solving and course scheduling (100 points)

What courses should you take in a given semester? Answering this question requires balancing your interests, satisfying prerequisite chains, graduation requirements, availability of courses; this can be a complex and tedious process. In this assignment, you will write a program that does automatic course scheduling for you based on your preferences and constraints. The program will cast the course scheduling problem as a constraint satisfaction problem (CSP) and then use backtracking search to solve that CSP to give you your optimal course schedule.

We have already implemented a basic backtracking search for solving weighted CSPs with unary and binary factors. First, in Problem 3.1, you will write three heuristics that will make the CSP solver much faster. In Problem 3.2, you will add helper functions to handle more complex non-binary factors. Finally, in Problem 3.3, you will create the course scheduling CSP and solve it using the solver created in Problems 3.1 and 3.2.

3.1 CSP solving (40 points)

We will be working with weighted CSPs, which associate a weight with each assignment x based on the product of m potential functions or factors.

$$weight(x) = \prod_{j=1}^{m} f_j(x)$$

where each potential $f_j(x) \geq 0$. For unweighted CSPs, $f_j(x) \in \{0, 1\}$. Our goal is to find the assignment(s) with the highest weight. In this problem, we will assume that each potential is either a unary potential (depends on exactly one variable) or a binary potential (depends on exactly two variables). We will refer to potentials and factors interchangeably.

Backtracking search operates over partial assignments and associates a weight with each partial assignment, which is the product of all the potentials that depend only on the assigned variables in x. When we assign a value to a new variable X_i, we multiply in all the potentials that depend only on X_i and the previously assigned variables. The function `get_delta_weight()` returns the contribution of these new potentials based on the unary potentials and binary potentials in the CSP. An important case is when `get_delta_weight()` returns 0. In this case, the weight of any full assignment that extends the new partial assignment will also be zero, so there is no need to search further from that new partial assignment.

We have implemented a basic backtracking solver for you. You can try it out in a Python shell on the simple Australia map coloring problem (this is also provided in `run_p1.py`):

```
import util, submission
csp = util.create_map_coloring_csp()
alg = submission.BacktrackingSearch()
alg.solve(csp)
print alg.optimalAssignment
```

Look at the function `BacktrackingSearch.reset_results()` in `submission.py` to see the other fields which are set as a result of solving the weighted CSP. You should read `util.CSP` and `submission.BacktrackingSearch` carefully to make sure that you understand how the backtracking search works on this CSP.

- (5 points) Let's create a CSP to solve the n-queens problem: Given an $n \times n$ board, we would like to place n queens on this board such that no two queens are on the same row, column, or diagonal. Implement `create_nqueens_csp()` by adding variables and some number of binary potentials. You can refer to the CSP examples we provided in `util.py` for guidance. You

can also run the code below to check your implementation. Note that the solver collects some basic statistics on the performance of the algorithm. You should take advantage of these statistics for debugging and analysis. You should get 92 (optimal) assignments with exactly 2057 operations (number of calls to `backtrack()`). Hint: If you get a larger number of operations, make sure your CSP is minimal.

```
import util, submission
csp = submission.create_nqueens_csp()
alg = submission.BacktrackingSearch()
alg.solve(csp)
print alg.optimalAssignment
```

Alternatively, you can run `python run_p1.py` and check the correctness of your CSP implementation.

- (10 points) You might notice that our search algorithm explores quite a large number of states even for the 8×8 board. Let us see if we can do better. One variable-ordering heuristic to consider is the most constrained variable (MCV). To choose an unassigned variable, pick the X_j that has the fewest number of values v which are consistent with the current partial assignment (`get_delta_weight` on $X_j = v$ returns a non-zero value). Implement this heuristic in `get_unassigned_variable()` under the condition `self.mcv = True`. You should observe a non-trivial reduction in the number of states explored.

- (10 points) A heuristic for ordering values of a variable's domain is the least constraining value (LCV). Given the next variable to be assigned X_j, sort its domain values a in descending order of the number of values of an unassigned neighboring variable X_k that is consistent with it (consistent means the binary potential on $X_j = a$ and $X_k = b$ is non-zero). Note that you should only count values b of X_k which are already consistent with the existing partial assignment. Implement this heuristic in `get_ordered_values()` under the condition `self.lcv = True`. You will need to use `binaryPotentials` in CSP.

- (15 points) So far, our heuristics have only looked at the local effects of a variable or value. Let us now implement arc consistency (AC-3). After we set variable X_j to value a, we remove the values b of all neighboring variables X_k that could cause arc-inconsistencies. If X_k's domain has changed, we use X_k's domain to remove values from the domains of its neighboring variables. This is repeated until no domains have changed. This domain pruning could significantly reduce the branching factor of backtracking search, although at some cost. Please fill in `arc_consistency_check()` and `backtrack()` to complete the implementation of arc consistency. You might find `copy.deepcopy()` on `self.domains` useful. You should make sure that your existing MCV and LCV implementation are compatible with your AC-3 algorithm as we will be using all three heuristics together during grading. You should observe a very significant reduction in the number of steps taken to reach the first full assignment.

3.2 Handling n-ary potentials (10 points)

So far, our CSPs only handle unary and binary potentials, but for course scheduling, we need potentials that involve more than two variables. In this problem, you will take one type of n-ary

constraints and reduce them to a set of binary potentials with auxiliary variables, as discussed in class.

- (10 points) Implement `get_sum_variable()`, which takes in a list of non-negative integer-valued variables and returns a variable whose value is constrained to be equal to the sum of the variables. You will need the domains of the variables passed in, which you can assume contain only non-negative integers. Use the technique of adding auxiliary variables discussed in class.

3.3 Course scheduling (50 points)

In this problem, we will apply your weighted CSP solver to the problem of course scheduling. We have scraped a subset of courses that are offered in our department. For each course in this dataset, we have information on which semesters it is offered, the prerequisites (which may not be fully accurate due to ambiguity in the listing), and the range of credits allowed. You can take a look at all the courses in `rice_cs_courses.json`. Please read `util.py`, paying particular attention to `util.Course` and `util.CourseBulletin` to understand how this information is parsed.

To specify a desired course plan, you would need to provide a profile which specifies your constraints and preferences for courses. A profile is specified in a text file (see `profile*.txt` for examples). The profile file has four sections. The first section specifies a fixed minimum and maximum (inclusive) number of credits you need to take for each semester. In the second section, you register for the semester that you want to take your courses in. For example, `register Fall2014` would sign you up for this semester. The semesters need not to be contiguous, but they must follow the exact format `FallNNNN` or `SprNNNN`. The third section specifies the list of courses that you have taken in the past using the `taken` keyword. The last section is a list of courses that you would like to take during the registered semesters, specified using `request`. Not every course listed in `request` must appear in the generated schedule. Conversely, a list of requests could potentially result in an infeasible schedule due to the additional constraints we will discuss next.

Note: there's no space between items separated by commas.

If you want to take a course in one of a specified set of semesters, use the `in` modifier. For example, if you want to take COMP310 in either Fall2013 or Fall2014, do:

```
request COMP310 in Fall2013,Fall2014
```

Another operator you can apply is `after`, which specifies that a course must be taken after another one. For example, if you want to choose COMP215 and take it after both COMP140 and COMP182, add:

```
request COMP215 after COMP140,COMP182
```

Note that this implies that if you take COMP215, then you must take or have taken both COMP140 and COMP182. In this case, we say that COMP140 and COMP182 are `prereqs` of this request. If you request course A after B, and B is a prerequisite of A (based on `CourseBulletin`), we will automatically add B to the variables. But be careful that they are **NOT** added to the request data

structure. So, in some questions below, you may need to loop over all variables added in the csp rather than courses in requests.

Finally, the last operator you can add is weight, which adds non-negative weight to each request. All requests have a default weight value of 1. Requests with higher weight should be preferred by your CSP solver.

Note that you can write multiple requests of courses in to one line such as:

```
request COMP215,COMP310,COMP410
```

But if there are constraints in that request, the constraints apply to all courses requested.

```
request COMP215,COMP310 in Fall2014,Fall2015 after COMP182 weight 5
```

It means that both COMP215 and COMP310 should be taken in Fall2014 or Fall2015, the have the same prerequisite COMP182 and they are all weighted 5. It is important to note that the request of a certain course may not be satisfied, but if it is, the constraints specified by the various operators `after, in` must be satisfied.

We have done all the parsing of the course catalog and course profile for you, so all you need to work with is the collection of `Request` objects in `Profile` and `CourseBulletin` to know when courses are offered and the number of units of courses. Your task is to take a profile and bulletin and construct a CSP. We have started you off with code in `SchedulingCSPConstructor` that constructs the core variables of the CSP as well as a basic constraint. The variables are requested courses (as mentioned above, untaken prereqs have been added to the variables automatically), and the value of such a variable is one of the semesters registered in this profile file or `None`, which indicates that that course will not be taken in any semester. We will add auxiliary variables to handle non-binary factors later. We have also implemented some basic constraints: `add_bulletin_constraints()`, which enforces that a course can only be taken if it is offered in that semester (according to the bulletin).

- (10 points) Implement the function `add_semester_constraints()`. This is when your profile specifies which semester(s) you want your requested courses to be taken in. This is not saying that one of the courses must be taken, but if it is, then it must be taken in one of the specified semesters. We have written a `verify_schedule()` function in `grader.py` that determines if your schedule satisfies all of the given constraints. Note that since we are not dealing with credits yet, it will print `None` for the number of credits of each course.
- (10 points) Now you will add the weights to your requests in `add_request_weights()`. By default, all requests have a weight of 1 regardless of whether it is satisfied or not. When a weight is explicitly specified, it should only contribute to the final weight if one of the requested courses is assigned a semester i.e. is not `None`. NOTE: Each grader test only tests the function you are asked to implement. To test your CSP with multiple constraints you can use `run_p3.py` and change the constraints that you want to add.
- (10 points) Implement the `add_prereq_constraints()` function. You must make sure that if a course is taken in a certain semester, its prerequisites must all be taken before that semester.

You should write your own function that checks the values (i.e. semesters) of the course you request and its prerequisites and make sure that the values of prerequisites are smaller than that of the course you request if not `None`.

- (10 points) Now you will add the credit constraints in `add_credit_constraints()`. You must ensure that the sum of credits per semester for your schedule are within the min and max threshold (inclusive). You should use `get_sum_variable()` here . In order for our solution extractor to obtain the number of credits, for every course, you must add a variable `(courseId, semester)` to the CSP taking on a value equal to the number of units being taken for that course during that semester. When the course is not taken during that semester, the credits should be 0. Some courses have variable credit units. For example, a course can be taken as a 3 credit course or 4 credit course. You solution should be able to assign the correct credit so that the requirement of sum of credits for a semester is met.

- (10 points) Now try to use the course scheduler for Spring 2018 (the next semester). Create your own `profile.txt` and then run the course scheduler:

  ```
  python run_p3.py profile.txt
  ```

  You might want to turn on the appropriate heuristic flags to speed up the computation. Does it produce a reasonable course schedule? Please submit your `profile.txt`; and include the schedule that your solver designed for you in `writeup.pdf`.

4 Sudoku and repair algorithms (10 points)

(Problem 6.15 in textbook) We introduced Sudoku as a CSP to be solved by a constructive algorithm such as a depth-first backtracking search augmented by forward-checking and arc-consistency. It is possible to solve Sudoku using a repair algorithm. How well would a local solver using the min-conflicts heuristic do on Sudoku problems? Would you recommend a repair-based or a constructive approach for solving Sudoku? Explain your choice.

5 The Zebra Problem (20 points)

This problem is optional for comp440 and required for comp557. This is Problem 6.7 from your textbook and it gives you experience in formulation of CSP problems and its impact on ease of solution. Consider the following logic puzzle: In five houses, each with a different color, live five different people of different nationalities, each of whom prefers a different brand of candy, a different drink and a different pet. Given the following facts, the questions to answer are: Where does the zebra live, and in which house do they drink water? Define the CSP: the variables, their domains and the constraints on the variables you have defined. Then solve the CSP either by hand, or using the constraint solver you wrote in the previous exercise.

- The Englishman lives in the red house.
- The Spaniard owns the dog.

- The Norwegian lives in the first house on the left.
- The green house is immediately to the right of the ivory house.
- The man who eats Hershey bars lives in the house next to the man with the fox.
- Kit Kats are eaten in the yellow house.
- The Norwegian lives next to the blue house.
- The Smarties eater owns snails.
- The Snickers eater drinks orange juice.
- The Ukrainian drinks tea.
- The Japanese eats Milky Ways.
- Kit Kats are eaten in a house next to the house where the horse is kept.
- Coffee is drunk in the green house.
- Milk is drunk in the middle house.