CSC242: Intro to AI
Project 2: Constraint Satisfaction

In this project, you will design and build a constraint satisfaction engine and demonstrate it on a number of constraint satisfaction problems (CSPs).

Requirements

1. You must implement the Backtracking Search algorithm for constraint satisfaction problems (AIMA 6.3 and Fig. 6.5). You do not need to implement the INFERENCE step (although you may if you want). You must have a single method or function that takes an instance of a CSP as argument and returns a solution (that is, a complete, satisfying assignment to the variables of values from their respective domains).

2. You must use your implementation to solve several CSPs described below. For each problem, you must have a method or function that constructs and returns an (or the) instance of the problem. You must pass the problem instance to your solver and print the result.

3. Your program must make it obvious what is happening when we run it. At a minimum, it must print each problem before solving it, and it must print the solution. Both of these must be readable. You may include additional information if you think it is informative. If your program prints *a lot* of information (*e.g.*, tracing), make sure that this can be turned or or off without editing your code (*e.g.*, using a command-line argument).

Constraint Satisfaction Problems

You must demonstrate your CSP solver by representing the following problems as CSPs and on the three following problems.

1. The Australia Map Coloring problem (AIMA 6.1.1) as seen in class. This problem is straightforward, only involves binary "not-equals" constraints, and is easy to solve. But you might want to challenge yourself by creating a bigger instance of the problem, for example using the map of the United States.

2. The Job Shop Scheduling problem (AIMA 6.1.2). As described in the book, this problem requires variables whose domains are integers, so that you can do math on them and compare their values. This problem uses discrete time steps and the range of possible times is also limited, as described in the book.

3. The n-Queens problem (AIMA 3.2.1, pp. 71–72) as seen in class. I suggest that you use the "complete state formulation" described in AIMA 4.1.1 (also seen in class). But note that you aren't doing heuristic search. You need to formulate the problem as a CSP and then solve it with your solver(s). Your program must ask the user for the value of n.

   You could formulate the constraints for n-Queens using "not-equals", but that will be painful. I suggest that you implement a binary "not-attacking" constraint, where "X_i not attacking X_j" is satisfied if the queen in column i is not attacking the queen in column j. It's easy to code this and, hint, it can be tested in constant time.

There are many other problems that you could try to solve with your solver(s). If you look for problems online, just be careful that you don't also look for code. . .

Suggestions for Success

Start by designing your data structures.

What are the elements of a constraint satisfaction problem? You should know. Turn them into abstract classes or interfaces. Be sure to include a class or interface representing a complete CSP (its name should probably be. . . I'll let you guess). Your implementation of the CSP algorithms will work off these abstract representations. Then you will create concrete implementations of this abstract representation to represent specific CSPs.

Next: What are the key algorithms for solving CSPs, as described in the project requirements? Create one or more classes that implement the algorithm(s) using the abstract representation of CSPs. That is, these methods should work on *any* CSP.

Implement the algorithms based on the AIMA pseudo-code. It is a very straightforward mapping. You can even use the pseudo-code as comments for your code. If your code doesn't match the pseudo-code, you probably aren't doing it right.

Next: Pick one of the required CSPs and implement the concrete classes that extend the abstract classes or implement the abstract interfaces. That is, if the abstract specification refers to `Vogon`s and `Demon`s and `Carnivore`s, you will have a class extending or

implementing `Vogon`, one for `Demon`, and one for `Carnivore`. You may also have a class extending or implementing the abstract representation of a CSP.

Next: You need to be able to create instances of that kind of CSP. That is, some code must create the different elements (`Vogon`s, `Demon`s, `Carnivore`s, etc.) and combine them into an instance of a CSP. Some problems have only one instance. Some problems may allow you to construct different instances. Regardless, you need clear constructors, methods, or functions that create any necessary instances.

You should have good constructors for all your classes so that the relationships among the parts are clear (and enforced by the Java compiler). If you're not using Java, you're on your own but you must still use good object-oriented design.

Finally, write code that creates an instance of a problem and an instance of a solver, and calls the solver to (try to) solve the problem. Your code should print out the initial state of the problem and the solution (if any) in meaningful, informative ways. Having good `toString()` methods will certainly help here.

Repeat this process for each of the required problems.

Note the benefit of using a constraint satisfaction framework for solving problems. Once you've written the abstract specification and implemented the solver(s), all you have to do to solve a problem is write the concrete instantiation of the specification for that problem. The solvers will solve all of them. Or at least they will *try* to solve all of them...

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) Any collaborators (see below)
   (b) How to build your project
   (c) How to run your project's program(s) to demonstrate that it/they meet the requirements
2. All source code for your project. Eclipse projects must include the project settings from the project folder. Non-Eclipse projects must include a `Makefile` or shell script that will build the program per your instructions, or at least have those instructions in your README.txt.

3. A completed copy of the submission form posted with the project description. Projects without this will receive a grade of 0. If you cannot complete and save a PDF form, submit a text file containing the questions and your (brief) answers.

Writeups other than the instructions in your README and your completed submission form are **not** required.

We must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better grade your will be.** It is your job to make both the building and the running of programs easy and informative for your users.

Programming Practice

Use good object-oriented design. No giant `main` methods or other unstructured chunks of code. Comment your code liberally and clearly.

You may use Java, Python, or C/C++ for this project. I recommend that you use Java. Any sample code we distribute will be in Java. Other languages (Haskell, Clojure, Lisp, *etc.*) by arrangement with the TAs only.

You may **not** use any non-standard libraries. Python users: that includes things like NumPy. Write your own code—you'll learn more that way.

If you use Eclipse, make it clear how to run your program(s). Setup Build and Run configurations as necessary to make this easy for us. Eclipse projects with poor configuration or inadequate instructions will receive a poor grade.

Python projects must use Python 3 (recent version, like 3.7.x). Mac users should note that Apple ships version 2.7 with their machines so you will need to do something different.

If you are using C or C++, you should use reasonable "object-oriented" design not a mish-mash of giant functions. If you need a refresher on this, check out the C for Java Programmers guide and tutorial. You **must** use "`-std=c99 -Wall -Werror`" **and** have a clean report from `valgrind`. Projects that do not follow both of these guidelines will receive a poor grade.

Late Policy

Late projects will **not** be accepted. Submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

Collaboration Policy

You will get the most out of this project if you write the code yourself.

That said, collaboration on the coding portion of projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC242.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the group should submit code on the group's behalf in addition to their writeup. Other group members should submit only a README indicating who their collaborators are.
- All members of a collaborative group will get the same grade on the project.

Academic Honesty

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

There is code out there for all these projects. You know it. We know it.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized

Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.