Project 2: Propositional Satisfiability Solver

During this project, you will exercise what you have learned about propositional logic, Boolean logic, and search. You will implement a program which checks whether a propositional logic formula is satisfiable—that is, which facts would need to be true for the entire formula to be true.

This assignment was originally designed by Dr. Stephen Ware, and we are fortunate to make use of the starter code that he has graciously developed and provided.

Jump to Section

SAT Framework

Before you start, you should familiarize yourself with some key concepts from the course. Propositional logic is introduced in the "Logic" lecture, which also covered Boolean Logic (which you can use to construct complex sentences/statements out of simple ones), logical equivalences (e.g., converting one statement into another one that has the same truth table), and conjunctive normal form (making a sentence "a conjunction of disjunctions" or, more colloquially, a bunch of OR statements that are all ANDed together).

Also, if you haven't done so already, you should read lots of Chapter 7 from the course textbook! Chapters 7.1-7.4 deal with logic in general, chapters 7.5-7.6 discuss some of the specific algorithms you might find useful for solving this project, as well as discussing Satisfiability problems (i.e., SAT problems) more broadly.

Just as with the Chess project before, you'll be given some code to get you started: a SAT framework. Once again, this code was generously produced and provided by Stephen Ware. It was written in Java 11 (but any version of Java after that should be able to run it just fine). To work on this project, if you have something earlier than Java 11 on your machine you will need to download the latest version of the Java Virtual Machine, install it on your computer, and ensure that Java is on your system path.

Once Java is installed, download and unzip the SAT framework from Canvas. It contains the following files and folders:

- `sat.jar`, code for representing and solving SAT problems.
- `benchmarks`, a folder containing many example SAT problems.
- `solvers`, a folder containing several example SAT solving algorithms.
- `RandomSolver.zip`, the source code for the Random SAT solver algorithm.
- `grade.bat`, a Windows batch file for grading your final submission.
- `grade.sh`, a Mac or Linux shell script for grading your final submission.

You should also check out the documentation of the SAT framework. Truly, I think looking at this documentation should be the very first thing you do (after you finish reading this assignment description document, of course)! You can access it by downloading the doc.zip file from Canvas, and then opening index.html in your favorite web browser.

Let's consider one of the example problems, cnf_2.sat, in the benchmarks folder. Feel free to open this with Sublime Text or VS Code, or whatever your favorite text editor is. It should look like this:

```
(and (or A B) (or (not A) B))
```

Which we'll rewrite here with some additional whitespace to make it more readable:

```
(and   (or A B)
       (or (not A) B))
```

Although it looks a little different from how we constructed propositional logic in lecture, I hope you can see the parallels.

In this example we have two ***symbols***: the propositional **variables** A and B.

These variables are in two **clauses**.

The first clause is: (or A B). As you might be able to infer, this is simply: "A OR B"

The second clause is: (or (not A) B). As you also might be able to infer this is: "!A OR B"

These two clauses are being ANDED together (i.e., they are in a conjunction). Thus, the whole thing put together is: "(A OR B) AND (!A OR B)"

To translate this same statement using the symbols from lecture: $(A \vee B) \wedge (\neg A \vee B)$

This logical formula is already in conjunctive normal form. That is to say: it is a conjunction of two disjunctive clauses, and each clause has two literals. Each clause is on its own line, for readability.

This formula is also **satisfiable**. That is to say, there exists an assignment of the variables that make the entire formula evaluate to true. Namely, if we set B to true, then both clauses become true, and thus the whole thing becomes true. Try it yourself and see!

Just as before you were given various chess agents to test your mettle against, this time you'll be given several different implementations of common SAT-solving algorithms. These live inside of the solvers folder that you downloaded above as .jar files. We'll talk about this in more detail below, but the five algorithms are: *brute*.jar, *dpll.jar*, *gsat.jar*, *random.jar*, and *walksat.jar*.

Also just as before, you can test these jars using commands on a terminal window. For example, to see how the WalkSAT algorithm performs on the cnf_2.sat example problem, open a terminal window, go to the folder where you unzipped the provided code, and execute the command:

```
java -jar sat.jar -a solvers/walksat.jar -p benchmarks/cnf_2.sat
```

The solver should be successful, and it should print the following output:

```
Reading file "benchmarks/cnf_2.sat"...
Solver:     WalkSAT
Problem:    cnf_2
Result:     success
Operations: 2
Time (ms):  0
Solution:   (and A B)
```

This indicates that the algorithm was successful and that it performed 2 operations while finding the solution. The solution as printed here --- (and A B) --- is saying that both A and B should be set to true. Note that this differs from what we said above (only set B to true). Setting A to true as well still solves the problem, but is unnecessary, and is thus not an optimal solution.

If you provide more than one solver or more than one problem, the program will print out a comparison of each solver's performance on each problem. A 1 means a solution was found, a 0 means a solution was not found. But if you are curious, there is more diagnostic information that can be printed as an html file! Append a -o and the name of the file you want at the end of the command to see how long each problem took to solve, the solution size, and the number of operations it took to get the answer. For example, to compare walksat and brute against cnf_1 and cnf_2 and get the full diagnostic information, you could write:

```
java -jar sat.jar -a solvers/walksat.jar solvers/brute.jar -p benchmarks/cnf_1.sat benchmarks/cnf_2.sat -o test.html
```

Which should produce a file called "test.html" that, when opened in a web browser, looks like this:

Satisfiability Benchmark Results

Problems Solved

	WalkSAT	brute	Total
cnf_1	1	1	2
cnf_2	1	1	2
Total	2	2	4

Solution Size

	WalkSAT	brute	Total
cnf_1	2	2	4
cnf_2	1	2	3
Total	3	4	7

Operations

	WalkSAT	brute	Average
cnf_1	2.00	2.00	2.00
cnf_2	2.00	2.00	2.00
Average	2.00	2.00	2.00

Time (ms)

	WalkSAT	brute	Average
cnf_1	1.00	0.00	0.50
cnf_2	0.00	0.00	0.00
Average	0.50	0.00	0.30

Which will generate a test.html file with all of the information your heart desires. These are sorted in terms of "goodness" from left to right (e.g., WalkSAT did better than Brute). Even though looking at these charts it looks like their performance was pretty close, we can see that WalkSAT had a slightly smaller total solution size than Brute did (e.g., Brute needlessly assigned a value to a variable that didn't need one, WalkSAT managed to avoid doing so).

If you like, you can also compare all of the provided solvers on every provided benchmark problem by looking at the results.html file also available on Canvas. As you can see, the dpll solver is the best overall, solving 43 problems total. See the "Grading" section below for a discussion of how the solvers are ranked.

Assignment

Your assignment is to write your own SAT solver. It will be graded based on how it performs relative to the other solvers provided with this project. You can make your own SAT solver by creating a .jar file which contains any class which extends edu.uno.ai.sat.Solver. Your class will need to override the abstract `Solve` method.

A SAT problem is made up of variables which can have one of three possible values: `true`, `false`, or `unknown`. An `Assignment` is an object that keeps track of which variables have which values. When your solver starts, it will be given an `assignment` object where all variables have the value `unknown`. Your goal is to set the variables to `true` and `false` in such a way that the whole logical formula is true.

Each time you set a variable's value (in other words, each time you call the `setValue()` method), that counts as one operation. A solver is limited to 100,000 operations per problem. Additionally, a solver is limited to 5 minutes of real-world time per problem. However, if you are writing efficient code, you should run out of operations long before you run out of time.

Your solver *must* be named according to your UNO student username. For example, because my university e-mail address is bsamuel@cs.uno.edu, my username is `bsamuel`.

Importing and Exporting an Eclipse Project

The file `RandomSolver.zip` contains the source code for the simplest (and lowest performing) solver, the Random Solver. You might want to start with Random Solver and modify it to create your own solver.

You can import the project into Eclipse by following these steps:

- Start Eclipse
- `File > Import`
- Under the General category, choose `Existing Projects into Workspace` and click `Next`.
- Choose the `Select archive file` option.
- Browse for `RandomSolver.zip`
- Click `Finish`.

Make sure to change the name of the solver from "`random`" to your student username. You should probably also rename the class to something other than `RandomSolver`. You can do that by right-clicking on the class and choosing `Refactor > Rename`.

While you are working, you can see how your solver is doing by running Test.java. This is an unofficial way to compare all solvers from inside an IDE like Eclipse, provided for your convenience. This is not how your solver will be graded (see the Grading section below for full details). When you are done, you must still export your solver as a jar file, and it will be graded as described below in the next section.

To export an Eclipse project as a `jar` file, follow these steps:

- Right click on the project and choose `Export`
- Under the `Java` category, choose `JAR` file and click `Next`.
- Choose the destination for the exported file, making sure that it is named with your student username and ends in `.jar`.
- Click `Finish`.

You must also export and submit your source code:

- Right click on the project and choose `Export`.
- Under the `General` category, choose `Archive File` and click `Next`.
- Choose the destination for the exported File, making sure that it is named with your student ID and ends in `.zip`.
- Click `Finish`.

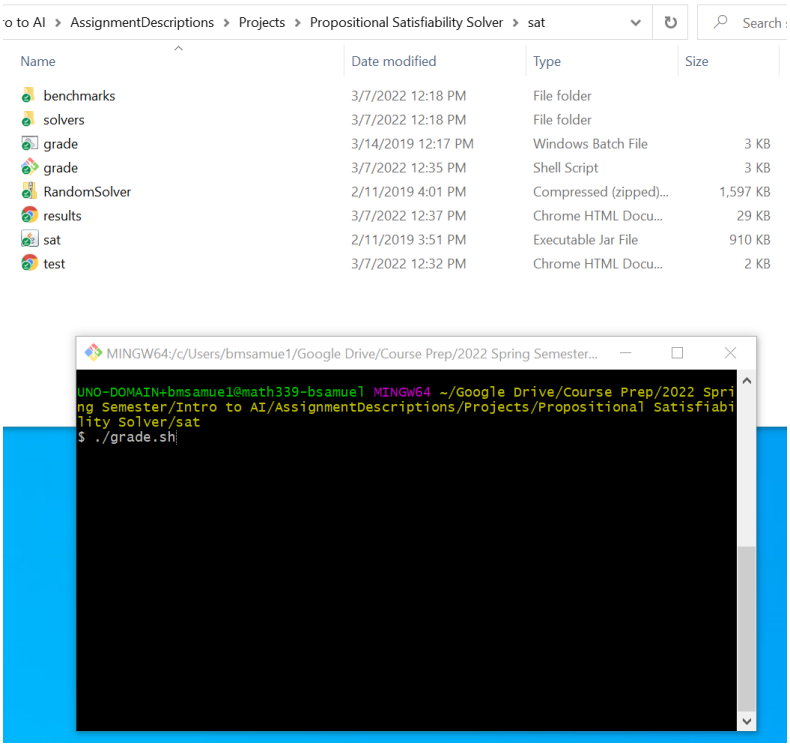
Grading

This is how I will grade your project: First, I will download your jar file to the solvers folder. Then I will run a command to see how your solver compares against all of the provided solvers. This doesn't have to be a mystery to you, you can run the command yourself before you submit your work!

**Important: make sure you add YOUR solver jar file to the "solvers" folder, and update line 3 of either grade.sh or grade.bat to be the name of YOUR solver's name (e.g., bsamuel.jar)

```
1  java -Xss10m -Xmx20g -jar sat.jar \
2  -a \
3  solvers/your.jar
4  solvers/random.jar \
5  solvers/brute.jar \
6  solvers/gsat.jar \
7  solvers/walksat.jar \
8  solvers/dpll.jar \
9  -p \
10 benchmarks/true.sat \
11 benchmarks/false.sat \
```

Once you've added your .jar file to the solvers folder: if you are using GitBash on Windows or a Mac, you'll want to run the ".sh" script. Open a terminal where your sat.jar file and the benchmarks and solvers folders are, and run the command "./grade.sh" – this will produce:



Again: I will substitute your `jar` file anywhere it says "`your.jar`" (and you should do the same if you wish to test this on your own machine).

Your grade will be determined as follows (depending on whether you are enrolled in 4525 or 5525):

Your solver…	CSCI 4525 Grade	CSCI 5525 Grade
Outperforms DPLL Solver	A+ (105%)	A (100%)
Outperforms WalkSAT Solver	A (100%)	B(80%)
Outperforms GSAT Solver	B (80%)	C (70%)
Outperforms Brute Force Solver	C (70%)	D (60%)
Outperforms Random Solver	D (60%)	F (50%)
Throws an Exception	F (50%)	F(0%)

You can see that your grade is all about "outperforming" other solvers. What does it mean for one solver to "outperform" another? Here are the criteria:

- A solver that solves more problems is ranked higher.

- If two solvers solve the same number of problems, the solver which generates shorter solutions is ranked higher. By shorter solutions, I mean solutions which involve providing assignments to fewer variables. For example, consider the cnf_2.sat example from above. Setting both `A` and `B` to true is a solution, but setting just `B` to true is a shorter solution, and thus a better solution).
- In case of a further tie, a solver which performs fewer operations (that is, fewer calls to `setValue()`) is ranked higher.
- In case of a further tie, a solver which takes less time to solve its problems is ranked higher.

The following requirements must be observed when submitting your project or it will not be graded:

- Submit your project on Canvas by the deadline. Late submissions will be eligible for partial credit.
- Submit your jar file, named for your student username (e.g., mine would be `bsamuel.jar`). If your solver does not load correctly or throws an exception, you will receive no credit for this project.
- Submit the source code for your bot as an Eclipse archive via the method described above. It must be named for your student username (e.g., `bsamuel.zip`). If I cannot import your source code, or if you export it incorrectly, I will not grade it.
- You must perform your search using the provided `Assignment` object and its `setValue` method. Do not attempt to circumvent the 100,000 operations limit (for example, by creating your own version of the Assignment class which does not obey the limit). Circumventing the 100,000 operation limit or 5 minute time limit will be considered cheating.

About the Solvers

Here is some information about the other algorithms you may find helpful when designing your solver:

- *Random* chooses a variable and randomly sets it to either true or false until it finds a solution. If any variables would have the value unknown they will be chosen first. Otherwise it picks a variable at random.
- *Brute* uses brute force. It starts with the first variable, sets it to true, and then tries to find a solution by assigning values to the remaining variables. If no solution is found, it sets the first variable to false instead and tries again. In other words, this solver will try every possible combination of true and false until it finds the solution or runs out of resources.
- *GSAT* is an implementation of the GSAT local search algorithm. It begins with a random assignment of `true` and `false` to each variable. Then it chooses a

variable and flips it to the opposite value. There is a 25% chance of choosing a variable at random. The other 75% of the time, it chooses the variable which will minimize the number of clauses that are false after the flip. AI: A Modern Approach talks about local search algorithms in section 7.6.2, though it doesn't talk about GSAT explicitly until the Historical Notes section at the end of the chapter. This is because… (see next bullet point!)

- *WalkSAT* is *very similar* to GSAT, as it is an implementation of the WalkSAT local search algorithm. Yes, both GSAT and WalkSAT take you to the same Wikipedia page. This is, again, because WalkSAT is very similar to the GSAT algorithm, but it chooses variables differently. Instead of choosing *any* variable, it *first chooses a clause that is false* and then chooses a variable **in that clause** to flip. 25% of the time the variable is chosen randomly, and the other 75% of the time it chooses the one which minimizes false clauses. AI: A Modern Approach discusses this algorithm (and even provides pseudocode for it) in section 7.6.2.
- *DPLL* is an implementation of the David-Putnam-Logemann-Loveland algorithm. It is a kind of brute force search, but after setting a variable, it simplifies the formula by eliminating *unit clauses* (clauses with just one variable) and *pure symbols* (variables that appear "the same way" in all clauses, i.e., either always negated, or always *not* negated). In other words, it can recognize some situations where a variable can only have one possible value, and it sets that variable to that value before making its next guess. This algorithm is described, and pseudocode is provided, in AI: A Modern Approach section 7.6.1.

Class Competition

Every solver which runs without throwing an exception will be entered into a class-wide competition. The first, second, and third place winners of this competition will receive bonus points on this project.

Help and Hints

Does every variable need to be set before I can know the value of the whole expression?

No! Sometimes we can know the value of the whole expression before every variable has a value. Consider the example from above, cnf_2.sat:

```
(and   (or A B)
       (or (not A) B))
```

Or that same expression using the syntax from lecture: $(A \vee B) \wedge (\neg A \vee B)$

As soon as you set `B` to `true`, the whole expression is known to be `true`. It doesn't matter what value you choose for `A`. In fact, you can just leave `A` set to `unknown` if you

want. And given that you are rewarded for your solutions being as small as possible, you likely should!

You can use `Assignment#getValue()` to check if the value of the whole expression is known yet. There are three possible cases: (1) The expression is known to be `true`, in which case you are done solving the problem. (2) The expression is known to be `false`, in which case you need to either backtrack or change one of the variables. (3) The expression is `unknown`, in which case you aren't done yet, and you need to choose a variable and set its value.

If you are implementing the DPLL algorithm, then case 1 and case 2 above are the base cases for your recursive `solve` method.

If you are implementing a local search algorithm, like GSAT or WalkSAT, the whole expression will probably start off `false`, unless you get really lucky and just happen to pick a random assignment that is a solution. The expression will stay `false` until you find a solution, and the moment it becomes `true` you are done.

How to measure how "good" it is to flip a variable.

This is relevant for those of you implementing one of the local search algorithms, like GSAT or WalkSAT.

One of the operations we need to do often is checking how good it would be to flip a value. For GSAT and WalkSAT the "best" variable to flip is the one that would lead to the fewest clauses being `false` (or, alternatively, the most clauses being `true`).

You can easily count how many clauses are false using `Assignment#countFalseClauses()`. However, this tells us how many clauses are false "right now" with the current variable assignment. What we need is a way to count how many clauses *would be* false with a different assignment.

Remember, we are asking a hypothetical question: "If I were to flip this variable, how many clauses would then be false?" We don't want to make any permanent changes to our assignment object by asking this question.

Here is the pseudocode for counting how many clauses would be false without making permanent changes to the assignment object:

```
Function countIf(Assignment a, Variable var){
 Let val be the current value of var in assignment a.
 Let opp be the opposite of val (if val is T then opp is F; if val is F then opp is T).
 Set var to opp in a.
 Let n be the number of false clauses.
 Set var to val.
 Return n.
}
```

This method temporary flips the value of a variable, counts the number of false clauses, and then flips the variable back to its original value before returning the number. No permanent changes will be made to the assignment object.

About unit clauses and pure symbols.

This is relevant for those of you who are implementing the DPLL algorithm. It is discussed in the book (section 7.6.1! Check it out, you'll be glad you did!), but here is a little sneak peek.

The brute force approach tries every possible combination of values for every variable. DPLL is just a variation of brute force, except that sometimes it can find a variable that definitely needs to be set to `true` or definitely needs to be set to `false`. There are two situations where this can happen: **unit clauses** and **pure symbols**.

A **unit clause** is a clause that only contains one unknown literal. Consider the example problem:

```
(or A B)
Assignment: A=F, B=?
```

In the above situation, the clause (or A B) is a unit clause. Even though it contains two literals, A is known to be false, so B is the only unknown literal in the clause. When you find a positive unit clause (which means there is only one unknown literal in the clause and its valence is `true`) you should set the variable to `true`. So in the above example, we should definitely set B to `true` because it can't possibly help us to try setting it to `false`. In fact, it would actively hurt us – setting B to false would make (or A B) false, thus making the entire assignment false.

```
(or A (not B)
Assignment: A=F, B=?
```

When you find a negative unit clause (which means that the clause has only one unknown literal and its valence is `false`) you should set the variable to `false`. So in the above example, we should definitely set B to `false`, because the whole clause would evaluate to false if we set to B to `true`.

A **pure symbol** is a variable whose value is `unknown` and which only ever appears in positive literals *or* only ever appears in negative literals. Consider the example problem:

```
(and    (or A B C (not D))
```

```
        (or A (not B) C (not D))
        (or (not A) (not C))
        (or (not A) D)
Assignment: A=F, B=?, C=?, D=?
```

Note that A is already set to `false`. That means that the third and fourth clauses are already known to be `true` because (`not A`) is true. Because the third and fourth clauses are already known to be `true` we can ignore them. Of the remaining variables, B, C, and D, which ones of them are pure symbols, and how does that help us?

B *is not* a pure symbol because it appears as a positive literal B in the first clause and a negative literal (not B) in the second clause.

C *is* a positive pure symbol because its value is `unknown` and it always appears as a positive literal C in every clause whose value is not yet known. We can definitely set C to `true`; there is no reason to even try setting C to `false`.

D *is* a negative pure symbol because its value is `unknown` and it always appears as a negative literal (not D) in every clause whose value is not yet known. We can definitely set D to `false`; there is no reason to even try setting D to `true`.

Do I need to keep checking for unit clauses and pure symbols every time?

Yes! Every time you set the value of a variable it might create new unit clauses and new pure symbols. That means that setting the value of a variable because it is in a unit clause might create new pure symbols, and setting the value of a pure symbol might create new unit clauses!

```
(or A B)
```

The above example is not a unit clause at first, but once you set A to `false` then it becomes a unit clause, because now there is only one unknown variable remaining: B.

Don’t forget to undo wrong assignments.

This is relevant for people who are implementing some variation of the brute force algorithm, such as DPLL.

Brute force algorithms will eventually try every possible combination of `true` and `false` for every variable until they find a solution. When you set the value of a variable and then eventually find out it was the wrong value, you need to make sure to undo the work you just did, otherwise the problem will continue to exist.

The following pseudocode for the brute force algorithm contains a bug and DOES NOT WORK:

```
Function brute(Assignment a){
        First base case: If a is true, return true.
        Second base case: If a is false, return false.
        Otherwise, if a is unknown:
                Choose a variable 'v' from the assignment 'a' whose value is unknown.
                Set v to true.
                Recursively call this function. If it succeeded, return true.
                Set v to false.
                Recursively call this function. If it succeeded, return true.
                Return false.
}
```

The reason this does not work is because sometimes we make wrong decisions, but then we forget to undo those wrong decisions once we realized they were wrong! Consider this example problem:

```
(and    (or A B)
        (or (not A) B)
        (or (not A) (not B)))
```

We start with the assignment A=?, B=? where every variable's value is `unknown`. The first unknown variable is A, so the first call to the `brute` function above chooses A and sets it to `true`. We recursively call the brute function, choose the next unknown variable, which is now B, and set it to `true`. The assignment A=T, B=T is not a solution (the third clause evaluates to false with this assignment), so we need to backtrack and make our most recent decision differently. Our most recent decision was to set B to `true`, so now we set B to `false`. Sadly, the assignment A=T, B=F is also not a solution, so we need to backtrack again, back to where we set the value of A. So now we set A to `false`… but here's the problem. Do you see it? Take a moment to think about it…

We forgot to set B back to `unknown` after we initially tried both values for it! The assignment object is currently A=F, B=F when it should be A=F, B=? The current assignment, A=F, B=F, is still not a solution, and moreover, because B isn't `unknown`, we're not going to try giving it different values. We're out of luck!

This bug happened because we forgot to set the value of B back to `unknown` after we tried both `true` and `false` and neither of them worked.

There are two ways to avoid this bug. The easy but inefficient way is to always make a clone of the assignment object before you make any changes to it. This will work, but it is inefficient because we are constantly making copies of the assignment object, which can be large.

The more efficient solution is to undo the changes we made before failing. That means that after you try setting a variable to both `true` and `false` and it doesn't work, set it back to unknown before the method fails.

Here's an important note: If you are implementing DPLL, don't forget to undo anything you did to unit clauses and pure symbols! Consider the example problem above. If we set A to `true` then the second clause becomes a unit clause, and we know that we need to set B to `false`. But, as we just saw, setting A to `true` was a bad decision, which makes it impossible to solve the problem, so eventually we're going to need to backtrack over that decision. When that happens, we have to make sure to undo the unit clause propagation where we set B to `true` or else the problem will have the same bug as before.

Here's a helpful method that you can use to make sure you don't accidentally forget to undo wrong assignments. You can essentially use this any place where you would normally call `setValue()` (as you can see, this method calls `setValue()` itself, but cleans up after itself if needed)

```
//Set a variable to a value, and if it doesn't work, undo it.
 private boolean tryValue(Assignment a, Variable var, Value val){
        //Back up the variable's current value.
        Value backup = a.getValue(var);
        //Set the variable to the given value.
        a.setValue(var, val);
        //Try to solve the problem with this new information.
        if(solve(a)){
                return true;
        }
        //If we failed to solve the problem, return the variable to its previous value.
        else{
                a.setValue(var, backup);
                return false;
        }
 }
```

Once again, there's a lot of documentation for this! Am I expected to know all of it? What packages and classes should I be focusing on?

Although you might find all of the documentation helpful, a lot of it is, once again, there to just get the project working (e.g., representations of propositional logic that you don't really need to mess with to succeed here).

In terms of what package to focus on, you can likely get away with *only* really worrying about the "edu.uno.ai.sat" package.

Package edu.uno.ai.sat

Contains objects for representing satisfiability problems.

Class Summary

Class	Description
Assignment	Represents a (full or partial) solution to a satisfiability `Problem`.
Clause	Represents one of the disjunctive clauses that make up the logical formula in a satisfiability `Problem`.
Literal	Represents one literal in a disjunctive `Clause`.
Main	The entry point for the satisfiability application.
Problem	Represents a logical formula which may or may not be satisfiable, organized into clauses, literals, and variables.
Result	The result of a `Solver`'s search for a satisfying assignment.
Settings	Project settings
Solver	The parent class of all satisfiability solving algorithms.
Variable	Represents one of the unique variables that appears somewhere in the formula of a satisfiability `Problem`.

Enum Summary

Enum	Description
Value	Represents the three possible values that a `Variable` in a satisfiability `Problem` can have: true, false, or unknown.

This package contains the classes that you will most likely be working with directly as you attempt to create a solver of your own. Within here, there are several classes that you will no doubt find helpful as well. Of these, the most important is likely the assignment class.

The **Assignment** class: has various methods to help you getting the current truth value of Variables, Literals, Clauses, and the Assignment value itself. Namely:

`Value`	`getValue()`	Returns the `Value` of the logical formula for the problem this assignment is associated with.
`Value`	`getValue(Clause clause)`	Returns the `Value` of a given clause.
`Value`	`getValue(Literal literal)`	Returns the `Value` of a given literal.
`Value`	`getValue(Variable variable)`	Returns the current `Value` of a variable in the formula.
`void`	`setValue(Variable variable, Value value)`	Sets the `Value` of the given variable.

getValue() – Given the current assignment of values of variables, this method returns a Value indicating if this assignment object is true or false or unknown. Getting your assignments to become “true” is basically the goal of the project; it means you have found an assignment of variables that solves the problem.

getValue(Clause clause) – you pass in a Clause, it tells you if, given the current assignment of variables, returns if that clause is true or false or uknown.

getValue(Literal literal) – You pass in a literal, and given the current assignment of variables, returns if that literal is true or false or unknown.

getValue(Variable variable) – You pass in a variable, and given the current assignment of variables, returns if that variable is true or false or unknown.

setValue(Variable, Value) – you pass in a variable and the value you want to set it to (either TRUE, FALSE, or UNKNOWN), and it changes the value of that variable to the new value. Invoking this method counts as "one operation" – of which you only have a finite amount of per problem!).

Of course, as you can see, those methods involve other classes and enums too (such as Literal, Variable, Clause, and Value). Familiarizing yourself with them is likely going to be helpful, but some particular things to draw your attention towards include…

The Variable class has a list of literals. This is a list of every time this literal shows up in the problem. So for example in the problem:

```
(and    (or A B)
        (or (not A) B)
        (or (not A) (not B)))
```

We have two variables (A and B), but a total of six literals. Three of those literals are A (i.e., A shows up a total of three times in this problem), and the other three are B (B also shows up three times). So again, the variable class has a reference to every literal that represents that variable. This could potentially be helpful for finding out if any given variable is a pure symbol!

The Clause class has a list of literals: This is the list of each literal that shows up in a specific clause. So, for example, in this clause

```
(or (not A) B)
```

There are two literals: the (not A) literal, and the B literal. Using this can be a helpful tool to helping you find unit clauses (remember: a unit clause doesn't mean the clause has exactly one literal, it means it has exactly one *unassigned* literal!)

The Literal class has a valence: This is how you know if the literal is a "positive" literal or a "negative" literal, i.e., whether the literal is negated or not. So for example, with this clause:

(or (not A) B)

The ‘A’ literal here would have a valence of ‘false’ because it is negated. The ‘B’ literal here would have a valence of ‘true’ because it is not negated. This also can be used to help you find pure symbols!

Value is an Enum with three values. This is used in the `Assignment` class’ “`setValue`” method to set the value of a variable. There are three possible values here: `FALSE`, `TRUE`, and `UNKNOWN`.

Although you may find other elements of the documentation helpful as well, I feel like the above is definitely what you should be focusing on to do well in this project!

And, ah, how should I get started again?

After downloading and unzipping the project from Canvas, I would open the “RandomSolver.zip” file in Eclipse. This is the source code for Random Bot, as well as the unofficial way of grading your project. I would start with that source code, and then make it “fancier.”

Although there are many paths to fanciness, I would suggest identifying an algorithm discussed in class (such as DPLL or WalkSAT) and attempting to implement it. How? By using your knowledge of the algorithm and the hints/tips listed above of good classes/methods you have at your disposal, of course!

I know Chess Bot was tricky for many! I think you’ll find that, compared to that, this is going to be at least a *little* more straightforward!

Good luck! Have fun!